Numéro d'ordre : 2 Année 1995

THÈSE

présentée devant
L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour l'obtention du titre de
DOCTEUR
Spécialité : Informatique

par **Catherine PARENT**

SYNTHÈSE DE PREUVES DE PROGRAMMES DANS LE CALCUL DES CONSTRUCTIONS INDUCTIVES

Date de soutenance : 24 janvier 1995

Rapporteurs : Robert Cori
Kees Hemerik

Jury : Luc Bougé
Pierre Casteran
Michel Cosnard (Président)
Kees Hemerik
Gérard Huet
Emmanuel Ledinot
Christine Paulin-Mohring

Thèse préparée à l'École normale supérieure de Lyon
au sein du Laboratoire de l'informatique du parallélisme,
Unité de Recherche Associée au CNRS n° 1398.

Robert Cori et Kees Hemerik ont accepté d'être rapporteurs sur cette thèse. Je leur en suis très reconnaissante et les remercie pour leur grande patience à la relecture du manuscrit. Je remercie aussi Pierre Casteran pour sa lecture si attentive du manuscrit et sa participation au jury.

Merci à Emmanuel Ledinot pour avoir accepté de faire partie du jury et pour ses nombreux commentaires tant sur la rédaction que sur l'implantation elle-même. Je remercie aussi la société DASSAULT pour le financement de cette thèse.

Je voudrais insister sur l'exceptionnel environnement de travail dans lequel j'ai eu la chance de travailler.

Tout d'abord, Christine Paulin, en tant que directeur de thèse, m'a énormément aidée et je ne pourrais jamais assez lui exprimer ma reconnaissance. Elle a toujours été présente pour me guider et je voudrais la remercier pour le temps énorme qu'elle a dû passer à relire les différentes versions du manuscrit avec une rapidité et une précision qui m'ont impressionnée.

Gérard Huet a été lui aussi toujours très présent et je le remercie pour la qualité de l'environnement qu'il a créé dans le cadre du groupe Coq. J'en profite pour remercier tous les membres de ce groupe que ce soit à Rocquencourt ou à Lyon. En particulier, Eduardo qui a subi mes différentes sautes d'humeur, Jean qui a fait une lecture très détaillée, Jean-Christophe et Judicael pour leur relecture, et Philippe qui n'a cessé de me remonter le moral. Ils ont tous été très patients avec moi et je les en remercie.

Gilles Kahn et Ranan Fraer ont relu cette thèse et je les remercie. Ranan a été formidable pour la précision avec laquelle il l'a relue et tous les commentaires qu'il a faits.

Bises à tout le LIP. Ces trois années se sont passées dans une ambiance très agréable, même si j'ai pu piquer quelques petites crises avec le café.

En particulier, merci à Luc Bougé et Michel Cosnard pour leur participation au jury.

Merci à Thibault qui est redoutable dans ses relectures, à Jean-François qui a accepté de relire l'introduction et la conclusion alors qu'il avait certainement autre chose de plus intéressant à faire (...), à Eric qui a réussi à éviter tout ça (merci à l'armée), à Richard et à ses parents (il paraît qu'il faut que ça devienne une tradition...), à Steph, Valérie et Victoria pour tout ce qu'ils savent, à Jocelyne pour son soutien "logistique", à Yves qui a l'art de savoir remonter le moral, Pierre, Afonso, Gilles, Jean-Louis et Philippe.

Merci aussi à tous ceux que j'oublie.

Enfin, merci à Xavier (merci d'avoir réussi à relire 4 pages avant de t'endormir...), mes parents et ma famille pour m'avoir permis d'être ici et m'avoir toujours soutenue et encouragée.

Table des matières

Introduction

La propriété la plus importante qu'un programme doit vérifier est d'accomplir les intentions de l'utilisateur. Une grande partie du travail de programmation est en effet consacrée à la vérification de la correction des résultats d'une exécution. Couramment, pour se convaincre de la validité de son algorithme, le programmeur l'essaie dans des cas particuliers et le modifie si les résultats produits ne correspondent pas à son attente. Quand il a essayé un nombre d'exemples qui lui semble suffisant, il estime que son programme est correct. Le temps passé à faire des tests est très souvent important par rapport au temps passé à écrire le programme. Les phases de tests sont nécessaires mais coûteuses. Il serait plus *rentable* de spécifier les programmes et de les prouver. Prouver un programme consiste à montrer par un raisonnement mathématique que le comportement induit par la spécification est bien suivi par le programme.

Spécifications de programmes

Spécifier un programme consiste à donner sous une forme explicite son comportement, c'est-à-dire ce qu'il fait, mais pas comment il le fait (voir [Hoa69]). Il s'agit d'exprimer quelles sont ses données (*entrées*), quelles propriétés elles vérifient et quels sont ses résultats (*sorties*), quelles propriétés ils vérifient. On peut prendre un exemple avec la spécification d'un algorithme de division sur des entiers naturels. Ce dernier a comme entrées deux entiers a (*dividende*) et b (*diviseur*) et calcule en sortie deux autres entiers q (*quotient*) et r (*reste*). Le quotient et le reste vérifient des propriétés par rapport à a et b à savoir $a = b*q+r$ et $b > r$. De plus, pour que la spécification ait un sens, b doit être non nul. La propriété que b doit vérifier initialement est donc $b > 0$ (nous parlons d'entiers naturels). Il faut bien noter que, comme une spécification décrit le *quoi* du programme mais pas le *comment*, elle est de façon générale valable pour plusieurs algorithmes. Par exemple, un algorithme de tri peut être spécifié par la donnée d'une liste quelconque et la sortie d'une autre liste contenant les mêmes éléments ordonnés. Une telle spécification décrit ce que fait un tri mais pas comment il le fait. En effet, la spécification sera la même pour un tri à bulles, un tri rapide ou n'importe quel autre algorithme de tri. En logique de Hoare [Hoa69] par exemple, une spécification s'exprime par une précondition sur l'état d'entrée des variables d'une part et une postcondition sur leur état de sortie d'autre part.

Preuves de programmes

Les méthodes de tests ne sont pas sûres à cent pour cent. Le programmeur n'est convaincu de la correction de son algorithme que parce qu'il estime avoir fait suffisamment de tests. L'intérêt des preuves de programmes est le suivant. Si l'on considère un programme important divisé en un ensemble de sous-programmes et si chaque sous-programme est prouvé alors chaque sous-programme a été spécifié. Le programme général est donc documenté, c'est-à-dire, pour chaque sous-programme, l'utilisateur sait comment l'utiliser et ce qu'il fait. Le programme peut alors être plus facilement portable puisque le comportement de chaque sous-partie est connu précisément.

La preuve de la spécification de la division prend en argument deux entiers a et b avec une preuve de $b > 0$, et rend deux entiers q et r avec une preuve de $(a = b*q+r) \wedge (b > r)$. Elle contient une méthode explicite de construction des entiers q et r.

Nous considérons des preuves constructives, c'est-à-dire que toute preuve contient non seulement des sous-preuves de vérification des propriétés recherchées mais aussi une méthode explicite de construction du résultat.

Preuves en théorie des types

Dans le cadre qui nous intéresse, à savoir celui de la théorie des types et plus particulièrement du Calcul des Constructions [Coq85, Coq89], une spécification est une formule logique qui établit un lien entre l'entrée et la sortie. Reprenons l'exemple de la spécification d'un algorithme de division sur les entiers naturels. La spécification informelle que nous avons donnée précédemment correspond à la formule suivante : $\forall a, b (b > 0) \rightarrow \exists q, r (a = b*q+r) \wedge (b > r)$. L'isomorphisme de Curry-Howard [How80] identifie spécification avec type, et preuve avec terme correctement typé. Il est à l'origine de systèmes de représentation et de vérification des connaissances mathématiques tels que Automath [DB80], la théorie intuitionniste des types de Martin-Löf [ML84] ou le Calcul des Constructions. Il donne un cadre uniforme pour manipuler à la fois des preuves et des programmes. Prouver une formule comme la spécification de la division, c'est donc trouver un terme correctement typé dont le type est cette formule.

Ainsi, une spécification est écrite dans le langage de la logique mathématique et prend la forme générale $\forall x P(x) \rightarrow \exists y Q(x, y)$, c'est-à-dire, pour toute entrée x vérifiant la propriété $P(x)$, il existe une sortie y vérifiant $Q(x, y)$. Pour se ramener encore au cas de la division, $P(a, b)$ est alors $b > 0$ et $Q(a, b, q, r)$ est $(a = b * q + r) \wedge (b > r)$. Ainsi, pour une spécification ayant la forme générale $\forall x P(x) \rightarrow \exists y Q(x, y)$, la preuve contient explicitement un témoin y et une méthode de construction de ce témoin. D'autre part, la preuve prend en argument une preuve de $P(x)$ et rend une preuve de $Q(x, y)$.

Extraction de programmes

Une preuve peut être vue aussi comme un programme contenant sa preuve de correction. Ainsi, reprenons l'exemple de la division. Donnons le squelette d'une preuve mathématique de la spécification $\forall a, b (b > 0) \rightarrow \exists q, r (a = b * q + r) \wedge (b > r)$. Il s'agit d'une preuve par récurrence sur a. Si a est nul, alors $q = r = 0$ et on a bien $0 = b * 0 + 0$ et $b > 0$. Sinon, on suppose qu'on connaît q et r pour le prédécesseur de a c'est-à-dire tels que $a - 1 = b * q + r$ et $b > r$, et on essaie de construire un q' et r' pour a. Il faut distinguer deux cas : soit $b \leq (r + 1)$, soit $b > (r + 1)$. Dans le premier cas, le quotient résultat q' est $q + 1$, le reste r est 0 et on a bien $a = b * (q + 1) + r$ et $b > 0$ (la première propriété vient du fait que $a - 1 = b * q + r$ et que $b > r$ et $b \leq (r + 1)$ donc $b = r + 1$; sa preuve n'est pas immédiate). Dans le second cas, le quotient résultat q' est q, le reste r' est $r + 1$ et on a bien $a = b * q + r + 1$ et $b > (r + 1)$.

La preuve contient une méthode de construction explicite de q et r mais elle prend aussi en argument une preuve de $b > 0$ et rend une preuve de $(a = b * q + r) \wedge (b > r)$ pour chacun des cas possibles pour a. Prouver la spécification revient donc à construire un programme correct vis-à-vis de cette spécification. Cependant, le programme construit est inefficace du fait de la présence de sa preuve de correction de façon interne (au niveau du programme, on ne s'intéresse par exemple pas à la justification du fait que $a = b * q + r$ pour le q et le r construit).

Pour obtenir des programmes plus réalistes, on peut distinguer la partie construction explicite du résultat de la partie preuve de correction. La première partie est la partie calculatoire de la preuve et la seconde la partie logique. Avec cette distinction, seule la partie calculatoire de la preuve peut alors être gardée en tant que programme. C'est l'extraction de programmes étudiée dans [PM89b], qui oublie les informations logiques d'une preuve pour obtenir un programme. Cette approche concerne la validation de programmes par preuve d'une spécification puis synthèse d'un programme à partir de cette preuve. On a un langage logique de développement de preuves qui est le Calcul des Constructions Inductives et un langage de programmation qui est le système ${F_\omega}^{Ind}$. Le langage de preuves est plus puissant que le langage de programmation puisqu'il permet d'écrire des propriétés sur des éléments du langage de programmation.

Dans cette thèse, nous étudions un autre aspect des preuves de programmes et même l'aspect inverse. En effet, prouver une spécification demande une certaine expertise et n'a rien d'automatique (voir le cas de la division). Une idée plus naturelle qu'extraire un programme à partir d'une preuve nous semble être de prouver directement le programme lui-même. En cela, nous rejoignons l'idée de la logique de Hoare. Celle-ci considère une spécification et un programme, et construit pas à pas une preuve du programme par analyse de sa structure. Le programme est prouvé correct quand il ne reste que des axiomes. Dans cette thèse, nous suivons la même idée mais dans un autre cadre, celui de la théorie des types. La logique de Hoare se place dans le cadre de programmes impératifs alors que nous considérons des programmes fonctionnels typés.

Dans le cas de la logique de Hoare, des règles transforment, suivant la structure du programme, la spécification en une spécification plus simple jusqu'à arriver à des axiomes. Dans notre cas, il s'agit de retrouver un terme de preuve dont le type est la spécification. Pour cela, nous allons aussi utiliser la structure du programme. En effet, nous pouvons faire plusieurs remarques sur le contenu logique d'un programme. Même si un programme n'est qu'une méthode de calcul, il contient des informations sur la preuve sous-jacente. Une structure conditionnelle ou un test dans un programme correspond au niveau logique à une analyse par cas. De même, une récursivité dans un programme correspond à l'utilisation d'un principe de récurrence dans une preuve. Si un programme appelle des sous-programmes, la preuve nécessite des sous-lemmes. Dans l'exemple de la division euclidienne, le programme ML est le suivant :

```
let rec div a b = match a with
                    0   -> (0,0)
                  | n -> let (q,r) = div (n-1) b in
                              if b<=(r+1) then (q+1,0)
                                           else (q,r+1) ;;
```

La spécification de ce programme $\forall a,b(b>0)\rightarrow\exists q,r(a=b*q+r)\wedge(b>r)$ a été vue précédemment. Analysons la structure de ce programme. On a une analyse par cas sur a. Au niveau de la preuve, on fait donc une récurrence sur a. On peut déduire deux sous-spécifications pour les deux cas $\exists q,r(0=b*q+r)\wedge(b>r)$ et $\forall n\exists q,r(n-1=b*q+r)\wedge(b>r)\rightarrow\exists q,r(n=b*q+r)\wedge(b>r)$. Ensuite, on a une analyse par cas sur $b\leq(r+1)$. Au niveau de la preuve, cela correspond aussi à une analyse par cas et on peut déduire deux sous-spécifications $(n=b*(q+1)+0)\wedge(b>0)$ et $(n=b*q+r+1)\wedge(b>r+1)$. On voit sur un tel exemple qu'un programme contient beaucoup d'informations sur sa preuve de correction vis-à-vis de sa spécification. Ceci est en fait exactement le dual de la remarque qui souligne qu'une preuve est un programme contenant trop d'informations logiques. On peut illustrer cela en donnant le programme annoté avec les sous-spécifications qui peuvent être retrouvées :

```
let rec div a b =
match a with
   0    -> {(0 = b * 0 + 0) ∧ (b > 0) }
            (0,0)
 | n -> { ∀n∃q, r(n − 1 = b * q + r) ∧ (b > r) → ∃q, r(n = b * q + r) ∧ (b > r) }
            let (q,r) = div (n-1) b in
               { ∃q, r(n = b * q + r) ∧ (b > r) }
               if b<=(r+1)
               then { (n = b * (q + 1) + 0) ∧ (b > 0) }
                    (q+1,0)
               else { (n = b * q + r + 1) ∧ (b > r + 1) }
                    (q,r+1);;
```

Synthèse de preuves

Dans cette thèse, nous développons une méthode de synthèse de preuves à partir de programmes en s'appuyant sur cette remarque. Les programmes considérés sont des programmes fonctionnels typés dans le système ${F_\omega}^{Ind}$. Les difficultés principales de cette méthode résident dans la possibilité de reconstruire complètement ou non le terme de preuve uniquement à partir du programme et de sa spécification. Le principe de base est d'analyser la structure du programme pour engendrer, à partir d'une spécification, des sous-spécifications plus simples associées à des sous-programmes plus simples eux-aussi. Le parallèle avec la logique de Hoare est donc très fort (voir le programme annoté précédent). A chaque génération de sous-spécification, nous engendrons une partie de terme de preuve correspondant à la règle utilisée. En fait, nous montrons que nous pouvons engendrer un terme de preuve partiel c'est-à-dire avec des *trous*, les trous correspondant à des lemmes restant à vérifier. Ces lemmes représentent en fait des propriétés logiques qui doivent être vérifiées pour assurer la correction du programme.

Cependant, on peut se rendre compte qu'il n'est pas trivial de reconstruire un terme de preuve uniquement à partir d'un programme et de sa spécification. Typiquement, une récursivité correspond à un principe de récurrence dans la preuve. Pratiquement, rien n'indique explicitement ou implicitement dans le programme quelle est la spécification à prouver. Dans l'exemple de la division, le programme indique une analyse par cas sur a, la preuve fait donc une récurrence sur a. La spécification est $b > 0 \rightarrow \exists q, r(a = b * q + r) \wedge (b > r)$. Doit-on appliquer la récurrence sur cette spécification, ou l'appliquer sur la spécification $\exists q, r(a = b * q + r) \wedge (b > r)$ dans le contexte $b > 0$? Sur cet exemple, les deux solutions fonctionnent. Dans le programme annoté précédent, la solution choisie est la seconde. Mais, on imagine facilement que, dans des cas plus compliqués, la première solution puisse engendrer des preuves plus compliquées que la seconde, voire impossibles, ou inversement. Il y a donc un problème de décision. Le parallèle peut être fait avec les problèmes bien connus de recherche d'invariants de boucle en logique de Hoare qui nécessitent l'intervention du programmeur.

Dans notre cas, l'intervention du programmeur va aussi être nécessaire. Cela signifie que, même si le programme contient des informations logiques sur la structure de sa preuve de correction par rapport à sa spécification, il n'en contient pas suffisamment. La difficulté principale est donc de pouvoir définir un langage de programmes annotés par des informations logiques sur des sous-spécifications de sous-programmes. Il faut de plus que ce langage soit suffisamment riche pour permettre de retrouver toutes les informations nécessaires à la preuve. Cela signifie la recherche d'un langage à mi-chemin entre un langage logique de développement de preuve comme le Calcul des Constructions Inductives et un langage de programmation comme le système ${F_\omega}^{Ind}$. Cela signifie aussi une méthode pour retrouver et réutiliser des informations logiques sur un programme.

Inversion de l'extraction

Il semble alors évident que nous recherchons un procédé inverse de l'extraction de programmes de [PM89b]. L'extraction oublie des informations logiques des preuves pour obtenir des programmes alors que nous souhaitons retrouver des informations logiques à partir de programmes et de spécifications pour reconstruire des preuves. Cependant, comme nous l'avons montré précédemment sur un exemple, les programmes écrits dans ${F_\omega}^{Ind}$ ne suffisent pas toujours et nous cherchons un langage plus riche.

Dans cette thèse, nous reprenons donc l'idée d'extraction de [PM89b] mais, non pas dans l'optique d'engendrer des programmes fonctionnels complètement épurés de toute information logique, mais plutôt dans l'optique d'engendrer des traces suffisamment complètes de preuves. Sur l'exemple de la division, l'extraction de [PM89b] extrait d'une preuve telle que la preuve informelle que nous avons donnée, un programme sans annotation (semblable à celui que nous avons présenté dans un premier temps). Dans notre optique, nous souhaitons plutôt engendrer un programme annoté (plus proche du deuxième programme exprimé et donc plus proche d'une trace de preuve). Ces traces de preuves sont des programmes annotés d'informations logiques. Nous définissons donc une fonction d'extraction faible vers des programmes annotés. Nous obtenons ainsi un nouveau langage de programmes annotés qui sont des formes de preuves condensées. Nous montrons que, à partir de programmes de ce langage et de leur spécification, nous pouvons retrouver de façon complètement déterministe des termes de preuves partiels. Ceci signifie que ces programmes sont suffisamment explicites quant à la preuve sous-jacente pour que celle-ci puisse être retrouvée automatiquement (modulo des propriétés à vérifier qui correspondent aux *trous* restant dans le terme de preuve partiel). En fait, nous prouvons de façon constructive que l'extraction faible que nous avons définie est inversible. De cette preuve, nous pouvons déduire un algorithme de reconstruction de preuves à partir de programmes dans ce nouveau langage. Nous montrons que cette méthode de reconstruction est correcte et complète (suivant une notion faible de complétude).

Le problème de ce langage est qu'il ne permet pas l'écriture de programmes suffisamment naturels. Intuitivement, les programmes sont sous une forme moitié programme moitié preuve comme vu précédemment pour l'exemple de la division. Nous proposons alors de donner une méthode heuristique qui permet, sur le modèle de la méthode déterministe pour les programmes annotés, de retrouver une preuve à partir d'un programme dans le langage de programmation ${F_\omega}^{Ind}$ de [PM89b]. Du fait des heuristiques, cette méthode n'est plus complète. Nous autorisons alors dans ${F_\omega}^{Ind}$ l'ajout d'annotations de la forme de celles du langage des programmes annotés. La méthode heuristique est modifiée pour pouvoir utiliser des annotations sur le modèle de la méthode déterministe de reconstruction. Nous avons comme conséquence directe que la méthode heuristique sur des programmes *suffisamment annotés* est complète. Nous assurons que, pour tout programme prouvable suffisamment annoté, nous retrouvons la preuve (c'est-à-dire un terme de preuve partiel avec des lemmes logiques à vérifier).

Finalement, l'algorithme heuristique a été mis en œuvre dans le système *Coq* (lui-même une implantation du Calcul des Constructions Inductives) sous la forme d'une tactique. Le passage de l'étude théorique à l'étude pratique est non trivial. De nombreux problèmes se posent lors du passage à l'implantation d'une telle méthode pour la rendre utilisable, notamment rendre possible l'écriture de programmes à peu près naturels et prendre les bonnes heuristiques. Cette tactique construit à partir d'un programme éventuellement annoté et de sa spécification un terme de preuve partiel. L'utilisateur doit prouver des lemmes logiques restants à l'aide des tactiques *Coq* usuelles.

Une bibliothèque importante d'exemples a été développée. De plus, des utilisateurs extérieurs au groupe développant le système *Coq* utilisent cette tactique. On trouvera en annexe un exemple détaillé développé dans *Coq* ainsi qu'une documentation de cette implantation.

Plan

Dans le premier chapitre, divers systèmes en théorie des types depuis le Calcul des Prédicats jusqu'au Calcul des Constructions sont présentés. Le but est de montrer l'évolution des possibilités d'expressivité de chacun pour motiver l'utilisation du Calcul des Constructions Inductives comme langage de preuve et du système ${F_\omega}^{Ind}$ comme langage de programmation. Les définitions et notations de base utilisées dans les chapitres suivants sont données. En particulier, le système F_ω et le Calcul des Constructions sont définis. Nous présentons aussi l'ajout des types inductifs au Calcul des Constructions et définissons donc ${F_\omega}^{Ind}$.

Le chapitre 2 est divisé en trois parties. La première détaille le principe et la méthode d'extraction de [PM89b] dans le Calcul des Constructions Inductives. La deuxième montre une optimisation sur cette extraction pour obtenir des programmes plus proches d'un style fonctionnel réel. La dernière partie souligne la non-adéquation de cette extraction comme fonction d'obtention de traces de preuves.

Le chapitre 3 définit une nouvelle fonction d'extraction, l'extraction faible, qui répond aux exigences présentées précédemment. Nous définissons par conséquent une nouvelle notion de typage pour les termes faiblement extraits qui correspondent aux termes d'un langage de programmes annotés. Nous montrons des propriétés similaires à celles de l'extraction de [PM89b] comme la cohérence de l'extraction faible c'est-à-dire que l'extraction faible d'un terme correctement typé dans le Calcul des Constructions Inductives est un terme correctement typé dans notre nouveau système de typage. Nous présentons ensuite pourquoi et comment cette extraction est inversible. La preuve d'inversion donne une méthode systématique et déterministe de reconstruction d'un terme de preuve à partir d'un programme et de sa spécification. Enfin, nous illustrons la méthode heuristique sur des programmes naturels par des exemples et concluons sur sa complétude.

Le chapitre 4 considère des programmes plus efficaces. Pour cela, nous détaillons une série d'*anti-optimisations* c'est-à-dire un ensemble de transformations sur des programmes ne contenant pas de redondance ou de code qui se simplifie trivialement, permettant de retrouver les structures de preuves sous-jacentes. Ce chapitre insiste aussi sur les problèmes rencontrés lors de l'implantation de la méthode.

Le chapitre 5 effectue une comparaison avec d'autres travaux en preuves de programmes.

Enfin, nous présentons en annexe A une documentation de l'implantation qui a été faite. Cette documentation est illustrée sur quelques exemples. D'autre part, l'annexe B développe un exemple complet de preuve d'un algorithme de division euclidienne.

Chapitre 1

Différents systèmes de théorie des types

Notre but est de synthétiser des preuves à partir de programmes. Dans un premier temps, il est nécessaire de motiver les choix d'un langage de preuve et d'un langage de programmation précis. C'est le but de ce chapitre. Nous présentons rapidement différents systèmes de la théorie des types. Le but est de souligner l'évolution depuis le Calcul des Prédicats jusqu'au Calcul des Constructions Inductives, qui sera la base des chapitres suivants. Dans cette évolution, nous souhaitons montrer l'enrichissement progressif d'un langage qui permet la formalisation du raisonnement mathématique. A chaque étape, nous souhaitons montrer les possibilités du langage présenté et pourquoi ce n'est pas celui qui a été choisi. Nous décrirons donc dans un premier temps, le Calcul des Prédicats et la déduction naturelle. Puis, la gradation des systèmes de types, λ-calcul, système F, système F_ω et Calcul des Constructions. Nous motiverons l'utilisation des types inductifs. Nous motiverons aussi le choix de ${F_\omega}^{Ind}$ comme langage de programmation et du Calcul des Constructions Inductives comme langage de preuve. Ce chapitre constituera donc l'ensemble des raisons qui nous ont fait choisir un cadre donné et posera les bases et les notations pour la compréhension des chapitres suivants.

1.1 Calcul des Prédicats, Déduction Naturelle

Dans un premier temps, nous décrirons brièvement le Calcul des Prédicats et la déduction naturelle. Cette section est fortement inspirée de [Dow91],[Lal90] et [Kri90] et on pourra y trouver plus de détails.

1.1.1 Calcul des Prédicats

Le Calcul des Prédicats se donne un ensemble de variables et un ensemble de symboles de fonctions et de prédicats. Dans ce calcul, il existe deux types d'objets: les termes qui sont des constructions sur les variables et les symboles de fonctions, et les formules qui sont des constructions sur les termes. Les termes peuvent être vus comme des programmes et les formules comme des spécifications.

Définition 1 *Termes :*

- *les variables sont des termes ;*
- *si $t_1, \ldots, t_n$ sont des termes et si f est un symbole de fonction d'arité n, alors $(f\ t_1 \ldots t_n)$ est un terme.*

Définition 2 *Formules du Calcul des Prédicats :*

- *si P est un prédicat et $t_1 \ldots t_n$ des termes, alors $(P\ t_1 \ldots t_n)$ est une formule ;*
- *$\perp$ est une formule ;*
- *si A et B sont des formules, alors $A \wedge B$ est une formule ;*
- *si A et B sont des formules, alors $A \vee B$ est une formule ;*
- *si A est une formule, alors $\neg A$ est une formule ;*
- *si A et B sont des formules, alors $A \rightarrow B$ est une formule ;*
- *si A est une formule, alors $\forall x.A$ est une formule ;*
- *si A est une formule, alors $\exists x.A$ est une formule.*

Les quantificateurs portent sur des variables de termes si on considère le Calcul des Prédicats du premier ordre. Ils peuvent aussi porter sur des variables de fonctions ou de prédicats si l'on considère le Calcul des Prédicats d'ordre supérieur (auquel cas les notions de termes et de formules doivent changer).

Exemples :

- Au premier ordre :
  Prenons le cas particulier de l'arithmétique. Dans ce cas, $\forall x.(x = x)$ et $(x > y) \wedge (x = z)$ sont des formules valides ;
- A l'ordre supérieur :
  Nous pouvons définir :
  - l'égalité de Leibniz : $x = y \equiv \forall P.(P\ x) \rightarrow (P\ y)$,
  - le constructeur $\wedge$: $x \wedge y \equiv \forall P.(x \rightarrow (y \rightarrow P)) \rightarrow P$,
  - le constructeur $\vee$: $x \vee y \equiv \forall P.(x \rightarrow P) \rightarrow ((y \rightarrow P) \rightarrow P)$.

Définition 3 *Variables libres et variables liées :*

- *Toute occurrence de la variable x dans les formules $\forall x.A$ et $\exists x.A$ est liée. Une variable est liée dans un terme si elle a au moins une occurrence liée dans ce terme ;*
- *L'ensemble des variables libres d'une formule est défini par récurrence :*
  - $Var(f\ t_1 \ldots t_n) = \bigcup_{i=1}^{n} Var(t_i)$,
  - $Var(\perp) = \emptyset$,
  - $Var(A \wedge B) = Var(A \vee B) = Var(A \rightarrow B) = Var(A) \cup Var(B)$.

– $Var(\forall x.A) = Var(\exists x.A) = Var(A) \setminus \{x\}$.

Exemples :

Dans la formule $\forall x.(x = x)$, x est liée. Comme conséquence, $\forall y.(y = y)$ est équivalente à la formule précédente au renommage près de x par y.

Par contre, $\forall x.(x = y)$ et $\forall y.(y = y)$ ne sont pas équivalentes. Dans la première, x est liée et y est libre. Dans la deuxième, les deux occurrences de y sont liées.

Enfin, nous définissons un mécanisme de substitution.

Définition 4 *Substitution :*
Soit A une formule, x une variable, et t un terme, $A[x \leftarrow t]$ est obtenu en substituant le terme t à toutes les occurrences libres de x dans A.

Dans l'exemple précédent $\forall x.(x = y)$, si nous substituons une variable libre x à y, nous obtenons un terme $\forall x.(x = x)$ où la première occurrence de x n'a pas la même signification que la deuxième. La sémantique de ce terme n'est pas la même que celle du précédent. Cela signifie qu'il faut effectuer un renommage pour obtenir par exemple le terme $\forall z.(z = x)$. Donc, il faut renommer les variables liées de A pour ne pas introduire de confusion au moment de la substitution.

Notation : dans toute la suite, on notera toujours $A[x \leftarrow t]$ pour la substitution de t à x dans A et on suppose que le renommage est toujours fait.

Exemple de substitution : $(x + y * z)[y \leftarrow x + y] = x + (x + y) * z$

Le Calcul des Prédicats est donc un formalisme qui permet de représenter des spécifications et des programmes. Cependant, il s'agit d'un formalisme assez pauvre car il permet uniquement de représenter des fonctions très simples. En arithmétique, on ne peut coder que des fonctions constantes ou des fonctions qui ajoutent des constantes.

1.1.2 Déduction Naturelle

Pour démontrer une formule à partir d'autres formules, on peut utiliser un système de dérivation. Un système de dérivation est un système contenant des règles d'inférence sur des termes d'un langage, représentées par des déductions d'un ou plusieurs jugements vers un autre. Ces règles d'inférence appliquées en série permettent de dériver des formules et donc de faire des preuves.

Définition 5 *Contextes et jugements :*
Un contexte est une liste de formules.
Un jugement est une expression de la forme $\Gamma \vdash A$ où Γ est le contexte d'hypothèses, $\vdash$ le symbole de déduction et A une formule.

Notations : on notera Γ, Δ la concaténation de deux contextes et $\Gamma[A]$ l'ajout d'une formule à un contexte.

Pour chaque connecteur de formules, on définit des règles d'introduction et des règles d'élimination.

Définition 6 *Règles de déduction :*

$$(Axiom) \ : \ \frac{A \in \Gamma}{\Gamma \vdash A}$$

$$(\neg - intro) \ : \ \frac{\Gamma[A] \vdash \perp}{\Gamma \vdash \neg A} \qquad (\neg - elim) \ : \ \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp}$$

$$(\perp -elim) \ : \ \frac{\Gamma \vdash \perp}{\Gamma \vdash A}$$

$$(\rightarrow -intro) \ : \ \frac{\Gamma[A] \vdash B}{\Gamma \vdash A \rightarrow B} \qquad (\rightarrow -elim) \ : \ \frac{\Gamma \vdash A \rightarrow B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

$$(\wedge - elim_1) \ : \ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$(\wedge - intro) \ : \ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad (\wedge - elim_2) \ : \ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

$$(\vee - intro_1) \ : \ \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$(\vee - intro_2) \ : \ \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad (\vee - elim) \ : \ \frac{\Gamma \vdash A \vee B \quad \Gamma \vdash A \rightarrow C \quad \Gamma \vdash B \rightarrow C}{\Gamma \vdash C}$$

$$(\forall - intro) \ : \ \frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \qquad (\forall - elim) \ : \ \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \leftarrow t]}$$

$$(\exists - intro) \ : \ \frac{\Gamma \vdash A[x \leftarrow t]}{\Gamma \vdash \exists x.A} \qquad (\exists - elim) \ : \ \frac{\Gamma \vdash \exists x.A \quad \Gamma \vdash \forall x.(A \rightarrow B)}{\Gamma \vdash B}$$

Pour la règle $(\forall - intro)$*,* x *est non libre dans* Γ*, c'est-à-dire soit lié soit n'apparaît pas dans* Γ*.*
Pour la règle $(\exists - elim)$*,* x *est non libre dans* B *et dans* Γ*.*

Cet ensemble de règles décrit la logique intuitionniste. La preuve d'une formule constitue ainsi un arbre de dérivation dont les nœuds sont les jugements (correspondant à une règle) et les feuilles sont les axiomes.

Si l'on ajoute l'axiome :

$$(classique) \ : \ \frac{}{\Gamma \vdash A \vee \neg A}$$

on a la logique classique.

Nous pouvons aussi mentionner un autre formalisme pour la dérivation de formules, le calcul des séquents de Gentzen. Il s'agit là aussi de dériver des formules dans un contexte d'hypothèses mais sous un aspect différent. En déduction naturelle, les règles de dérivation modifient uniquement les formules à droite du jugement. Dans le cas du calcul des séquents, les règles de dérivation modifient les formules aussi bien à droite qu'à gauche des jugements. Nous ne détaillerons pas le calcul des séquents car il n'entre pas dans notre propos, mais on peut consulter [Lal90, Coq91, Laf91] pour plus d'informations.

1.2 Théorie des types

1.2.1 λ-calcul simplement typé

Le λ-calcul typé est un formalisme fortement lié à la déduction naturelle. En effet, les termes du λ-calcul (λ-termes) représentent les dérivations de la déduction naturelle (à chaque nœud de l'arbre de dérivation correspond un λ-terme), et les types des λ-termes représentent les formules de la déduction naturelle. C'est l'isomorphisme de Curry-Howard [How80] :

λ-calcul typé	*déduction naturelle*
λ-terme	dérivation
type	formule
type d'un terme	conclusion d'une dérivation
abstraction	introduction de $\rightarrow$
application	élimination de $\rightarrow$

Le calcul le plus simple est le λ-calcul pur. Il s'agit en fait uniquement d'un codage. C'est uniquement une autre façon de noter les choses et qui permet ensuite un codage vers des systèmes plus puissants. De façon usuelle, on utilise la notation λx.t pour représenter la fonction mathématique usuelle $f : x \longmapsto t$. Nous utiliserons la notation $[x]t$.

Définition 7 *Termes du λ-calcul pur :*

$$T ::= x \mid (T\ T) \mid [x]T$$

Notations :

- $((T_1\ T_2)\ T_3) = (T_1\ T_2\ T_3)$;
- $[x_1]\ [x_2]\ [x_3]t = [x_1\ x_2\ x_3]t$.

Exemple :

Si nous nous plaçons dans la cadre de l'arithmétique (λ-calcul avec constantes), nous disposons d'une constante 0, d'un symbole de fonction S d'arité 1 (le successeur), d'un symbole de fonction $+$ d'arité 2 (l'addition). Nous pouvons alors définir la fonction qui à un terme associe ce terme augmenté de 1 par le terme : $[x](+\ x\ (S\ 0))$.

De même que pour le Calcul des Prédicats, le problème du renommage des variables liées se pose. Le terme $[x](x\ [y](y\ x))$ est identique au terme $[x](x\ [z](z\ x))$. On dit que les deux termes sont *α-équivalents*.

Nous pouvons définir une notion de réduction sur les termes du λ-calcul. En effet, en mathématique, lorsqu'on applique une fonction $f : x, y \longmapsto f(x, y)$ à l'argument 2, on obtient $y \longmapsto f(2, y)$ par substitution de x par 2 dans la formule de f. De même, en λ-calcul, nous pouvons définir une notion de réduction sur les termes.

Définition 8 *β-réduction et β-équivalence :*
La relation de β-réduction en un pas est la plus petite relation vérifiant :

- $([x]t\ u) \succ t[x \leftarrow u]$;
- *si* $t_1 \succ t_2$, *alors* $(t_1\ u) \succ (t_2\ u)$;
- *si* $t_1 \succ t_2$, *alors* $(u\ t_1) \succ (u\ t_2)$;
- *si* $t_1 \succ t_2$, *alors* $[x]t_1 \succ [x]t_2$.

La relation de β-réduction $\succ^*$ est la plus petite relation transitive qui contient $\succ$. Un terme est dit normal s'il ne peut plus être réduit par $\succ^*$.

La β-équivalence $=_\beta$ est la fermeture réflexive, symétrique et transitive de $\succ$. La propriété de Church-Rosser assure que deux termes β-équivalents se réduisent par $\succ^*$ en un même terme.

Exemples de réduction :

- Reprenons le terme de l'exemple précédent $[x](x\ [y](y\ x))$ et appliquons-le à une variable z. La β-réduction donne : $(z\ [y](y\ z))$;
- Prenons un autre exemple qui fait apparaître un problème de la β-réduction dans le λ-calcul pur. Il peut y avoir des chaînes infinies de β-réduction. Le terme $\delta = [x](x\ x)$ appliqué à lui-même boucle. En effet, $(\delta\ \delta) \succ (\delta\ \delta)$.

Pour résoudre ce problème des termes qui peuvent boucler par β-réduction, on introduit le λ-calcul simplement typé en ajoutant des types dans les lambdas.

Définition 9 *Termes du λ-calcul simplement typé :*

$$T ::= x \mid (T\ T) \mid [x : U]T$$

$$U ::= Set \mid X \mid type\ simple \mid U \rightarrow U$$

Les types simples sont des types prédéfinis comme les entiers, les booléens, les listes etc. La sorte Set est le type de base.

Set est un type *à part*. Il n'est pas sur le même plan que les types simples. En fait, Set est le type des types simples. Mais, ici, l'ajout de Set dans les types, alors qu'il n'est pas sur le même plan, n'est justifié que par un souci d'uniformité. La présentation est artificiellement compliquée mais se justifie par le fait que le cadre restera le même pour les autres systèmes plus compliqués dans la suite. Ainsi, dans la définition qui suit, le jugement Γ $bien\ form\acute{e}$ n'est, dans le cas du λ-calcul simplement typé, qu'une définition purement syntaxique qui n'a pas du tout besoin d'être définie de manière simultanée avec le jugement de typage.

Définition 10 *Règles du λ-calcul simplement typé :*
Nous gardons la même notation pour les contextes que pour la déduction naturelle. Nous définissons un jugement de typage $\Gamma \vdash t : P$ pour, dans Γ bien formé, t est bien typé et a le type P. Parallèlement, nous définissons un jugement Γ $bien\ form\acute{e}$ pour Γ est un environnement bien formé.

$$\overline{[]\ bien\ form\acute{e}}$$

$$\frac{\Gamma \ bien\ formé}{\Gamma[A : Set]\ bien\ formé}\ A \text{ variable "fraîche"}$$

$$\frac{\Gamma \vdash A : Set}{\Gamma[x : A]\ bien\ formé}\ x \text{ variable "fraîche"}$$

$$\frac{A : Set \in \Gamma \quad \Gamma\ bien\ formé}{\Gamma \vdash A : Set}$$

$$\frac{\Gamma \vdash A : Set \quad \Gamma \vdash B : Set}{\Gamma \vdash A \rightarrow B : Set}$$

$$\frac{x : A \in \Gamma \quad \Gamma\ bien\ formé}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : B}$$

$$\frac{\Gamma \vdash A : Set \quad \Gamma[x : A] \vdash B : Set \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : A \rightarrow B}$$

Exemple : reprenons le terme donné comme exemple pour le λ-calcul pur. Il s'écrit alors $[x : nat](+\ x\ (S\ 0))$ si nat est le type des entiers. Nous avons $0 : nat$, $S : nat \rightarrow nat$ et $+ : nat \rightarrow nat \rightarrow nat$. Donc, $[x : nat](+\ x\ (S\ 0)) : nat \rightarrow nat$.

Dans ce cas, le terme δ ne peut être défini. En effet, il faudrait définir $\delta = [x : T](x\ x)$. Mais un tel terme n'est pas typable. La première occurrence de x doit être un type fonctionnel puisqu'elle est appliquée à un argument. Donc, $T = A \rightarrow B$. La deuxième occurrence de x doit alors être de type A. Ce qui est impossible car x ne peut avoir deux types différents. On évite donc, grâce au typage, des termes qui peuvent boucler.

Le λ-calcul simplement typé est le plus simple des systèmes de types existants. Il a aussi la plus faible expressivité en termes de fonctions représentables. Il ne permet de définir que des fonctions arithmétiques très simples. Dans la suite, nous allons montrer comment augmenter cette expressivité en modifiant le système de types considéré avec de nouvelles constructions et de nouvelles règles [Hue91, Hue92, Gal91].

1.2.2 λ-calcul polymorphe : le système F

On peut représenter la fonction identité par $[x : A]x$ en λ-calcul simplement typé. C'est la fonction identité sur les objets de type A. Mais, on aimerait pouvoir définir une fonction identité pour un ensemble de types et non pas une fonction identité pour chaque type. On voudrait une fonction paramétrée par le type et donc, on voudrait pouvoir autoriser la quantification sur les types eux-mêmes et non plus seulement sur les variables de termes. C'est le polymorphisme. Le terme $[A : Set][x : A]x$ est alors l'identité polymorphe. L'intérêt principal de la quantification sur les types est surtout de pouvoir coder des types de données (comme les listes, les booléens ou les entiers) et des fonctions polymorphes. On enrichit ainsi énormément le langage. Nous devons donc nous placer dans un autre système de types qui autorise le polymorphisme, par exemple, le système F de Girard [Gir72].

Pour rester uniforme, nous gardons la même notation pour les abstractions sur les types que pour les abstractions sur les termes. Dans d'autres présentations, on peut trouver la syntaxe $\Lambda S.t$ pour les abstractions sur les types.

Définition 11 *Termes du système F :*

$$T ::= x \mid (T\ T) \mid [x:U]T \mid (T\ U)$$

$$U ::= Set \mid X \mid U \rightarrow U \mid \forall X.U$$

Comme nous l'avons dit plus haut, le système F permet par rapport au λ-calcul simplement typé de rendre les objets polymorphes. Pour passer du λ-calcul simplement typé au λ-calcul polymorphe (F), nous ajoutons les trois règles suivantes aux règles de typage.

Définition 12 *Règles supplémentaires pour le système F :*

$$\frac{\Gamma[A:Set] \vdash B : Set}{\Gamma \vdash \forall A.B : Set}$$

$$\frac{\Gamma[A:Set] \vdash t : B}{\Gamma \vdash [A:Set]t : \forall A.B}$$

$$\frac{\Gamma \vdash t : \forall A.B \qquad \Gamma \vdash C : Set}{\Gamma \vdash (t\ C) : B[A \leftarrow C]}$$

Exemples : nous donnons des exemples de termes tirés de [Hue91] qui peuvent s'écrire dans F mais pas dans le λ-calcul simplement typé.

- L'identité polymorphe s'écrit dans le système F $[A:Set][x:A]x$ et est de type $\forall A.A \rightarrow A$;
- Nous pouvons définir des types de base comme les entiers, les booléens et les listes paramétrées par un type A. La flèche $\rightarrow$ est associative à droite.
  - $bool \equiv \forall A.A \rightarrow A \rightarrow A$ avec les constructeurs $False \equiv [A:Set][x:A][y:A]x$ et $True \equiv [A:Set][x:A][y:A]y$. On a $True : bool$ et $False : bool$.
  - $nat \equiv \forall A.(A \rightarrow A) \rightarrow (A \rightarrow A)$ avec les constructeurs $O \equiv [A:Set][s:A \rightarrow A][z:A]z$ et $S \equiv [n:nat][A:Set][s:A \rightarrow A][z:A](s\ (n\ A\ s\ z))$. On a $O : nat$ et $S : nat \rightarrow nat$.
  - $list \equiv \forall B.(A \rightarrow B \rightarrow B) \rightarrow (B \rightarrow B)$ avec les constructeurs $nil \equiv [B:Set][c:A \rightarrow B \rightarrow B][n:B]n$ et $cons \equiv [x:A][l:list][B:Set][c:A \rightarrow B \rightarrow B][n:B](c\ x\ (l\ B\ c\ n))$. On a $nil : list$ et $cons : A \rightarrow list \rightarrow list$.
- Nous pouvons définir le produit de types $A * B$ pour deux types A et B donnés mais pas le constructeur $*$ lui-même :

$$A * B \equiv \forall C.(A \rightarrow B \rightarrow C) \rightarrow C$$

Le pouvoir d'expressivité des fonctions représentables dans ce système a donc considérablement augmenté. Il s'agit de toutes les fonctions de l'arithmétique du second ordre. Cependant, nous n'avons pas encore l'expressivité maximale, à savoir les fonctions de l'arithmétique d'ordre supérieur.

1.2.3 Ordre supérieur : le système F_ω

Nous voudrions pouvoir généraliser cette idée en autorisant des fonctionnelles de types et non plus seulement des abstractions de types. Le but est de pouvoir écrire des opérateurs sur les types (comme le constructeur du produit). C'est le passage à l'ordre supérieur dont un représentant est le système F_ω. Pour passer du λ-calcul polymorphe au système F_ω, nous ajoutons une nouvelle sorte. En effet, puisque nous voulons faire des fonctionnelles de types, nous généralisons *Set* en un ensemble d'arités. La sorte des arités est *TypeSet* (en particulier, on a $Set : TypeSet$) et les arités sont construites à partir de *Set* et de produits sur des types. Nous devons donc ajouter toutes les règles nécessaires au traitement de cette nouvelle structure.

Définition 13 *Termes du système F_ω :*

$$T ::= x \mid (T\ T) \mid [x : U]T \mid (T\ U)$$

$$U ::= Set \mid X \mid [X : U]U \mid (U\ U) \mid (X : U)U$$

$U \to U$ est une abréviation pour $(X : U)U$ utilisée lorsque X n'apparaît pas dans le second U.

Exemple : des types dont la syntaxe est valide sont : $(A : Set)A$, $A \to Set$ ....

Les changements étant assez importants, nous redonnons la totalité des règles.

Définition 14 *Le système F_ω :*

$$\frac{}{[]\ bien\ form\acute{e}}$$

$$\frac{\Gamma \vdash A : s}{\Gamma[x : A]\ bien\ form\acute{e}} \quad x \text{ variable “fraîche” et } s \in \{Set, TypeSet\}$$

$$\frac{}{\Gamma \vdash Set : TypeSet}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[A : T] \vdash B : s'}{\Gamma \vdash (A : T)B : s'} \quad (s, s') \in \{(Set, Set), (TypeSet, Set), (TypeSet, TypeSet)\} \tag{1.1}$$

$$\frac{x : A \in \Gamma \quad \Gamma\ bien\ form\acute{e}}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash (x : A)B : s \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : (x : A)B} \quad s \in \{Set, TypeSet\} \tag{1.2}$$

$$\frac{\Gamma \vdash t : (x : T)B \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : B[x \leftarrow t']}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$$

La règle de conversion est nécessaire dans la pratique pour pouvoir appliquer les types construits à des fonctionnelles.

Exemple :

Reprenons l'exemple du produit. Un seul constructeur pour le produit de deux types quelconques peut maintenant être défini :

$$* \equiv [a : Set][b : Set](c : Set)(a \to b \to c) \to c.$$

On a bien une fonctionnelle à deux arguments sur les types.

Avec ce système, nous pouvons donc maintenant coder toutes les fonctions de l'arithmétique d'ordre supérieur. Il semble que ce soit tout à fait satisfaisant comme langage de programmation puisqu'on peut exprimer toutes ces fonctions. Par contre, en tant que langage de preuve, on est encore limité car, si on peut écrire des programmes qui conviennent par leur pouvoir d'expressivité, on ne peut pas exprimer de propriétés sur ces programmes.

1.2.4 Types dépendants : le Calcul des Constructions

Nous savons jusque-là faire dépendre des termes par rapport à des types (on a aussi bien sûr les dépendances termes-termes et types-types). Nous aimerions maintenant faire dépendre des types par rapport à des termes. L'intérêt est de pouvoir exprimer des propriétés sur des termes (ou des programmes) et de définir des structures de données plus fines. Par exemple, on voudrait pouvoir définir le type des entiers pairs ou le type des listes de longueur n. C'est le Calcul des Constructions. Avec cette possibilité supplémentaire, nous pourrons considérer ce langage comme langage de preuve.

Définition 15 *Termes du Calcul des Constructions :*

$$T ::= x \mid (T\ T) \mid [x:U]T \mid (T\ U)$$

$$U ::= Set \mid X \mid [X:U]U \mid (U\ U) \mid (X:U)U \mid (U\ T)$$

Les seules règles à modifier sont les règles 1.1, 1.2 pour lesquelles nous rajoutons les possibilités suivantes.

Définition 16 *Modifications des règles pour le Calcul des Constructions :*

$$\frac{\Gamma \vdash A : Set \quad \Gamma[x:A] \vdash B : TypeSet}{\Gamma \vdash (x:A)B : TypeSet}$$

$$\frac{\Gamma \vdash A : Set \quad \Gamma[x:A] \vdash B : TypeSet \quad \Gamma[x:A] \vdash t : B}{\Gamma \vdash [x:A]t : (x:A)B}$$

Exemple :

Nous pouvons maintenant axiomatiser des logiques [Hue91]. Par exemple, nous pouvons construire le prédicat d'égalité à zéro sur la signature de l'arithmétique.

$$[nat : Set; 0 : nat; S : nat \rightarrow nat; Eq : nat \rightarrow nat \rightarrow Set] \vdash [n : nat](Eq\ n\ 0) : nat \rightarrow Set$$

Nous pouvons remarquer maintenant que nous avons évolué progressivement à travers ces quatre systèmes et qu'ils sont très ressemblants. On peut utiliser une présentation uniforme des systèmes qu'on appelle les PTS signifiant *Pure Type Systems* [Bar91]. Il s'agit de PTS fonctionnels. Tous les systèmes sont présentés avec un même ensemble de règles, une règle paramétrique ainsi que des axiomes spécifiques à chaque système. Les règles sont sous la forme suivante :

Définition 17 *Pure Type Systems :*

– *Règles générales à tous les systèmes :*

$$\frac{}{[]\ bien\ formé}$$

$$\frac{\Gamma \vdash A : s}{\Gamma[x : A]\ bien\ formé} \quad x\ variable\ “fraîche”$$

$$(variable)\ :\ \frac{\Gamma \vdash A : s}{\Gamma[x : A] \vdash x : A}$$

$$(affaiblissement)\ :\ \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma[x : C] \vdash A : B}$$

$$(application)\ :\ \frac{\Gamma \vdash t : (x : A)B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : B[x \leftarrow t']}$$

$$(abstraction)\ :\ \frac{\Gamma[x : A] \vdash t : B \quad \Gamma \vdash (x : A)B : s}{\Gamma \vdash [x : A]t : (x : A)B}$$

$$(conversion)\ :\ \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$$

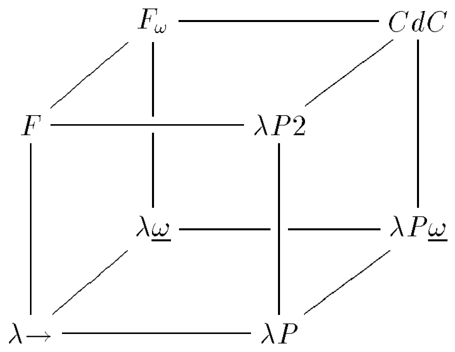
– *Règles spécifiques :*

$$(axiome)\ :\ \frac{}{[] \vdash s : k}$$

$$(produit)\ :\ \frac{\Gamma \vdash A : s_1 \quad \Gamma[x : A] \vdash B : s_2}{\Gamma \vdash (x : A)B : s_2}$$

Chaque système existant peut être caractérisé en donnant uniquement les règles axiomes (une pour chaque sorte s de type k) et l'ensemble des couples (s_1, s_2) possibles pour la règle spécifique. On a des systèmes pour chaque ensemble de couples possible sachant que s_1 et s_2 sont dans l'ensemble $\{Set, TypeSet\}$. Le λ-calcul simplement typé est représenté par l'axiome $Set : TypeSet$ et par $\{Set, Set\}$, le système F par l'axiome $Set : TypeSet$ et par $\{(Set, Set), (TypeSet, Set)\}$, le système F_ω par l'axiome $Set : TypeSet$ et par $\{(Set, Set), (TypeSet, Set), (TypeSet, TypeSet)\}$ et le Calcul des Constructions par l'axiome $Set : TypeSet$ et par $\{(Set, Set), (TypeSet, Set), (TypeSet, TypeSet), (Set, TypeSet)\}$.

On peut répartir ces systèmes sur un cube, le cube de Barendregt [Bar91]. Les arêtes ont une signification précise et correspondent à une inclusion. Quand on va vers le haut du cube, on ajoute le polymorphisme. Quand on va sur la droite, on ajoute les types dépendants. Quand on va vers l'arrière du cube, on passe à l'ordre supérieur.



Nous avons donc maintenant un langage qui nous permet d'exprimer des propriétés sur des termes grâce aux types dépendants. Cela signifie que le Calcul des Constructions peut se voir comme un langage de preuve permettant d'exprimer des propriétés sur les programmes du système F_ω. C'est ce que nous considérerons dans toute la suite.

1.2.5 Calcul des Constructions avec Types Inductifs

Le Calcul des Constructions est le système le plus expressif puisqu'il est à l'extrémité supérieure droite du cube de Barendregt. Nous allons cependant ajouter une construction supplémentaire à ce calcul. En effet, on peut définir des types de données dans le calcul tel qu'il a été présenté jusque là. Mais ce n'est pas souvent de la façon la plus naturelle. L'introduction des types inductifs permet d'exprimer des types de données sous une forme plus courante. La description qui suit est fortement inspirée de [PM93]. Il faut tout de même noter qu'ici nous présentons une version des types inductifs avec paramètres explicites qui correspond plus à l'implantation dans le système *Coq*[DFH+93]. Dans [PM93], les types inductifs sont présentés sans paramètres, ceux-ci étant traités comme des abstractions.

Nous définissons tout d'abord la notion d'arité.

Définition 18 *Arité :*
Une arité est un terme qui suit la syntaxe suivante, où M est un terme quelconque :

$$Ar ::= Set \mid (x : M)Ar$$

Nous introduisons un nouveau type de termes pour les types inductifs.

Définition 19 *Types inductifs :*

$$T ::= \mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : Ar)\{c_1 : C_1 | \ldots | c_m : C_m\}$$

Ar est une arité. Les p_i sont les paramètres du type inductif, n le nombre de paramètres et les c_i sont les constructeurs du type inductif.

Cette représentation n'est pas tout à fait celle qui correspond à l'implantation dans le système *Coq*. Dans le cas de l'implantation, on trouve $(X\ p_1 \ldots p_n)$.

Exemples :

- Le type des listes sera défini par :

$$list := \mathsf{Ind}[A : Set](X : Set)\{nil : X |\ cons : A \rightarrow X \rightarrow X\}$$

- Le constructeur de formules $\exists$ sera défini par :

$$sig := \mathsf{Ind}[A : Set; P : A \rightarrow Set](X : Set)\{exist : (x : A)(P\ x) \rightarrow X\}$$

- Le constructeur de formules $\wedge$ sera défini par :

$$and := \mathsf{Ind}[A : Set; B : Set](X : Set)\{conj : A \rightarrow B \rightarrow X\}$$

- Le constructeur de formules $\vee$ sera défini par :

$$or := \mathsf{Ind}[A : Set; B : Set](X : Set)\{or\_introl : A \rightarrow X |\ or\_intror : B \rightarrow X\}$$

- Un exemple de type inductif avec une arité différente de *Set* est la définition de la relation qui caractérise la fonction d'Ackermann :

$$\begin{aligned}
Ack \quad := \quad & \mathsf{Ind}(X : nat \rightarrow nat \rightarrow nat \rightarrow Set) \\
& \{Ack0 : (n : nat)(X\ 0\ n\ (S\ n)) \\
& |Ackn0 : (n, p : nat)(X\ n\ (S\ 0)\ p) \rightarrow (X\ (S\ n)\ 0\ p) \\
& |AckS : (n, m, p, q : nat)(X\ (S\ n)\ m\ q) \rightarrow (X\ n\ q\ p) \rightarrow (X\ (S\ n)\ (S\ m)\ p)\}
\end{aligned}$$

Les types des constructeurs suivent la syntaxe suivante :

Définition 20 *Syntaxe des types des constructeurs :*

$$Co ::= X \mid (Co\ m) \mid (P \rightarrow Co) \mid (x : M)Co$$

avec X n'apparaissant pas dans m ni dans M, et P strictement positif par rapport à X.

Définition 21 *Un terme strictement positif par rapport à X a la syntaxe suivante :*

$$Pos ::= X \mid (Pos\ m) \mid (x : M)Pos$$

avec X n'apparaissant ni dans m ni dans M.

Exemples :

- Le terme $A \rightarrow X \rightarrow X$ est est un constructeur par rapport à X; $A \rightarrow X$ est positif par rapport à X ;
- Un exemple simple de type négatif par rapport à X est $X \rightarrow A$.

Notations :

On note *constructeur(C_i, X)* pour dire C_i est un constructeur par rapport à X.

On utilise souvent une notation vectorielle pour des suites de variables. On note $\vec{a}$ pour $a_1 \dots a_n$ et $(\vec{a} : \vec{A})t$ pour $(a_1 : A_1)\dots(a_n : A_n)t$.

Nous introduisons une règle de typage pour les types inductifs.

Définition 22 *Règle de typage pour les types inductifs :*

$$\frac{\begin{array}{c}(\forall i = 1 \dots m) \\ \Gamma[p_j : P_j] \vdash A : s \quad \Gamma[p_j : P_j; X : A] \vdash C_i : s' \quad \textit{constructeur}(C_i, X)\end{array}}{\Gamma \vdash \textsf{\textit{Ind}}[p_1 : P_1; \dots; p_n : P_n](X : A)\{c_1 : C_1| \dots |c_m : C_m\} : (p_1 : P_1)\dots(p_n : P_n)A}$$

avec $A = (\vec{x} : \vec{A})s'$.

Exemple : le type du type inductif des listes est $Set \rightarrow Set$.

Nous avons aussi un nouveau terme pour les constructeurs des types inductifs.

Définition 23 *Constructeurs :*

$$T ::= \textsf{\textit{Constr}}(i, T)$$

Nous introduisons une règle de typage pour les constructeurs.

Définition 24 *Règle de typage pour les constructeurs :*

$$\frac{\Gamma \vdash Ind \equiv \textit{\textsf{Ind}}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\} : (p_1 : P_1) \ldots (p_n : P_n)A \quad 1 \leq i \leq m}{\Gamma \vdash \textit{\textsf{Constr}}(i, Ind) : (p_1 : P_1) \ldots (p_n : P_n)C_i[X \leftarrow (Ind\ p_1 \ldots p_n)]}$$

Exemple : le type du constructeur nil des listes est $(A : Set)(list\ A)$. Celui de $cons$ est $(A : Set)A \rightarrow (list\ A) \rightarrow (list\ A)$.

Il reste maintenant à pouvoir exprimer l'élimination sur des définitions inductives. En effet, on souhaite pourvoir définir des fonctions par induction structurelle sur un terme (dont le type est un type inductif). Pour cela, on a besoin d'un schéma de récursion primitive. C'est ce qui correspond à la notion d'élimination. Pour plus de détails, voir [PM93, PC89].

Nous définissons un nouveau type de termes pour l'élimination.

Définition 25 *Élimination :*

$$T ::= \textit{\textsf{Elim}}(c, Q)\{f_1| \ldots |f_n\}$$

où c est le terme sur lequel on fait l'élimination, Q le type du terme d'élimination et les f_i les différentes branches de l'élimination.

Pour pouvoir donner la règle de typage pour l'élimination dépendante, on a besoin de définir un nouveau type.

Définition 26 *Soient s et s' deux sortes, $A \equiv (\vec{x} : \vec{A})s$, X une variable de type A, Q une variable de type $(\vec{x} : \vec{A})(X\ \vec{x}) \rightarrow s'$, C un type de constructeur par rapport à X et c un terme de type C, nous définissons un nouveau type noté $C\{X, Q, c\}$ par récurrence sur C :*

$$(P \rightarrow C)\{X, Q, c\} = (p : P)P\{X, Q, p\} \rightarrow C\{X, Q, (c\ p)\} \quad \textit{si } X \textit{ strictement positif dans } P$$
$$(x : M)C\{X, Q, c\} = (x : M)C\{X, Q, (c\ x)\} \quad \textit{si } X \textit{ n'apparaît pas dans } M$$
$$(X\ \vec{a})\{X, Q, c\} = (Q\ \vec{a}\ c)$$

Remarque : le premier cas de la définition précédente correspond à un cas particulier de type de constructeur.

Alors si I représente $\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}$ avec $A \equiv (\vec{x} : \vec{A})s$, si s' est une sorte et si nous avons deux vecteurs $\vec{a} : \vec{A}$ et $\vec{p} : \vec{P}$, nous avons la règle de typage (gardée par s') :

$$\frac{\Gamma \vdash c\ :\ (I\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{p}\ \vec{x}) \rightarrow s' \quad \Gamma \vdash f_i\ :\ \overset{(\forall i = 1 \ldots m)}{C_i\{(I\ \vec{p}), Q, (\mathsf{Constr}(i, I)\ \vec{p})\}}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1| \ldots |f_m\}\ :\ (Q\ \vec{a}\ c)} \quad (Dep_{s,s'})$$

Pour pouvoir donner la règle de typage pour l'élimination non-dépendante, nous avons besoin de définir un nouveau type.

Définition 27 *Soient s et s' deux sortes, $A \equiv (\vec{x} : \vec{A})s$, X une variable de type A, Q une variable de type $(\vec{x} : \vec{A})s'$, C un type de constructeur par rapport à X et c un terme de type C, nous définissons un nouveau type $C\{X, Q\}$ par récurrence sur C :*

$$(P \rightarrow C)\{X, Q\} = P \rightarrow P\{X, Q\} \rightarrow C\{X, Q\} \quad \textit{si } X \textit{ strictement positif dans } P$$
$$(x : M)C\{X, Q\} = (x : M)C\{X, Q\} \quad \textit{si } X \textit{ n'apparaît pas dans } M$$
$$(X\ \vec{a})\{X, Q\} = (Q\ \vec{a})$$

Alors si I représente $\mathsf{Ind}[p_1 : P_1; \ldots ; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}$ avec $A \equiv (\vec{x} : \vec{A})s$, si s' est une sorte et si nous avons deux vecteurs $\vec{a} : \vec{A}$ et $\vec{p} : \vec{P}$, nous avons la règle de typage (gardée par s') :

$$\frac{\Gamma \vdash c \ : \ (I\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q : (\vec{x} : \vec{A})s' \ \ \Gamma \vdash f_i \ : \ \overset{(\forall i = 1 \ldots m)}{C_i\{(I\ \vec{p}), Q\}}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1| \ldots |f_m\} \ : \ (Q\ \vec{a})} \quad (NoDep_{s,s'})$$

Situons précisément les éliminations qui peuvent être utilisées. En effet, que ce soit l'élimination dépendante ou l'élimination non-dépendante, elles sont paramétrées par un couple (s, s'). Dans ce chapitre, nous considérons un PTS avec une seule sorte $Set : TypeSet$. Dans les chapitres suivants, nous allons devoir distinguer deux sortes de termes : les termes avec contenu calculatoire et les termes sans contenu calculatoire (comme nous l'avons indiqué dans l'introduction). Cette distinction se fait en ajoutant une sorte dans le calcul, la sorte $Prop : Type$.

Les éliminations utiles autorisées sont les suivantes : $Dep_{Set,Set}$, $Dep_{Set,Prop}$, $NoDep_{Prop,Prop}$, $NoDep_{Set,Type}$. Les trois premières sont appelées éliminations faibles, la dernière élimination forte.

Avec ces quatre règles ajoutées à celles du Calcul des Constructions standard, on a le Calcul des Constructions avec types inductifs. Par contre, le système F_ω avec types inductifs (noté ${F_\omega}^{Ind}$) est représenté par les règles standards du système F_ω plus la règle du type inductif, celle du constructeur et celle de l'élimination non-dépendante (Set, Set). En effet, l'élimination dépendante n'a pas de sens au niveau de ${F_\omega}^{Ind}$, puisqu'il n'y a pas de dépendance des types envers les termes.

Notations du système *Coq* :

Nous présentons ici la syntaxe *Coq* [DFH+93] pour les définitions de certains types inductifs ainsi que pour les éliminations. Ceci nous permettra d'utiliser cette syntaxe plus légère dans les chapitres suivants dans des exemples.

L'expression `A+{B}` est la syntaxe pour $(sumor\ A\ B)$. $sumor$ est une variante de *or* qui sera détaillée dans le chapitre suivant.

L'expression `{x:A|P(x)}` est la syntaxe pour $(sig\ A\ [x : A](P\ x))$.

Enfin, l'expression `<P>Match c with f1...fn end` est la syntaxe pour $\mathsf{Elim}(c, P)\{f_1| \ldots |f_n\}$.

Exemples :

- Pour illustrer ces définitions, nous pouvons donner l'exemple d'une élimination dépendante sur une liste paramétrée par un type. I est alors le type inductif $list$ vu plus haut. Si on a un paramètre $A : Set$, le prédicat d'élimination Q est alors de type $(list\ A) \to Set$. On a deux cas d'élimination f_1 et f_2 dont les types sont calculés de la façon suivante :

$$\begin{array}{lll}
f_1 & : & X\{(list\ A), Q, (nil\ A)\} \\
 & = & (Q\ (nil\ A)) \\
f_2 & : & (A \to X \to X)\{(list\ A), Q, (cons\ A)\} \\
 & = & (x : A)(X \to X)\{(list\ A), Q, (cons\ A\ x)\} \\
 & = & (x : A)(l : (list\ A))X\{(list\ A), Q, l\} \to X\{(list\ A), Q, (cons\ A\ x\ l)\} \\
 & = & (x : A)(l : (list\ A))(Q\ l) \to (Q\ (cons\ A\ x\ l))
\end{array}$$

On obtient donc bien les types correspondants aux deux cas d'une élimination sur une liste paramétrique, ce qui illustre la définition 26.

– Nous pouvons illustrer la notion d'élimination forte et son utilité sur un exemple. Supposons qu'on veuille définir une fonction d'égalité (équivalente à celle de Leibniz qui est utilisée dans le système *Coq* mais définit différemment) qui ait de bonnes propriétés d'injection. Typiquement, on souhaite une fonction dans *Prop* qui rende $True : Prop$ si deux entiers sont égaux et $False : Prop$ sinon. Une telle fonction se définit de la façon suivante :

```
[n,m:nat]<Prop>Match n with
                <Prop>Match m with
                      True
                      [m':nat][H:Prop]False
                end
                [n':nat][H:Prop]<Prop>Match m with
                                       False
                                       [m':nat][H':Prop]H'
                                 end
         end
```

avec la syntaxe *Coq*. Une telle définition n'est possible que si l'on autorise l'élimination forte. En effet, pour chacune des éliminations précédentes, Q est de type $Type$, $n : nat$, $m : nat$ et $nat : Set$.

Enfin, il reste à définir une notion de réduction pour les types inductifs. C'est la ι-réduction. Elle augmente la β-réduction. Pour définir la ι-réduction, on a besoin d'une définition préalable.

Définition 28 *Soient s une sorte, I le type inductif $\mathsf{Ind}[\vec{p} : \vec{P}](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}$ avec $A \equiv (\vec{x} : \vec{A})s$, X une variable de type A, C un type de constructeur par rapport à X, Q un terme tel que $Q = (\vec{x} : \vec{Q})(X\ \vec{m})$, F et f deux termes, nous définissons un nouveau terme $C[X, F, f]$ par récurrence sur C :*

$$(Q \to C)[X, F, f] = [q : Q]C[X, F, (f\ q\ [\vec{x} : \vec{Q}](F\ \vec{m}\ (q\ \vec{x})))] \quad \textit{si } X \textit{ strictement positif dans } Q$$
$$((x : M)C)[X, F, f] = (x : M)C[X, F, (f\ x)] \quad \textit{si } X \textit{ n'apparaît pas dans } M$$
$$(X\ \vec{a})[X, F, f] = (f\ \vec{a})$$

Alors, nous pouvons définir la ι-réduction.

Définition 29 *ι-réduction :*

Soit s' une sorte, I le type inductif $\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}$, Q un terme et $\vec{f}$ une séquence de termes de longueur m, nous définissons :

$$Fun\_Elim(I, Q, \vec{f}, \vec{p}) = [\vec{x} : \vec{A}][c : (I\ \vec{p}\ \vec{x})]\mathsf{Elim}(c, Q)\{f_1| \ldots |f_n\}$$

La règle de réduction est alors :

$$\mathsf{Elim}((\mathsf{Constr}(i, I)\ \vec{p}\ \vec{m}), Q)\{f_1| \ldots |f_n\} \succ_\iota (C_i[(I\ \vec{p}), Fun\_Elim(I, Q, \vec{f}, \vec{p}), f_i]\ \vec{m})$$

Exemple :

Reprenons l'exemple des listes paramétrées. Nous avons les deux cas *nil* et *cons*.

$$Fun\_Elim(list, Q, [f_1; f_2], A) \succ_\beta [c : (list\ A)]\mathsf{Elim}(c, Q)\{f_1|f_2\}$$

$$
\begin{aligned}
\mathsf{Elim}(nil\ A, Q)\{f_1|f_2\} &\succ_\iota X[(list\ A), Fun\_Elim(list, Q, [f_1; f_2], A), f_1] \\
&= f_1
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{Elim}(cons\ A\ x\ l, Q)\{f_1|f_2\} \succ_\iota \\
&((A \to X \to X)[(list\ A), Fun\_Elim(list, Q, [f_1; f_2], A), f_2]\ x\ l) \\
&= ([x : A](X \to X)[(list\ A), Fun\_Elim(list, Q, [f_1; f_2], A), (f_2\ x)]\ x\ l) \\
&= ([x : A][l : (list\ A)]X[(list\ A), Fun\_Elim(list, Q, [f_1; f_2], A), (f_2\ x\ l \\
&\quad \mathsf{Elim}(l, Q)\{f_1|f_2\})]\ x\ l) \\
&= ([x : A][l : (list\ A)](f_2\ x\ l\ \mathsf{Elim}(l, Q)\{f_1|f_2\})\ x\ l) \\
&= (f_2\ x\ l\ \mathsf{Elim}(l, Q)\{f_1|f_2\})
\end{aligned}
$$

Exemples :

- On peut définir par une élimination une fonction booléenne qui teste si un entier est pair ou pas.

```
[n:nat]<bool>Match n with
            true
            [m:nat][H:bool](not H)
      end
```

- On peut définir par une élimination une fonction qui rend la queue d'une liste. Si la liste est vide, on rend la liste vide.

```
[l:list]<list>Match l with
            nil
            [a:A][m:list][H:list]m
      end
```

Conclusion

Ce chapitre a décrit progressivement différents systèmes de formalisation du raisonnement. Nous sommes partis du Calcul des Prédicats. Ce dernier est trop peu expressif. Il ne permet que la représentation de fonctions très simples. Nous avons donc introduit le λ-calcul simplement typé qui permet de coder les fonctions de l'arithmétique du premier ordre. Comme cela n'est pas encore suffisant, nous avons présenté le λ-calcul polymorphe qui permet de coder des types de données et d'exprimer les fonctions de l'arithmétique du second ordre. Enfin, nous sommes arrivés aux fonctions de l'arithmétique d'ordre supérieur grâce au système $F_\omega$avec types inductifs. Ce dernier sera notre langage de programmation car il a la plus grande expressivité.

Ensuite, pour pouvoir exprimer des propriétés sur les programmes du système F_ω, il a fallu passer un niveau supplémentaire. C'est le Calcul des Constructions qui permet grâce aux types dépendants de représenter de telles propriétés. Ce sera notre langage de preuve.

Enfin, nous avons introduit les types inductifs. Ceux-ci permettent de coder de façon plus naturelle les types de données. Ainsi, dans la suite, nos systèmes de base seront : le Calcul des

Constructions avec types inductifs comme langage de preuve (voir 1.2.4, 1.2.5) et le système F_ω avec types inductifs comme langage de programmation (voir 1.2.3).

Maintenant que nous avons cerné ces deux langages et les raisons pour lesquelles nous les avons choisis, nous allons voir comment retrouver des programmes à partir de preuves.

Chapitre 2

Synthèse de programmes à partir de preuves : l'extraction

Nous avons motivé dans le chapitre précédent le choix de F_ω avec types inductifs (noté ${F_\omega}^{Ind}$ dans toute la suite) comme langage de programmation et du Calcul des Constructions avec types inductifs comme langage de preuve.

Faire des preuves dans un système comme le Calcul des Constructions Inductives, c'est se donner un type valide dans ce calcul (la spécification) et chercher un habitant de ce type. Ainsi, comme nous l'avons vu pour la déduction naturelle, les types représentent les formules à prouver et les termes le chemin suivi pour montrer cette formule. Un habitant d'un type T est tout simplement un terme de type T. Pour un même type, il peut bien sûr exister plusieurs habitants. Le terme obtenu est alors le terme de preuve correspondant à la formule.

Toute preuve est alors un programme correct. En effet, la spécification représente les propriétés qu'on souhaiterait vérifier et la preuve une méthode pour vérifier ces propriétés. Preuve et programme peuvent donc être assimilés. C'est l'isomorphisme de Curry-Howard. Prouver la spécification revient à construire un programme correct vis-à-vis de cette spécification.

Cependant, le programme construit est inefficace du fait de la présence de sa preuve de correction de façon interne. Reprenons l'exemple de la division présenté dans l'introduction. Une spécification d'un algorithme de division est $\forall a, b(b > 0) \rightarrow \exists q, r(a = b * q + r) \wedge (b > r)$. Donnons le squelette de la preuve mathématique de cette spécification. Il s'agit d'une preuve par récurrence sur a. Si a est nul, alors $q = r = 0$ et on a bien $0 = b * 0 + 0$ et $b > 0$. Sinon, on suppose qu'on connaît q et r pour le prédécesseur de a c'est-à-dire tels que $a - 1 = b * q + r$ et $b > r$, et on essaie de construire un q' et r' pour a. Il faut distinguer deux cas : soit $b \leq (r + 1)$, soit $b > (r + 1)$. Dans le premier cas, le quotient résultat q' est $q + 1$, le reste r est 0 et on a bien $a = b * (q + 1) + r$ et $b > 0$ (la première propriété vient du fait que $a - 1 = b * q + r$ et que $b > r$ et $b \leq (r + 1)$ donc $b = r + 1$; sa preuve n'est pas immédiate). Dans le second cas, le quotient résultat q' est q, le reste r' est $r + 1$ et on a bien $a = b * q + r + 1$ et $b > (r + 1)$. La preuve contient donc une méthode de construction explicite de q et r mais elle prend aussi en argument une preuve que $b > 0$ et rend une preuve de $(a = b * q + r) \wedge (b > r)$ pour chacun des cas possibles pour a. Mais, au niveau du programme, on ne s'intéresse pas par exemple à la justification du fait que $a = b * q + r$ pour le q et le r construits.

Donc, si la preuve contient un algorithme explicite pour vérifier la spécification, elle contient aussi beaucoup d'informations annexes (justifications de $a = b * q + r$ pour la division par exemple). Cette idée est liée à la sémantique de Heyting des preuves constructives : une preuve constructive d'une formule F de $\forall x.P(x) \Rightarrow \exists y.Q(x, y)$ contient un algorithme qui, étant donnés un terme x et une preuve de $P(x)$ rend un terme y et une preuve de $Q(x,y)$. x est l'entrée du programme et y sa sortie. Or, si une partie de la preuve sert à calculer la sortie y, une autre partie sert à prouver $Q(x,y)$. On veut connaître l'existence de cette partie preuve qui garantit l'existence du programme mais, par contre, on ne veut pas avoir à l'exécuter afin de rester efficace. La partie preuve pure est la partie logique de la preuve et la partie qui sert à calculer la sortie est sa partie calculatoire. Avec cette distinction, seule la partie calculatoire de la preuve peut alors être gardée en tant que programme.

C'est pourquoi la notion d'extraction est introduite. Une fonction d'extraction permet de retirer toutes les parties logiques d'une preuve et de garder comme programme uniquement les parties calculatoires. Chaque preuve différente correspond à un programme différent. Dans ce chapitre, nous allons présenter la fonction d'extraction de [PM89a] dans le Calcul des Constructions Inductives, comme méthode de synthèse de programmes à partir de preuves.

2.1 Extraction de programmes

Après une preuve, on souhaite ne plus garder le terme complet comme programme mais oublier les parties inutiles au calcul lui-même. L'extraction est donc une fonction d'oubli sur les termes du Calcul des Constructions Inductives vers les termes de ${F_\omega}^{Ind}$. Elle consiste à distinguer des termes avec contenu calculatoire de termes sans contenu calculatoire (voir exemple précédent de la division).

Pour cela, nous allons modifier l'ensemble des sortes du calcul. Nous disposons pour l'instant d'une seule sorte $Set : TypeSet$. Il faut noter que *TypeSet* n'est pas un terme du calcul. Il a un rôle particulier qui est d'assurer l'uniformité des jugements $\Gamma \vdash t : TypeSet$ signifiant qu'une sorte t est correcte. Nous allons ajouter deux nouvelles sortes $Prop$ et $Type$, destinées à faire la distinction entre les termes qui servent au calcul (termes de contenu calculatoire) et ceux qui ne servent pas (termes sans contenu calculatoire). Nous aurons donc $Prop : Type$ pour les termes sans contenu calculatoire, $Set : TypeSet$ pour les termes avec contenu calculatoire.

En terme de PTS, le nouveau calcul est donc défini par l'ensemble $\{(s,s') \mid (s,s') \in S \times S\}$ où $S = \{Prop, Set, Type, TypeSet\}$ avec les axiomes $Prop : Type$ et $Set : TypeSet$.

2.1.1 Définitions

Nous allons maintenant détailler le processus d'extraction dans le Calcul des Constructions Inductives [PM89b, PM89a, PMW92]. Pour cela, nous avons besoin de plusieurs définitions préalables. Nous définissons une notion de niveau sur les termes du calcul. Cette définition est nécessaire pour pouvoir stratifier le calcul. En effet, les termes et les types sont définis de façon entremêlée. Il est donc très difficile de faire des preuves avec de telles définitions. En les stratifiant à l'aide de niveaux par exemple, les définitions sur de tels termes et les démonstrations sont beaucoup plus simples. Nous donnons tout d'abord la nouvelle définition d'arité avec le système de marquage *Prop*, *Set*.

Définition 30 *Arité :*
Une arité est un terme qui suit la syntaxe suivante :

$$Ar ::= Prop \mid Set \mid (x : M)Ar$$

Définition 31 *Niveaux des termes :*

- *Un terme est dit de niveau 0 s'il s'agit d'une arité (voir définition 30). C'est le domaine des types propositionnels ;*

- *Un terme est dit de niveau 1 si son type est de niveau 0. C'est le domaine des schémas propositionnels ou prédicats ;*

- *Un terme est dit de niveau 2 si son type est de niveau 1. C'est le domaine des preuves.*

Remarques :

- Chaque terme a un niveau unique défini par son typage.
- Un type est toujours de niveau 0 ou 1.
- Un argument d'une application est toujours de niveau 1 ou 2.
- Un produit est toujours de niveau 0 ou 1.
- Une abstraction est toujours de niveau 1 ou 2.

Exemples :

- Niveau 0 :
  Si A est un type quelconque, $A \to Set$ est un terme de niveau 0 ;
- Niveau 1 :
  Si A et B sont des types quelconques (mais B n'est pas une arité), alors $(x : A)B$ et $[x : A]B$ sont des termes de niveau 1. Les types inductifs sont de niveau 1. Une élimination forte est de niveau 1.
- Niveau 2 :
  Les constructeurs de types inductifs et les éliminations sont des termes de niveau 2. La variable $+ : nat \to nat \to nat$ de l'arithmétique est de niveau 2 si on suppose que le type nat est un type inductif à deux constructeurs 0 et S. Alors, le terme $[x : nat](+\ x\ (S\ 0))$ est de niveau 2.

Définition 32 *Arité informative :*
Une arité informative est un terme qui suit la syntaxe suivante :

$$ArInf ::= Set \mid (x : M)ArInf$$

Définition 33 *Terme informatif, terme logique :*

- *Un terme de niveau 0 est informatif s'il s'agit d'une arité informative ;*

- *Un terme de niveau 1 est informatif si son type est une arité informative ;*

- *Un terme de niveau 2 est informatif si le type de son type est Set.*

Un terme est logique s'il n'est pas informatif.

Exemples :

- Niveau 0 :
  Si A est un type quelconque, $A \to Set$ est un terme informatif de niveau 0 et $A \to Prop$ est un terme logique de niveau 0 ;

- Niveau 1 :
  Si A et B sont des types quelconques (mais B différent de *Set* ou *Prop*), alors $(x : A)B$ et $[x : A]B$ sont des termes informatifs de niveau 1 si B est informatif de niveau 1. Un type inductif est informatif si son arité est informative ;
- Niveau 2 :
  Le terme $[x : nat](+\ x\ (S\ 0))$ est informatif de niveau 2 car l'arité de *nat* est *Set*.

Remarque : Le caractère informatif d'un terme dépend uniquement du typage.

2.1.2 Fonction d'extraction

Nous pouvons avec ces définitions donner une définition de l'extraction de [PM89a]. Nous allons donc définir trois fonctions, une pour chaque niveau, ε_0, $\varepsilon_1$et ε_2.

Nous avons un problème spécifique aux constructeurs de type inductif. En effet, les paramètres appliqués aux constructeurs ne doivent pas être traités de la même façon que des arguments standards d'une application de niveau 2. On ne souhaite conserver que les paramètres informatifs de niveau 1. Or, dans une application de niveau 2, on garde tous les arguments informatifs quelque soit leur niveau. Le comportement attendu n'est donc pas le même pour les paramètres appliqués à un constructeur que pour les arguments standards d'une application de niveau 2. Pour permettre cela, nous allons considérer qu'un constructeur apparaît toujours appliqué à ses paramètres.

Si l'on regarde en détail la définition suivante, on constate qu'en fait on oublie tous les termes logiques mais aussi les dépendances des types par rapport aux preuves que nous appelons spécifications. Ainsi, dans l'extraction d'un type sous forme d'application, on ne conserve l'argument que s'il est informatif de niveau 1. On oublie donc bien les preuves en argument. Autrement dit, l'extraction retire tous les termes logiques ainsi que tous les termes de niveau 2 à l'intérieur des types. L'extraction projette donc une spécification vers un type *simple* de ${F_\omega}^{Ind}$.

De plus, au niveau des types inductifs, on peut noter qu'on ne conserve que les paramètres dont le type est un type propositionnel (niveau 0) informatif. En effet, sur l'arité A d'un type inductif, on applique ε_0 qui correspond à ce qu'on vient de dire. Pour le problème des paramètres dans les constructeurs, on considère donc qu'un constructeur apparaît toujours appliqué à ses arguments. On traite ses paramètres différemment des arguments d'un terme de niveau 2, en leur appliquant ε_1, et donc en ne gardant que les paramètres informatifs de niveau 1.

Notation : un constructeur appliqué à ses paramètres est noté $\mathsf{Constr}(i, ind, \vec{p})$ où i est le numéro du constructeur, *ind* le type inductif correspondant et $\vec{p}$ le vecteur des paramètres.

Définition 34 *Extraction sur les termes informatifs :*

$$\varepsilon_0(Set) = Set$$

$$\varepsilon_0((x : A)B) = \begin{cases} \varepsilon_0(A) \rightarrow \varepsilon_0(B) & \textit{si } A \textit{ informatif de niveau 0} \\ \varepsilon_0(B) & \textit{sinon} \end{cases}$$

$$\varepsilon_1(x) = \bar{x} \quad \textit{où } \bar{x} \textit{ est de type } \varepsilon_0(T_x) \textit{ si } x \textit{ est de type } T_x$$

$$\varepsilon_1((x : A)B) = \begin{cases} (\bar{x} : \varepsilon_i(A))\varepsilon_1(B) & \textit{si } A \textit{ informatif} \\ \varepsilon_1(B) & \textit{sinon} \end{cases}$$

$$\varepsilon_1([x : A]B) = \begin{cases} [\bar{x} : \varepsilon_0(A)]\varepsilon_1(B) & \textit{si } A \textit{ informatif de niveau 0} \\ \varepsilon_1(B) & \textit{sinon} \end{cases}$$

$$\varepsilon_1(A\ B) = \begin{cases} (\varepsilon_1(A)\ \varepsilon_1(B)) & \textit{si } B \textit{ informatif de niveau 1} \\ \varepsilon_1(A) & \textit{sinon} \end{cases}$$

$\varepsilon_1(\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1 | \ldots | c_m : C_m\}) =$

$\mathsf{Ind}[\bar{p_i} : \varepsilon_0(P_i)](\bar{X} : \varepsilon_0(A))\{\overline{c_1} : \varepsilon_1(C_1)| \ldots |\overline{c_m} : \varepsilon_1(C_m)\}$ *pour les* P_i *informatifs de niveau 0*

$\varepsilon_2(x) = \bar{x}$ *où* $\bar{x}$ *est de type* $\varepsilon_1(T_x)$ *si* x *est de type* T_x

$$\varepsilon_2([x : A]B) = \begin{cases} [\bar{x} : \varepsilon_i(A)]\varepsilon_2(B) & \textit{si } A \textit{ informatif} \\ \varepsilon_2(B) & \textit{sinon} \end{cases}$$

$$\varepsilon_2(A\ B) = \begin{cases} (\varepsilon_2(A)\ \varepsilon_i(B)) & \textit{si } B \textit{ informatif} \\ \varepsilon_2(A) & \textit{sinon} \end{cases}$$

$\varepsilon_2(\mathsf{Elim}(c, P)\{f_1| \ldots |f_n\}) = \mathsf{Elim}(\varepsilon_2(c), \varepsilon_1(P))\{\varepsilon_2(f_1)| \ldots |\varepsilon_2(f_n)\}$

$\varepsilon_2(Constr(i, ind, \vec{p})) = \mathsf{Constr}(i, \varepsilon_1(ind), \varepsilon_1(\vec{p}))$

L'extraction n'est définie que sur les termes informatifs. Tous les termes logiques sont oubliés.

Remarque : Chaque terme a un niveau unique (voir remarque précédente). On note donc ε_i la fonction d'extraction correspondant à un terme de niveau i.

Définition 35 *Extraction sur les contextes :*
$\varepsilon(\Gamma)$ *est le contexte extrait de* Γ *obtenu en appliquant* ε_i *sur les variables informatives du contexte et leur type, et en oubliant les variables logiques.*

Nous souhaitons maintenant montrer la bonne formation des termes extraits obtenus. Pour cela, nous introduisons une nouvelle définition sur les contextes.

Définition 36 *Contexte de niveau 0 :*
Γ_0 *est le contexte de niveau 0 associé à* Γ*. Il est obtenu à partir de* Γ *en ne gardant que les couples* (X, A) *où* X *est une variable de niveau 1 et* A *un terme de niveau 0 et tel que toute variable n'apparaît comme première composante que d'au plus un couple.*

Proposition 1 *Si* $\Gamma \vdash t : P$ *et* P *est informatif alors :*

- *si* $P = TypeSet$*, alors* $\vdash \varepsilon_0(t) : TypeSet$ *;*
- *si* P *est de type* $TypeSet$*, alors* $\varepsilon(\Gamma)_0 \vdash \varepsilon_1(t) : \varepsilon_0(P)$ *;*
- *si* P *est de type* Set*, alors* $\varepsilon(\Gamma) \vdash \varepsilon_2(t) : \varepsilon_1(P)$*.*

Preuve. Par récurrence sur la longueur de la dérivation. □

2.1.3 Exemples

Reprenons la définition du constructeur de formules $\exists$. La définition avec termes marqués est :

$$sig := \mathsf{Ind}[A : Set; P : A \rightarrow Prop](X : Set)\{exist : (x : A)(P\ x) \rightarrow X\}$$

P étant logique, il est supprimé des arguments ainsi que du type du constructeur $\exists$.

$$\varepsilon_1(sig) = \mathsf{Ind}[\bar{A} : Set](\bar{X} : Set)\{\overline{exist} : \bar{A} \rightarrow \bar{X}\}$$

Définissons maintenant le type inductif du constructeur de formules $\vee$ entre un terme informatif et un terme logique :

$$sumor := \mathsf{Ind}[A : Set; B : Prop](X : Set)\{inleft : A \to X |\ inright : B \to X\}$$

B étant logique, il est supprimé des arguments ainsi que du type du constructeur *inright*.

$$\varepsilon_1(sumor) := \mathsf{Ind}[\bar{A} : Set](\bar{X} : Set)\{\overline{inleft} : \bar{A} \to \bar{X} |\ \overline{inright} : \bar{X}\}$$

Définissons le type inductif des entiers :

$$nat := \mathsf{Ind}(X : Set)\{0 : X | S : X \to X\}$$

Son extraction est lui-même.

Prenons maintenant le terme suivant qui est une preuve de la spécification du prédécesseur : $(n : nat)\{m : nat | (S\ m) = n\} + \{0 = n\}$. La syntaxe utilisée est celle de *Coq* (voir section 1.2.5). Cette spécification signifie que, pour tout entier positif n, soit il existe une entier m prédécesseur de n soit $n = 0$. Une preuve intuitive de cette spécification consiste en une analyse par cas sur n. Si $n = 0$ alors c'est la partie droite de la spécification qui est vraie. Sinon, $n = y + 1$, et c'est la partie gauche de la spécification qui est vraie avec comme témoin y. Un terme de preuve possible pour cette spécification est donc :

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
           (inright {m:nat|(S m)=0} (0=0) (refl_equal nat 0))
           [y:nat][H:{m:nat|(S m)=y}+{0=y}]
             (inleft {m:nat|(S m)=(S y)} (0=(S y))
                 (exist nat [m:nat](S m)=(S y) y (refl_equal nat (S y))))
       end
```

Détaillons l'extraction complète de ce terme T sachant les deux points précédents.

Détaillons tout d'abord l'extraction du premier argument T_1 de l'élimination.

$$\varepsilon_2(T_1) = (\varepsilon_2(inright)\ \varepsilon_1(sig\ nat\ [m : nat]((Sm) = 0)))$$

car l'égalité est sans contenu calculatoire et donc supprimée, et `refl_equal` aussi. Donc,

$$\varepsilon_2(T_1) = (\overline{inright}\ (\overline{sig}\ nat))$$

pour les mêmes raisons.

Détaillons maintenant l'extraction du second argument T_2 de l'élimination.

$$\begin{aligned} \varepsilon_2(T_2) &= [y : nat][H : \varepsilon_1((sumor\ (sig\ nat\ [m : nat]((S\ m) = y))\ (0 = y)))] \\ &\quad \varepsilon_2((inleft\ (sig\ nat\ [m : nat]((S\ m) = (S\ y)))\ (O = (S\ y)) \\ &\quad (exist\ nat\ [m : nat]((S\ m) = (Sy))\ y\ (refl\_equal\ nat\ (S\ y))))) \\ &= [y : nat][H : (\varepsilon_1(sumor)\ (\varepsilon_1(sig)\ nat))] \\ &\quad (\varepsilon_2(inleft)\ (\varepsilon_1(sig)\ nat)\ (\varepsilon_2(exist)\ nat\ \overline{y})) \end{aligned}$$

pour les mêmes raisons que précédemment. Donc,

$$\begin{aligned} \varepsilon_2(T_2) &= [y : nat][H : (\overline{sumor}\ (\overline{sig}\ nat))] \\ &\quad (\overline{inleft}\ (\overline{sig}\ nat)\ (\overline{exist}\ nat\ \overline{y})) \end{aligned}$$

Pour pouvoir calculer l'extraction totale du terme T, il ne nous manque que l'extraction de n qui est $\overline{n}$ et celle du prédicat de l'élimination.

$$\begin{array}{rcl} \varepsilon_1([n0:nat](sumor\ (sig\ nat\ [m:nat]((S\ m)=n0))\ (0=n0))) & = & (\varepsilon_1(sumor)\ (\varepsilon_1(sig)\ nat)) \\ & = & (\overline{sumor}\ (\overline{sig}\ nat)) \end{array}$$

car, *nat* est de niveau 1 dans une abstraction de niveau 1 et est donc supprimé, et l'égalité et `refl_equal` sont sans contenu calculatoire et donc supprimés aussi.

L'extraction du terme T est finalement le terme suivant :

```
[n:nat]<(sumor (sig nat))>Match n with
          (inright (sig nat))
          [y:nat][H:(sumor (sig nat))](inleft (sig nat) (exist nat y))
       end
```

si l'on oublie les barres sur les constantes et qu'on sous-entend qu'on est dans le contexte extrait.

2.2 Extraction des types singletons

L'extraction des types singletons est une optimisation apportée à l'extraction de [PM89a] dans le but d'obtenir des programmes plus proches de la réalité. Il faut noter que nous détaillons ici cette optimisation mais que nous ne nous en servirons pas dans le chapitre suivant et uniquement dans le chapitre 4.

Pourquoi distinguer ces types en particulier ? En fait, un type singleton sur A représente l'ensemble des 1-uplets (a) pour $a : A$. L'extraction de [PM89a] extrait certains types en des ensembles de 1-uplets. Typiquement, un type $\{x : A|(P\ x)\}$ est extrait en $(sig\ A)$. D'un point de vue programmation, on ne souhaite pas garder d'ensembles de 1-uplets. En effet, pour utiliser les éléments d'une paire, on la déstructure c'est-à-dire on fait une élimination sur le type des paires. Pour le cas d'un ensemble de 1-uplets, il n'est pas naturel de dire qu'on va le déstructurer pour avoir accès à un 1-uplet. L'idée naturelle est que, puisqu'il y a isomorphisme entre A et un type singleton sur A, on les identifie. L'idée de l'extraction des types singletons est donc de ne plus extraire un type singleton en un autre type singleton, mais plutôt en le type qui lui est sous-jacent c'est-à-dire le type sur lequel il est singleton.

Montrons sur un exemple l'amélioration que nous souhaitons apporter grâce à cette considération. Reprenons l'exemple précédent sur le prédécesseur.

Exemple :

Le terme de preuve est le suivant pour la spécification $(n : nat)\{m : nat|(S\ m) = n\}+\{0 = n\}$.

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
          (inright {m:nat|(S m)=0} (0=0) (refl_equal nat 0))
          [y:nat][H:{m:nat|(S m)=y}+{0=y}]
            (inleft {m:nat|(S m)=(S y)} (0=(S y))
                (exist nat [m:nat](S m)=(S y) y (refl_equal nat (S y))))
       end
```

L'extraction sans optimisations fourni le terme :

```
[n:nat]<(sumor (sig nat))>Match n with
          (inright (sig nat))
          [y:nat][H:(sumor (sig nat))](inleft (sig nat) (exist nat y))
       end
```

Or, dans cet exemple, *sig* est un type singleton sur *nat*. Au lieu d'extraire $\{m : nat|(Sm) = n0\}$ en $(sig\ nat)$, on voudrait extraire en *nat*. Ceci signifie que le constructeur *exist* disparaîtrait aussi à l'extraction. De plus, toute élimination sur un terme de type $(sig\ nat)$ disparaîtrait aussi (dans cet exemple, il n'y en a pas).

Donc, cela permettrait d'obtenir un terme beaucoup plus simple, à savoir :

```
[n:nat]<(sumor nat)>Match n with
          (inright nat)
          [y:nat][H:(sumor nat)](inleft nat y)
       end
```

beaucoup plus proche du programme ML que l'on aurait écrit pour la fonction qui rend le prédécesseur de n s'il existe et erreur sinon.

```
let pred n = match n with
               0 -> error"n=0"
             | y+1 -> y ;;
```

On voit que l'on s'en rapproche beaucoup plus qu'avant si l'on considère que *sumor* simule les exceptions.

2.2.1 Définitions

Définition 37 *Type singleton :*
Un type inductif singleton T (ou type singleton) est un type inductif à un seul constructeur vérifiant la propriété suivante : $T \equiv \mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : I)\{c_1 : C_1\}$ *où* I *est une arité et* $\varepsilon_1(C_1) = A \rightarrow X$ *pour* A *tel que* $\varepsilon(\Gamma)[\overline{p_i} : \varepsilon_0(P_i)] \vdash A : \varepsilon_0(I)$ *où* Γ *est un contexte dans lequel* T *est bien typé.* T *est alors dit type singleton sur* A*.*

On peut noter que les conditions ci-dessus entraînent que A ne contient pas X et que I termine par *Set*.

Exemple : Un exemple de type singleton est *sig*.

En effet : $sig = \mathsf{Ind}[A : Set; P : A \rightarrow Prop](X : Set)\{exist : (x : A)(P\ x) \rightarrow X\}$ et $\varepsilon_1((x : A)(P\ x) \rightarrow X) = A \rightarrow X$. Dans l'exemple précédent, *sig* est utilisé comme type singleton sur *nat*.

2.2.2 Fonction d'extraction sur les types singletons

Nous pouvons maintenant formaliser les intuitions vues dans l'exemple du prédécesseur. Ainsi, on a la première équation ci-dessous (on ne garde dans les deux cas que les $P_i : TypeSet$) :

$$\varepsilon_1(\mathsf{Ind}[p_i : P_i](X : I)\{c_1 : C_1\}) = \begin{cases} [\bar{p_i} : \varepsilon_0(P_i)]A(p_i) & \text{si type singleton sur } A \\ \mathsf{Ind}[\bar{p_i} : \varepsilon_0(P_i)](\bar{X} : \varepsilon_0(I))\{\overline{c_1} : \varepsilon_1(C_1)\} & \text{sinon} \end{cases}$$

Comme pour l'extraction de [PM89a], on ne garde que les paramètres dont le type est un type propositionnel informatif.

Pour être complet, il reste à voir les cas de l'extraction d'une élimination sur un type singleton et de l'extraction d'un constructeur d'un type singleton.

Une élimination sur un type singleton ne contient qu'un cas unique (puisqu'il n'y a qu'un constructeur unique). Comme on l'a dit précédemment, éliminer sur un type singleton signifie récupérer le terme sous-jacent au type singleton pour l'utiliser dans la suite. Donc, si on souhaite que l'extraction d'un type singleton sur A soit A, l'extraction d'une élimination sur un type singleton ne doit plus exister. On n'a plus besoin d'éliminer et on peut directement garder le terme pour l'utiliser dans la suite.

$$\varepsilon_2(\mathsf{Elim}(c, P)\{f_1 | \ldots | f_n\}) = \begin{cases} (\varepsilon_2(f_1)\ \varepsilon_2(c)) \text{ si le type de } c \text{ est un type singleton} \\ \mathsf{Elim}(\varepsilon_2(c), \varepsilon_1(P))\{\varepsilon_2(f_1) | \ldots | \varepsilon_2(f_n)\} \text{ sinon} \end{cases}$$

Enfin, le cas constructeur n'a plus de raison d'être non plus. Au lieu de dire que l'on a le premier constructeur d'un type singleton sur A, on souhaite simplement dire qu'on a quelque chose de type A. L'extraction d'un constructeur d'un type singleton sur A devient alors la fonction identité sur A.

$$\varepsilon_2(Constr(i, ind, \vec{p})) = \begin{cases} (([\bar{p_i} : \varepsilon_0(P_i)][\bar{x} : A]\bar{x})\ \varepsilon_1(\vec{p})) & \text{si } ind \text{ est un type singleton sur } A \\ \mathsf{Constr}(i, \varepsilon_1(ind), \varepsilon_1(\vec{p})) & \text{sinon} \end{cases}$$

Nous pouvons maintenant montrer l'effet de simplification sur les programmes de cette extraction des types singletons. Nous verrons sur ce petit exemple que le programme est déjà beaucoup plus lisible et proche d'un code fonctionnel.

Exemple : Formellement, sur l'exemple du prédécesseur, nous avons :

$$\varepsilon_1(sig) = [A : Set]A$$
$$\varepsilon_2(exist) = [A : Set][x : A]x$$

La forme extraite globale est donc bien celle que nous avons donnée précédemment.

2.3 Non-inversibilité de l'extraction

Nous avons une fonction qui permet de synthétiser des programmes fonctionnels à partir de preuves dans le Calcul des Constructions Inductives. Cette idée ne nous semble pas la plus naturelle. Il semble qu'un programmeur souhaiterait plutôt donner un programme et sa spécification et avoir une preuve *automatique* que ce programme est correct vis-à-vis de la spécification.

Pour cela, une idée naturelle est d'inverser la fonction d'extraction. En effet, l'extraction est une fonction d'oubli d'un langage de preuves vers un langage de programmation. Nous recherchons une fonction qui reconstruise un terme de preuve à partir d'un terme de programme, donc une fonction d'un langage de programmation vers un langage de preuves. Malheureusement, on est limité par le fait que toute l'intelligence sur la manière de prouver des propriétés est inexistante dans le programme. On peut prendre un exemple et montrer qu'il est impossible, étant donnés une spécification et un programme, de retrouver un terme de preuve. On considère l'extraction sans optimisations; avec celles-ci, c'est encore pire. En effet, l'extraction avec optimisations oublie encore plus de choses que l'extraction sans optimisations. De plus, les optimisations transforment la structure du terme (des éliminations et des constructeurs disparaissent).

Exemple :

Dans le cas d'une spécification $\{x : A|(P\ x)\}$ et d'un programme a, il faut savoir reconstruire une preuve de $(P\ a)$ pour un P arbitraire. C'est impossible. En effet, nous avons déjà souligné qu'il existe plusieurs preuves possibles pour une même spécification. Il serait éventuellement possible de toutes les énumérer mais ce n'est pas une solution viable en pratique. Il est donc impossible de retrouver une preuve uniquement à partir de sa spécification.

D'autre part, les spécifications intermédiaires disparaissent. Prenons le cas d'une spécification $(n : nat)(even\ n) \rightarrow \{p : nat|2 * p = n\}$. Dans une preuve, on peut renforcer cette spécification en faisant la distinction entre les cas n pair et n impair et obtenir une spécification intermédiaire $(n : nat)(even\ n) \rightarrow \{p : nat|2 * p = n\} \wedge (odd\ n) \rightarrow \{p : nat|2 * p + 1 = n\}$. A partir d'un programme, il est impossible de retrouver une telle spécification intermédiaire car il faut retrouver des informations logiques qui ne sont pas dans le programme.

Les deux points présentés précédemment sont distincts. Dans le premier point, le problème est de retrouver des preuves logiques. Il sera toujours impossible de les retrouver. Dans le deuxième point, le problème est en fait de retrouver les types dépendants des preuves qui ont été oubliés dans l'extraction.

Conclusion

Dans ce chapitre, nous avons introduit la notion d'extraction de [PM89a]. L'extraction est une fonction qui oublie les parties logiques d'une preuve ainsi que les dépendances des types par rapport aux preuves. Elle extrait des programmes de ${F_\omega}^{Ind}$ à partir de preuves du Calcul des Constructions Inductives.

Nous avons ensuite dégagé une optimisation sur cette extraction. Nous avons identifié des types particuliers, les types singletons. Nous avons montré que ces types et les structures de preuves qu'ils induisent étaient conservés par l'extraction de [PM89a]. Or, ce ne sont pas des structures naturelles dans un programme. Nous avons donc montré comment optimiser l'extraction et les supprimer. Les termes obtenus sont beaucoup plus proches d'un langage à la ML.

Cependant, cette approche de synthèse de programmes à partir de preuves ne nous semble pas la plus naturelle. Il semble qu'une autre approche puisse consister à prendre un programme et sa spécification et à retrouver le terme de preuve correspondant. Cette idée revient à inverser la fonction d'extraction. Malheureusement, nous avons montré que ceci était impossible sur la fonction d'extraction de [PM89a].

L'objet du chapitre suivant va donc être d'introduire une nouvelle fonction d'extraction qui, elle, pourra être inversée, c'est-à-dire pourra permettre de retrouver les types dépendants des preuves et laisser en lemmes des preuves de propriétés logiques.

Chapitre 3

Synthèse de preuves à partir de programmes : extraction faible et inversion

Nous souhaitons synthétiser des preuves à partir de programmes. Nous considérons que, dans la pratique, un programmeur a généralement une idée d'un algorithme et une idée de la spécification qu'il doit vérifier. Nous souhaiterions de façon naturelle donner un programme et une spécification et montrer que ce programme vérifie la spécification. Dans le chapitre précédent, nous avons vu que, si nous marquions les termes du Calcul des Constructions Inductives suivant leur contenu calculatoire, nous savions extraire de manière unique le contenu calculatoire d'une preuve. La fonction d'extraction est définie par récurrence sur la structure du terme de preuve. Elle produit un terme de programme qui a une structure identique au terme de preuve mais contenant moins d'informations. Donc, on peut facilement voir que le terme de programme est un squelette du terme de preuve dont il est extrait, épuré de tout ce qui est inutile au calcul lui-même.

Pour cela, l'idée naturelle est d'inverser la fonction d'extraction comme nous l'avons déjà motivé précédemment. Malheureusement, nous avons vu à la fin du chapitre précédent qu'inverser cette fonction d'extraction était impossible. Nous allons montrer qu'une fonction d'extraction faible, qui oublie moins de termes, peut être définie. Cette fonction peut alors être inversée au sens où, si nous avons un terme faiblement extrait et une spécification associée (type du terme de preuve recherché), alors nous pouvons retrouver un terme de preuve partiel correspondant, de façon totalement déterministe. Le terme est partiel car nous ne pouvons espérer engendrer qu'un certain nombre de lemmes logiques correspondant à des obligations de preuves. Si ces obligations sont prouvées, alors le programme est correct. Ceci signifie qu'un terme faiblement extrait est une représentation condensée d'une preuve, qui réduit leur taille sans empêcher de retrouver de façon déterministe le terme de preuve à partir de la spécification.

Malheureusement, les termes faiblement extraits ne sont pas satisfaisants comme programmes naturels. Nous donnons alors une méthode qui, partant de programmes extraits, permet, par des heuristiques, de retrouver des termes de preuve. Cette méthode n'est, à ce stade, plus déterministe car certains termes manquants dans les programmes extraits ne peuvent être retrouvés de façon déterministe. Nous introduisons alors, un mécanisme d'annotations des programmes pour ajouter des informations logiques dans les programmes. Nous pouvons alors montrer que nous sommes ramené au problème des termes faiblement extraits et que, par conséquent, la méthode est correcte et complète au sens d'une notion de terme suffisamment annoté.

3.1 Extraction faible

3.1.1 Motivations

Lorsqu'on fait une preuve dans le Calcul des Constructions Inductives, les types sont des annotations qui donnent ce que prouve le terme. Ces types ne servent pas au calcul lui-même. Il serait en fait possible de prendre le terme pur et de calculer avec. Dans l'extraction de [PM89a], on se sert de la distinction *Prop/Set* dans les types pour enlever dans le terme pur des morceaux que l'on sait ne pas vouloir exécuter. En plus, on fait de l'élagage dans les types (suppression des types dépendants), ce qui assure que le programme est correctement typé dans le système ${F_\omega}^{Ind}$ donc fortement normalisable par exemple.

Maintenant, nous allons nous intéresser à une opération intermédiaire qui va enlever seulement les preuves logiques. Le contenu calculatoire des termes ainsi obtenus (appelés programmes annotés) sera donc le même que dans l'extraction de [PM89a] mais, par contre, les types vont être gardés de manière à conserver l'information logique sur la façon dont la preuve s'enchaîne. La fonction d'extraction faible va donc conserver les spécifications et non plus les extraire vers des types simples de ${F_\omega}^{Ind}$.

Les termes ne seront alors plus correctement typés au sens usuel du Calcul des Constructions Inductives et nous serons obligés d'introduire un nouveau jugement de typage pour assurer la cohérence d'un programme annoté. Nous allons donc construire une fonction sur le modèle de l'extraction de [PM89a] (nous ne considérons pas l'optimisation des types singletons dans ce chapitre), dont le rôle sera de supprimer uniquement les termes logiques de niveau 2. Ce qui signifie que le terme obtenu sera un programme annoté de spécifications (c'est-à-dire de types dépendants correspondant par exemple aux prédicats d'élimination utilisés). Les preuves logiques supprimées correspondront à des preuves de propriétés logiques que le programme doit vérifier.

Enfin, une remarque importante est que, dans le Calcul des Constructions Inductives, des preuves logiques peuvent apparaître à l'intérieur des types. Par exemple, on peut définir la fonction racine de type $(n : nat)(n > 0) \to nat$. Ce genre de situation pose problème. En effet, si l'on a deux preuves h_1 et h_2 de $4 > 0$ et deux spécifications $(racine\ 4\ h_1) = 2$ et $(racine\ 4\ h_2) = 2$, leur extraction faible sera identique (en l'occurrence, $(racine\ 4) = 2$). Différencier ces deux termes, c'est-à-dire passer d'une preuve de l'un à une preuve de l'autre est a priori un exercice impossible. Cependant, ce genre de situation n'arrive quand même pas très souvent et si nous ne voulons pas avoir à garder toutes les hypothèses logiques, il faut pouvoir écrire le programme annoté sans indication de preuves logiques dans les types. Nous verrons donc qu'une restriction à la méthode que nous proposons est l'identification de toutes les preuves logiques d'une même proposition.

3.1.2 Fonction d'extraction faible

Extraction faible sur les termes

Nous appellerons fonction d'extraction faible la nouvelle fonction d'extraction. Cette fonction est donc définie par niveaux comme la fonction d'extraction de [PM89a] (voir section 2.1.2). Elle s'applique à tous les termes sauf aux preuves logiques, puisqu'elle les supprime.

Définition 38 *Extraction faible sur les termes :*

$$\sigma_2(x) = \bar{x} \quad \textit{où } \bar{x} \textit{ est de type } \sigma_1(T_x) \textit{ si } x \textit{ est de type } T_x$$

$$\sigma_2([x : A]B) = \begin{cases} \sigma_2(B) & \textit{si } A \textit{ de type Prop} \\ [\bar{x} : \sigma_i(A)]\sigma_2(B) & \textit{sinon} \end{cases}$$

$$\sigma_2(A\ B) = \begin{cases} \sigma_2(A) & \textit{si } B \textit{ logique de niveau 2} \\ (\sigma_2(A)\ \sigma_i(B)) & \textit{sinon} \end{cases}$$

$$\sigma_2(\textit{Elim}(c, P)\{f_1| \ldots |f_n\}) = \textit{Elim}(\sigma_2(c), \sigma_1(P))\{\sigma_2(f_1)| \ldots |\sigma_2(f_n)\}$$

$$\sigma_2(\textit{Constr}(i, ind)) = \textit{Constr}(i, \sigma_1(ind))$$

$$\sigma_1(x) = \bar{x} \quad \textit{où } \bar{x} \textit{ est de type } \sigma_0(T_x) \textit{ si } x \textit{ est de type } T_x$$

$$\sigma_1((x : A)B) = (\bar{x} : \sigma_i(A))\sigma_1(B)$$

$$\sigma_1([x : A]B) = \begin{cases} \sigma_1(B) & \textit{si } A \textit{ de type Prop} \\ [\bar{x} : \sigma_i(A)]\sigma_1(B) & \textit{sinon} \end{cases}$$

$$\sigma_1(A\ B) = \begin{cases} \sigma_1(A) & \textit{si } B \textit{ logique de niveau 2} \\ (\sigma_1(A)\ \sigma_i(B)) & \textit{sinon} \end{cases}$$

$$\sigma_1(\textit{Elim}(c, P)\{f_1| \ldots |f_n\}) = \textit{Elim}(\sigma_2(c), \sigma_0(P))\{\sigma_1(f_1)| \ldots |\sigma_1(f_n)\}$$

$$\sigma_1(\textit{Ind}[p_1 : P_1; \ldots ; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}) =$$

$$\textit{Ind}[\overline{p_i} : \sigma_j(P_i)](\bar{X} : \sigma_0(A))\{\overline{c_1} : \sigma_1(C_1)| \ldots |\overline{c_m} : \sigma_1(C_m)\} \quad \textit{pour } P_i \textit{ n'est pas de type Prop}$$

$$\sigma_0(K) = K \quad \textit{où } K \textit{ est une sorte}$$

$$\sigma_0((x : A)B) = \begin{cases} \sigma_0(B) & \textit{si } A \textit{ de type Prop} \\ (\bar{x} : \sigma_0(A))\sigma_0(B) & \textit{sinon} \end{cases}$$

Comme nous l'avons déjà dit, d'une façon générale, nous supprimons toutes les dépendances par rapport à des preuves logiques (termes de niveau 2 logiques). On peut par exemple voir ceci sur le cas de l'application de niveau 1 et celle de niveau 2. Pour conserver la cohérence (par exemple, la substitutivité de l'extraction faible), nous allons supprimer aussi au niveau des abstractions les types logiques de niveau 1 (de type *Prop*). On peut voir ceci sur le cas de l'abstraction de niveau 1 et celle de niveau 2.

Il faut mettre l'accent sur une règle en particulier, la règle sur le produit de niveau 1. A cette étape, nous ne supprimons rien. En effet, dans le cas d'un produit de niveau 1, nous sommes en présence d'un type dépendant; nous ne voulons surtout rien extraire car c'est exactement le genre de choses que nous ne savons pas retrouver de façon déterministe. En effet, si nous revenons à l'exemple du chapitre précédent avec odd-even, si nous supprimons les informations $(even\ n)$ et $(odd\ n)$, nous n'avons aucune chance de les retrouver.

Pour les types inductifs, nous n'avons plus le problème des paramètres dans les constructeurs qui sont traités différemment. En effet, nous ne supprimons plus les paramètres de niveau 1. Nous ne supprimons que les dépendances par rapport aux preuves. En fait, le problème n'a pas vraiment disparu car il reste un problème au niveau des types mais on peut l'ignorer grâce au jugement de typage que nous définissons plus loin. Nous revenons donc à la notion des constructeurs sans les paramètres, qui est plus légère.

Extraction faible sur les contextes

Pour être complet, il faut aussi définir une extraction faible sur les contextes. Puisque nous avons défini l'extraction faible comme supprimant les dépendances des termes vis-à-vis des preuves logiques, nous n'avons qu'à supprimer dans le contexte toutes les variables logiques de niveau 2 et extraire faiblement les autres.

Définition 39 *Extraction faible sur les contextes :*

$\sigma(\Gamma)$, le contexte faiblement extrait de Γ, est obtenu en appliquant σ_1 sur le type des variables informatives de niveau 2 de Γ, σ_0 sur le type des variables de niveau 1 de Γ et en oubliant les variables logiques de niveau 2.

Exemple : Nous reprenons l'exemple vu au chapitre précédent pour l'extraction de [PM89a].

– Nous reprenons l'extraction faible de *sig*. On a :

$$\sigma_1(sig) = \mathsf{Ind}[\bar{A} : Set; \bar{P} : \bar{A} \rightarrow Prop](\bar{X} : Set)\{\overline{exist} : (\bar{x} : \bar{A})(\bar{P}\ \bar{x}) \rightarrow \bar{X}\}$$

On remarque que l'extraction faible laisse *sig* inchangé. Dans la suite, on ignorera les barres sur *sig* et *exist*.

– Nous reprenons l'extraction faible de *sumor*. On a :

$$\sigma_1(sumor) := \mathsf{Ind}[\bar{A} : Set; \bar{B} : Prop](\bar{X} : Set)\{\overline{inleft} : \bar{A} \rightarrow \bar{X} |\ \overline{inright} : \bar{B} \rightarrow \bar{X}\}$$

On remarque que l'extraction faible laisse *sumor* inchangé. Dans la suite, on ignorera les barres sur *sumor*, *inleft* et *inright*.

– Les deux exemples précédents restent inchangés. Si quelque chose devait disparaître, ce serait les variables de preuve à l'intérieur des types. Par exemple, si on a la fonction *racine* de type $(n : nat)(n > 0) \rightarrow nat$ alors, une spécification peut être $(racine\ 4\ h_4) = 2$ où h_4 est une preuve de $4 > 0$. Le terme extrait, lui, est $(racine\ 4) = 2$.

– L'extraction faible du type *nat* est toujours *nat*. On ignore les barres sur *nat*.

– Nous passons maintenant à l'exemple lui-même.

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
        (inright {m:nat|(S m)=0} (0=0) (refl_equal nat 0))
        [y:nat][H:{m:nat|(S m)=y}+{0=y}]
          (inleft {m:nat|(S m)=(S y)} (0=(S y))
             (exist nat [m:nat](S m)=(S y) y (refl_equal nat (S y))))
       end
```

Détaillons tout d'abord l'extraction du premier argument T_1 de l'élimination. Dans tout la suite, nous n'indiquons pas de barre sur les constructeurs *inleft*, *inright* et *exist* puisque nous avons fait remarquer plus haut que *sig* et *sumor* restaient inchangés par extraction faible.

$$\sigma_2(T_1) = (\sigma_2(inright)\ \sigma_1(\{m : nat|(S\ m) = 0\})\ \sigma_1(0 = 0))$$

car `refl_equal` est logique de niveau 2. Donc,

$$\sigma_2(T_1) = (inright\ \{m : nat|(S\ m) = 0\}\ (0 = 0))$$

Détaillons maintenant l'extraction du second argument T_2 de l'élimination.

$$\begin{aligned}
\sigma_2(T_2) \quad = \quad & [\bar{y} : nat][\bar{H} : \sigma_1(\{m : nat|(S\ m) = \bar{y}\} + \{0 = \bar{y}\})] \\
& \sigma_2((inleft\ \{m : nat|(S\ m) = (S\ \bar{y})\}\ (O = (S\ \bar{y})) \\
& (exist\ nat\ [m : nat]((S\ m) = (S\ \bar{y}))\ \bar{y}\ (refl_equal\ nat\ (S\ \bar{y}))))) \\
= \quad & [\bar{y} : nat][\bar{H} : \{m : nat|(S\ m) = \bar{y}\} + \{0 = \bar{y}\}] \\
& (\sigma_2(inleft)\ \{m : nat|(S\ m) = (S\ \bar{y})\}\ \sigma_1(0 = (S\ \bar{y})) \\
& (\sigma_2(exist)\ nat\ \sigma_1([m : nat]((S\ m) = (S\ \bar{y})))\ \overline{y}))
\end{aligned}$$

car `refl_equal` est sans contenu calculatoire. Donc,

$$\begin{aligned}\sigma_2(T_2) \quad = \quad & [\bar{y} : nat][\bar{H} : \{m : nat|(S\ m) = \bar{y}\} + \{0 = \bar{y}\}] \\ & (inleft\ \{m : nat|(S\ m) = (S\ \bar{y})\}\ (O = (S\ \bar{y})) \\ & (exist\ nat\ [m : nat]((S\ m) = (S\ \bar{y}))\ \overline{y}))\end{aligned}$$

Pour pouvoir calculer l'extraction totale du terme T, il ne nous manque que l'extraction de n qui est $\overline{n}$ et celle du prédicat de l'élimination.

$$\begin{aligned}\sigma_1([n0 : nat]\{m : nat|(S\ m) = n0\} + \{0 = n0\}) \quad = \\ [\overline{n0} : nat]\{m : nat|(S\ m) = \overline{n0}\} + \{0 = \overline{n0}\}\end{aligned}$$

L'extraction du terme T est finalement le terme suivant :

```
[n:nat]<[n0:nat]{m:nat|(S m)=n0}+{0=n0}>Match n with
        (inright {m:nat|(S m)=0} (0=0))
        [y:nat][H:{m:nat|(S m)=y}+{0=y}]
          (inleft {m:nat|(S m)=(S y)} (0=(S y))
                  (exist nat [m:nat](S m)=(S y) y))
      end
```

si l'on oublie les barres sur les constantes et qu'on sous-entend qu'on est dans le contexte faiblement extrait.

3.1.3 Nouveau jugement de typage

Dans le chapitre précédent, nous avons décrit l'extraction. Nous avons vu que l'extraction transformait des termes du Calcul des Constructions Inductives en termes de ${F_\omega}^{Ind}$ (perte des types dépendants). Or, le typage de ${F_\omega}^{Ind}$ est inclus dans le typage du Calcul des Constructions Inductives. Dans chacun des deux mondes, nous avons donc des jugements de typage différents bien que très proches. Dans le cas de l'extraction faible, nous transformons des termes du Calcul des Constructions Inductives en d'autres termes du Calcul des Constructions Inductives (nous avons toujours les types dépendants de preuves logiques). Mais, on voit bien que les règles de typage ne peuvent plus être celles du Calcul des Constructions Inductives car on a des termes non-standards représentant des programmes (monde extrait) annotés avec des spécifications (types dépendants du monde des preuves). L'intérêt de tels termes est qu'ils contiennent plus d'informations que les termes extraits de ${F_\omega}^{Ind}$ et ces informations permettront l'inversion grâce à une nouvelle notion de typage. Nous définissons donc un nouveau jugement de typage pour ces termes. L'intérêt de pouvoir typer de tels termes est une détection précoce des erreurs. En effet, on pourrait imaginer avoir des termes purs. Mais, avec le typage, on peut écarter les programmes mal typés ainsi que les programmes dont le type n'est pas cohérent avec la spécification. On remarquera que les règles sont un peu spéciales car certaines sont gardées par le niveau et l'information d'un de leur terme (3.1,3.2).

Nous définissons le jugement de typage $\Gamma \vdash t \in P$ qui a l'interprétation suivante : t est un programme annoté dont les annotations sont des spécifications (c'est-à-dire des propriétés contenant des parties logiques et calculatoires) bien typées au sens usuel, cohérent avec la spécification P (c'est-à-dire l'extraction au sens de [PM89a] de t est de type $\varepsilon(P)$). Ce jugement concerne les objets informatifs et les propositions logiques. Il n'est défini que pour P qui n'est pas logique de niveau 1 (qui n'est pas de type *Prop*). Ce jugement de typage représente un PTS avec types inductifs proche du Calcul des Constructions avec types inductifs mais ce sont les deux règles 3.3 et 3.4 qui font la différence. Ces règles permettent de rajouter ou de supprimer à tout moment des produits logiques de niveau 1 dans les types. Ce sont ces deux règles qui permettent de typer les programmes annotés.

Remarque : Identification des types inductifs isomorphes.

Il est important de noter que nous souhaitons une propriété importante de ce nouveau jugement de typage. Nous voulons identifier les types inductifs isomorphes. En particulier, nous voulons identifier *bool* et $(sumbool\ A\ B)$ dont l'extraction de [PM89a] commune est $\mathsf{Ind}(X : Set)\{\overline{left} : X |\ \overline{right} : X\}$. Donc, nous voulons $\Gamma \vdash t \in bool$ ssi $\Gamma \vdash t \in (sumbool\ A\ B)$. En particulier, $\Gamma \vdash true \in (sumbool\ A\ B)$ et $\Gamma \vdash left \in bool$.

Par les règles 3.3 et 3.4, on identifie les types dont les extractions sont identiques. Donc, on réalise le souhait d'identification des types inductifs isomorphes. Nous reviendrons plus en détail sur ce point dans le chapitre 4.

Définition 40

$$\overline{[]\ bien\ form\acute{e}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ bien\ form\acute{e}} \quad s \in \mathcal{S} = \{Prop, Set, Type, TypeSet\}$$

$$\frac{\Gamma\ bien\ form\acute{e}}{\Gamma \vdash Prop \in Type}$$

$$\frac{\Gamma\ bien\ form\acute{e}}{\Gamma \vdash Set \in TypeSet}$$

$$\frac{\Gamma\ bien\ form\acute{e} \quad x : T \in \Gamma}{\Gamma \vdash x \in T}$$

$$\frac{\Gamma \vdash T \in s \quad \Gamma[x : T] \vdash T' \in s'}{\Gamma \vdash (x : T)T' \in s'} \quad (s, s') \in \mathcal{R} = \mathcal{S} \times \mathcal{S} \setminus \{(Prop, Type), (Prop, TypeSet)\} \tag{3.1}$$

$$\frac{\Gamma \vdash (x : T)T' \in s \quad \Gamma[x : T] \vdash t \in T'}{\Gamma \vdash [x : T]t \in (x : T)T'} \quad s \in \mathcal{S}$$

$$\frac{\Gamma \vdash t \in (x : A)B \quad \Gamma \vdash t' \in A}{\Gamma \vdash (t\ t') \in B[x \leftarrow t']} \quad A\ \textit{de niveau 0 ou informatif de niveau 1} \tag{3.2}$$

$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \rightarrow U}{\Gamma \vdash t \in U} \tag{3.3}$$

$$\frac{\Gamma \vdash t \in U \quad \Gamma \vdash A \in Prop}{\Gamma \vdash t \in A \rightarrow U} \tag{3.4}$$

$$\frac{\Gamma \vdash T \in s \quad \Gamma \vdash T' \in s' \quad \Gamma \vdash t \in T \quad T =_{\beta\iota} T'}{\Gamma \vdash t \in T'}$$

$$\frac{\begin{array}{c}(\forall i = 1 \ldots m) \\ \Gamma[p_j : P_j] \vdash A \in s \quad \Gamma[p_j : P_j; X : A] \vdash C_i \in s' \quad constructeur(C_i, X)\end{array}}{\Gamma \vdash \mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1 | \ldots | c_m : C_m\} \in (p_1 : P_1) \ldots (p_n : P_n)A} \quad \not\exists P_j : Prop$$

$$\frac{\Gamma \vdash Ind \equiv \textsf{\textit{Ind}}[\vec{p} : \vec{P}](X : A)\{c_1 : C_1 | \dots c_m : C_m\} \in B \quad 1 \leq i \leq m}{\Gamma \vdash \textsf{\textit{Constr}}(i, Ind) \in (p_1 : P_1) \dots (p_n : P_n) C_i[X \leftarrow (Ind\ p_1 \dots p_n)]}$$

$$\frac{\Gamma \vdash c \in (I\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q \in (\vec{x} : \vec{A})(I\ \vec{p}\ \vec{x}) \rightarrow s' \quad \begin{array}{c}(\forall i = 1 \dots m)\\ \Gamma \vdash f_i \in C_i\{(I\ \vec{p}), Q, (\textsf{\textit{Constr}}(i, I)\ \vec{p})\}_\sigma\end{array}}{\Gamma \vdash \textsf{\textit{Elim}}(c, Q)\{f_1 | \dots | f_m\} \in (Q\ \vec{a}\ c)}$$

Notons une petite modification sur la définition du terme $C_i\{X, Q, c\}_\sigma$. En effet, a priori, nous ne voulons pas mentionner les variables de preuves dans les constructeurs. Par exemple, si un constructeur est $c : (n : nat)(n > 0) \rightarrow C$, le principe d'élimination doit être $(n : nat)(n > 0) \rightarrow (P\ (c\ n))$ et non pas $(n : nat)(h : n > 0)(P\ (c\ n\ h))$ comme avec la définition de $C_i\{X, Q, c\}$ du chapitre 1. Nous modifions donc légèrement cette définition pour pallier à ce problème et dans la suite, la notation $C_i\{X, Q, c\}_\sigma$ fera référence à cette nouvelle définition.

Définition 41 *Soient s et s' deux sortes, $A \equiv (\vec{x} : \vec{A})s$, X une variable de type A, Q une variable de type $(\vec{x} : \vec{A})(X\ \vec{x}) \rightarrow s'$, C un type de constructeur par rapport à X et c un terme de type C, on définit $C\{X, Q, c\}_\sigma$ par récurrence sur C (avec les mêmes conditions que pour la définition 26) :*

$$(P \rightarrow C)\{X, Q, c\}_\sigma = (p : P)P\{X, Q, p\}_\sigma \rightarrow C\{X, Q, (c\ p)\}_\sigma$$
$$(x : M)C\{X, Q, c\}_\sigma = \begin{cases} M \rightarrow C\{X, Q, c\}_\sigma & si\ M : Prop \\ (x : M)C\{X, Q, (c\ x)\}_\sigma & sinon \end{cases}$$
$$(X\ \vec{a})\{X, Q, c\}_\sigma = (Q\ \vec{a}\ c)$$

Exemple : Typage de l'extraction faible de l'exemple précédent.

Nous allons uniquement détailler le typage du terme $\sigma(T_1)$, premier argument de l'élimination. Nous avons les informations de type suivantes :

- $\overline{sig}\ :\ (A : Set)(A \rightarrow Prop) \rightarrow Set$;
- $nat\ :\ Set$;
- $[m : nat]((S\ m) = n0)\ :\ nat \rightarrow Prop$ car l'égalité sur les entiers est de type $nat \rightarrow nat \rightarrow Prop$;
- $\overline{inright}\ :\ (A : Set)(B : Prop)(B \rightarrow (sumor\ A\ B))$.

Nous avons alors l'arbre de dérivation de typage suivant :

$$\cfrac{\cfrac{\cfrac{\vdash \{m : nat | (S\ m) = 0\} \in Set \quad \vdash \overline{inright} \in (A : Set)(B : Prop)(B \rightarrow A + \{B\})}{\vdash (\overline{inright}\ \{m : nat | (S\ m) = 0\}) \in (B : Prop)(B \rightarrow \{m : nat | (S\ m) = 0\} + \{B\})} \quad \vdash 0 = 0 \in Prop}{\vdash \sigma(T_1) \in (0 = 0) \rightarrow \{m : nat | (S\ m) = 0\} + \{0 = 0\}}}{\vdash \sigma(T_1) \in \{m : nat | (S\ m) = 0\} + \{0 = 0\}}$$

Les feuilles de cet arbre sont triviales avec les points ci-dessus énoncés. On remarque l'utilisation de la règle 3.1 sans laquelle on ne saurait pas typer correctement. On peut ensuite suivre le même cheminement pour le reste du terme T et on obtient :

$$T \in \{m : nat | (S\ m) = n\} + \{0 = n\}$$

Remarques :

1. Dans la règle 3.1, on n'a pas besoin des couples $(Prop, Type)$ et $(Prop, TypeSet)$ car ce sont les formations des types dépendants de preuves logiques, qui ont été supprimés.
2. Dans tout ce qui suit, il faut faire attention que, contrairement à ε, σ s'applique à des termes logiques (sauf les preuves).

3.2 Propriétés

Dans cette partie, nous allons montrer que le nouveau jugement de typage que nous avons introduit vérifie toutes les bonnes propriétés que nous attendions. Ainsi, nous allons montrer que, si un terme est bien typé dans ce nouveau sens, alors son extraction est bien typée dans le Calcul des Constructions Inductives au sens usuel. Ce qui correspond bien à ce que nous souhaitions, puisque cela implique que l'extraction faible est moins restrictive que l'extraction de [PM89a]. Nous montrons aussi, comme pour l'extraction de [PM89a], que, si un terme de preuve est bien typé dans le Calcul des Constructions Inductives, alors son extraction faible est bien typée dans le nouveau jugement de typage.

3.2.1 Cohérence du nouveau jugement de typage

Nous voulons montrer que le nouveau jugement de typage $\in$ n'introduit pas d'incohérence. En effet, nous savons que $\Gamma \vdash t : P \Rightarrow \varepsilon(\Gamma) \vdash \varepsilon(t) : \varepsilon(P)$. L'interprétation d'une telle propriété est que, si on a une preuve t d'une spécification P, alors le programme extrait de cette preuve est bien typé de type l'extraction de la spécification.

Pour notre nouveau jugement de typage, nous souhaitons que, si $\Gamma \vdash t \in P$, alors t est cohérent avec P. Ceci se traduit par $\Gamma \vdash t \in P \Rightarrow \varepsilon(\Gamma) \vdash \varepsilon(t) : \varepsilon(P)$.

Nous pouvons énoncer une première propriété. En effet, cette remarque se justifie par le fait que σ ne retire que les preuves logiques alors que ε retire tout le reste des objets logiques ainsi que les termes informatifs dans les propositions. De plus, d'après l'explication informelle du début de ce chapitre, réappliquer l'extraction forte à l'extraction faible donne trivialement l'extraction forte.

Proposition 2 $\forall t, \varepsilon(\sigma(t)) = \varepsilon(t)$.

Remarque : on prendra comme convention par la suite que $\varepsilon_{-1}(TypeSet) = TypeSet$.

Nous pouvons montrer la cohérence avec un énoncé un peu différent de ce que nous avions annoncé.

Proposition 3 *Cohérence du nouveau jugement de typage par rapport à l'extraction forte :*
Si $\Gamma \vdash t \in P$ *et* P *est informatif alors* $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}(P)$.

Preuve. Par récurrence sur la longueur de la dérivation du nouveau jugement de typage. On passe sur les règles de bonne formation des contextes qui sont triviales. Les règles sont les mêmes que celles du typage standard sauf les règles 3.2, 3.3 et 3.4. Nous ne détaillerons donc que ces trois dernières.

– Règle de formation de l'application (règle 3.2) :

$$\frac{\Gamma \vdash t \in (x:T)T' \quad \Gamma \vdash t' \in T}{\Gamma \vdash (t\ t') \in T'[x \leftarrow t']}$$

On a comme hypothèse de récurrence $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}((x:T)T')$ car T' est de type *TypeSet* donc $(x:T)T'$ aussi. On distingue deux cas suivant le niveau j de t'.

- $j \leq i$ et t' informatif.
  On a alors $\varepsilon_i(t\ t') = (\varepsilon_i(t)\ \varepsilon_j(t'))$ et $\varepsilon_{i-1}((x:T)T') = (x:\varepsilon_{j-1}(T))\varepsilon_{i-1}(T')$. On a aussi l'hypothèse de récurrence pour t' donnant $\varepsilon_i(\Gamma) \vdash \varepsilon_j(t') : \varepsilon_{j-1}(T)$. Donc, on a $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t\ t') : \varepsilon_{i-1}(T'[x \leftarrow t'])$.
- Sinon, $j > i$ ou t' logique.
  On a alors $\varepsilon_i(t\ t') = \varepsilon_{i-1}(t)$, $\varepsilon_{i-1}(T'[x \leftarrow t']) = \varepsilon_{i-1}(T')$ et $\varepsilon_{i-1}((x:T)T') = \varepsilon_{i-1}(T')$. Donc, on a $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t\ t') : \varepsilon_{i-1}(T'[x \leftarrow t'])$.

- Règle 3.3 :
$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \rightarrow U}{\Gamma \vdash t \in U}$$
  On a comme hypothèse de récurrence $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}(A \rightarrow U)$. Comme $\Gamma \vdash A \in Prop$, A est logique, donc $\varepsilon_{i-1}(A \rightarrow U) = \varepsilon_{i-1}(U)$. Donc, $\varepsilon_i(\Gamma) \vdash \varepsilon_i(t) : \varepsilon_{i-1}(U)$.

- Règle 3.4 :
$$\frac{\Gamma \vdash t \in U \quad \Gamma \vdash A \in Prop}{\Gamma \vdash t \in A \rightarrow U}$$
  Cas similaire au précédent.

□

3.2.2 Substitutivité de l'extraction faible

Comme pour toute fonction qui s'applique à des λ-termes, nous avons besoin d'un lemme de substitutivité de l'extraction faible. Dans ce cas précis, le lemme de substitutivité prend une forme un peu spéciale du fait de la définition de σ en fonction de l'information du terme et de son niveau. En fait, nous avons deux lemmes de substitutivité pour traiter tous les cas possibles.

Proposition 4 *Premier lemme de substitutivité :*
Pour tout terme t de niveau 0, 1 ou informatif de niveau 2, on a :

$$\sigma_i(T[x \leftarrow t]) = \sigma_i(T)[x \leftarrow \sigma_j(t)]$$

où i est le niveau de T et j le niveau de t et de x. De plus, x et t doivent être de même type.

Preuve. Par récurrence sur la structure de T.

Niveau 2 :

- T est la variable x :

$$\begin{array}{rcl} \sigma_2(x[x \leftarrow t]) & = & \sigma_2(t) \\ \sigma_2(x)[x \leftarrow \sigma_2(t)] & = & x[x \leftarrow \sigma_2(t)] = \sigma_2(t) \end{array}$$

- Le cas où T est une variable différente de x est trivial.

- $T = [x : A]B$:

Cas 1 : A de niveau 0 ou informatif de niveau 1.

$$\begin{array}{rcl}
\sigma_2(([x:A]B)[y \leftarrow t]) & = & \sigma_2([x:A[y \leftarrow t]]B[y \leftarrow t]) \\
& = & [x:\sigma_i(A[y \leftarrow t])]\sigma_2(B[y \leftarrow t]) \\
& = & [x:\sigma_i(A)[y \leftarrow \sigma_j(t)]]\sigma_2(B)[y \leftarrow \sigma_j(t)] \ \text{ par HR} \\
\sigma_2([x:A]B)[y \leftarrow \sigma_j(t)] & = & ([x:\sigma_i(A)]\sigma_2(B))[y \leftarrow \sigma_j(t)] \\
& = & [x:\sigma_i(A)[y \leftarrow \sigma_j(t)]]\sigma_2(B)[y \leftarrow \sigma_j(t)]
\end{array}$$

Cas 2 : A logique de niveau 1 (de type *Prop*).

$$\begin{array}{rcl}
\sigma_2(([x:A]B)[y \leftarrow t]) & = & \sigma_2([x:A[y \leftarrow t]]B[y \leftarrow t]) \\
& = & \sigma_2(B[y \leftarrow t]) \\
& = & \sigma_2(B)[y \leftarrow \sigma_j(t)] \ \text{ par HR} \\
& = & \sigma_2([x:A]B)[y \leftarrow \sigma_j(t)]
\end{array}$$

- $T = (A\ B)$:

Cas 1 : B de niveau 0 ou 1 ou informatif de niveau 2.

$$\begin{array}{rcl}
\sigma_2((A\ B)[y \leftarrow t]) & = & \sigma_2((A[y \leftarrow t]\ B[y \leftarrow t])) \\
& = & (\sigma_2(A[y \leftarrow t])\ \sigma_i(B[y \leftarrow t])) \\
& = & (\sigma_2(A)[y \leftarrow \sigma_j(t)]\ \sigma_i(B)[y \leftarrow \sigma_j(t)]) \ \text{ par HR} \\
\sigma_2((A\ B))[y \leftarrow \sigma_j(t)] & = & (\sigma_2(A)\ \sigma_i(B))[y \leftarrow \sigma_j(t)] \\
& = & (\sigma_2(A)[y \leftarrow \sigma_j(t)]\ \sigma_2(B)[y \leftarrow \sigma_j(t)])
\end{array}$$

Cas 2 : B logique de niveau 2.

$$\begin{array}{rcl}
\sigma_2((A\ B)[y \leftarrow t]) & = & \sigma_2(A[y \leftarrow t]\ B[y \leftarrow t]) \\
& = & \sigma_2(A[y \leftarrow t]) \\
& = & \sigma_2(A)[y \leftarrow \sigma_j(t)] \ \text{ par HR} \\
& = & \sigma_2(A\ B)[y \leftarrow \sigma_j(t)]
\end{array}$$

- $T = \mathsf{Elim}(c, Q)\{f_1|\ldots|f_n\}$:

$$\begin{array}{rcl}
\sigma_{i+1}(\mathsf{Elim}(c,Q)\{f_1|\ldots|f_n\}[y \leftarrow t]) & = & \sigma_{i+1}(\mathsf{Elim}(c[y \leftarrow t], Q[y \leftarrow t])\{f_i[y \leftarrow t]\}) \\
& = & Elim(\sigma_2(c[y \leftarrow t]), \sigma_i(Q[y \leftarrow t])) \\
& & \{\sigma_{i+1}(f_1[y \leftarrow t])|\ldots|\sigma_{i+1}(f_n[y \leftarrow t])\}) \\
& = & Elim(\sigma_2(c)[y \leftarrow \sigma_j(t)], \sigma_i(Q)[y \leftarrow \sigma_j(t)]) \\
& & \{\sigma_{i+1}(f_i)[y \leftarrow \sigma_j(t)]\} \ \text{ par HR} \\
\sigma_{i+1}(\mathsf{Elim}(c,Q)\{f_1|\ldots|f_n\})[y \leftarrow \sigma_j(t)] & = & \mathsf{Elim}(\sigma_2(c), \sigma_i(Q))\{\sigma_{i+1}(f_i)\}[y \leftarrow \sigma_j(t)] \\
& = & Elim(\sigma_2(c)[y \leftarrow \sigma_j(t)], \sigma_i(Q)[y \leftarrow \sigma_j(t)]) \\
& & \{\sigma_{i+1}(f_i)[y \leftarrow \sigma_j(t)]\}
\end{array}$$

- $T = \mathsf{Constr}(i, ind)$:

$$\begin{array}{rcl}
\sigma_2(\mathsf{Constr}(i, ind)[y \leftarrow t]) & = & \sigma_2(\mathsf{Constr}(i, ind[y \leftarrow t])) \\
 & = & \mathsf{Constr}(i, \sigma_1(ind[y \leftarrow t])) \\
 & = & \mathsf{Constr}(i, \sigma_1(ind)[y \leftarrow \sigma_j(t)]) \text{ par HR} \\
\sigma_2(\mathsf{Constr}(i, ind))[y \leftarrow \sigma_j(t)] & = & \mathsf{Constr}(i, \sigma_1(ind))[y \leftarrow \sigma_j(t)] \\
 & = & \mathsf{Constr}(i, \sigma_1(ind)[y \leftarrow \sigma_j(t)])
\end{array}$$

Niveau 1 :

- T est une variable x : même chose que pour le niveau 2.
- $T = [x : A]B$: même chose que pour le niveau 2.
- $T = (A\ B)$: même chose que pour le niveau 2.
- $T = \mathsf{Elim}(c, Q)\{f_1| \ldots |f_n\}$: même chose que pour le niveau 2.
- $T = (x : A)B$:

$$\begin{array}{rcl}
\sigma_1(((x : A)B)[y \leftarrow t]) & = & \sigma_1((x : A[y \leftarrow t])B[y \leftarrow t]) \\
 & = & (x : \sigma_i(A[y \leftarrow t]))\sigma_1(B[y \leftarrow t]) \\
 & = & (x : \sigma_i(A)[y \leftarrow \sigma_j(t)])\sigma_1(B)[y \leftarrow \sigma_j(t)] \text{ par HR} \\
\sigma_1((x : A)B)[y \leftarrow \sigma_j(t)] & = & (x : \sigma_i(A))\sigma_1(B)[y \leftarrow \sigma_j(t)] \\
 & = & (x : \sigma_i(A)[y \leftarrow \sigma_j(t)])\sigma_1(B)[y \leftarrow \sigma_j(t)]
\end{array}$$

- $T = \mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}$:

$$\begin{array}{rl}
 & \sigma_1(\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\}[y \leftarrow t]) \\
= & \sigma_1(\mathsf{Ind}[p_i : P_i[y \leftarrow t]](X : A[y \leftarrow t])\{c_1 : C_1[y \leftarrow t]| \ldots |c_m : C_m[y \leftarrow t]\}) \\
= & \mathsf{Ind}[\sigma(p_i : P_i[y \leftarrow t])](X : \sigma_0(A[y \leftarrow t]))\{c_i : \sigma_1(C_i[y \leftarrow t])\} \\
= & \mathsf{Ind}[\sigma(p_i : P_i[y \leftarrow t])](X : \sigma_0(A)[y \leftarrow \sigma_j(t)]) \\
 & \{c_1 : \sigma_1(C_1)[y \leftarrow \sigma_j(t)]| \ldots |c_m : \sigma_1(C_m)[y \leftarrow \sigma_j(t)]\} \text{ par HR} \\
= & \sigma_1(\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1| \ldots |c_m : C_m\})[y \leftarrow \sigma_j(t)]
\end{array}$$

Niveau 0 :

- Le cas des sortes est évident.
- Le cas $T = (x : A)B$ est identique au cas $T = [x : A]B$ pour le niveau 1.

□

Remarque : dans la preuve précédente, on utilise deux propriétés :

- Si A est de niveau k alors $A[y \leftarrow t]$ aussi,
- Si A est informatif alors $A[y \leftarrow t]$ aussi.

Proposition 5 *Deuxième lemme de substitutivité :*
Pour tout terme t logique de niveau 2, on a :

$$\sigma_i(T[x \leftarrow t]) = \sigma_i(T)$$

où i est le niveau de T et avec les niveaux et les informations de x et t cohérents.

Preuve. Suit exactement le même schéma que pour le premier lemme. □

3.2.3 Cohérence de l'extraction faible

Proposition 6 $\forall \Gamma$, $\forall t$, $\forall P$, *tel que P n'est pas de type Prop,* $\Gamma \vdash t : P \Rightarrow \sigma(\Gamma) \vdash \sigma_i(t) \in \sigma_{i-1}(P)$.

Preuve. Par récurrence sur la longueur de la dérivation du jugement de typage standard.

- On passe sur les règles de bonne formation des contextes qui sont triviales.
- Pour la règle de formation du produit, on se référera à celle de l'abstraction ci-après. Elles sont très similaires.
- Règle de formation de l'abstraction :

$$\frac{\Gamma \vdash (x : T)T' : s \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}$$

  On veut montrer que $\sigma(\Gamma) \vdash \sigma_i([x : T]t) \in \sigma_{i-1}((x : T)T')$.

  Cas 1 : $(x : T)T'$ de niveau 1.
  On a alors $\sigma_1((x : T)T') = (x : \sigma_i(T))\sigma_1(T')$ et $\sigma(\Gamma[x : T]) = \sigma(\Gamma)[x : \sigma_i(T)]$.

  Cas 1.1 : T de niveau 0 ou informatif de niveau 1.
  On a alors $\sigma_2([x : T]t) = [x : \sigma_i(T)]\sigma_2(t)$. Les hypothèses de récurrence ont la forme :

$$\begin{cases} \sigma(\Gamma) \vdash (x : \sigma_i(T))\sigma_1(T') \in \sigma_0(s) \\ \sigma(\Gamma)[x : \sigma_i(T)] \vdash \sigma_2(t) \in \sigma_1(T') \end{cases}$$

  On peut déduire par les règles de typage : $\sigma(\Gamma) \vdash [x : \sigma_i(T)]\sigma_2(t) \in (x : \sigma_i(T))\sigma_1(T')$.

  Cas 1.2 : T logique de niveau 1 (de type *Prop*).
  On a alors $\sigma_2([x : T]t) = \sigma_2(t)$ et $\sigma(\Gamma[x : T]) = \sigma(\Gamma)$. Les hypothèses de récurrence ont la forme :

$$\begin{cases} \sigma(\Gamma) \vdash (x : \sigma_1(T))\sigma_1(T') \in \sigma_0(s) \\ \sigma(\Gamma) \vdash \sigma_2(t) \in \sigma_1(T') \end{cases}$$

  Or, $\sigma(\Gamma) \vdash \sigma_1(T) \in Prop$ car T est de type *Prop*. Donc, par 3.4, $\sigma(\Gamma) \vdash \sigma_2(t) \in (x : \sigma_1(T))\sigma_1(T')$.

  Cas 2 : $(x : T)T'$ de niveau 0.

  Cas 2.1 : T de niveau 0 ou informatif de niveau 1.
  On a alors $\sigma_0((x : T)T') = (x : \sigma_i(T))\sigma_0(t)$ et $\sigma_1([x : T]t) = [x : \sigma_i(T)]\sigma_1(t)$. Les hypothèses de récurrence ont la forme :

$$\begin{cases} \sigma(\Gamma) \vdash (x : \sigma_i(T))\sigma_0(T') \in \sigma_0(s) \\ \sigma(\Gamma)[x : \sigma_i(T)] \vdash \sigma_1(t) \in \sigma_0(T') \end{cases}$$

  On peut déduire par les règles de typage : $\sigma(\Gamma) \vdash [x : \sigma_i(T)]\sigma_1(t) \in (x : \sigma_i(T))\sigma_0(T')$.

Cas 2.2 : T logique de niveau 1 (de type *Prop*).

On a alors $\sigma_0((x:T)T') = \sigma_0(T')$, $\sigma_1([x:T]t) = \sigma_1(t)$ et $\sigma(\Gamma[x:T]) = \sigma(\Gamma)$. Donc, par hypothèse de récurrence, $\sigma(\Gamma) \vdash \sigma_1(t) \in \sigma_0(T')$.

– Règle de formation de l'application :

$$\frac{\Gamma \vdash t : (x:T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

On veut montrer que $\sigma(\Gamma) \vdash \sigma_i(t\ t') \in \sigma_{i-1}(T[x \leftarrow t'])$.

Cas 1 : T' de niveau 1 donc t de niveau 2.

Cas 1.1 : T de niveau 0 ou informatif de niveau 1.

On a alors $\sigma_2(t\ t') = (\sigma_2(t)\ \sigma_i(t'))$ et $\sigma_1(T'[x \leftarrow t']) = \sigma_1(T')[x \leftarrow \sigma_i(t')]$ par le premier lemme de substitutivité. Les hypothèses de récurrence ont la forme :

$$\begin{cases} \sigma(\Gamma) \vdash \sigma_2(t) \in (x:\sigma_i(T))\sigma_1(T') \\ \sigma(\Gamma) \vdash \sigma_i(t') \in \sigma_{i-1}(T) \end{cases}$$

On peut déduire par les règles de typage : $\sigma(\Gamma) \vdash (\sigma_2(t)\ \sigma_i(t')) \in \sigma_1(T')[x \leftarrow \sigma_i(t')]$.

Cas 1.2 : T logique de niveau 1 (de type *Prop*).

On a alors $\sigma_2(t\ t') = \sigma_2(t)$ et $\sigma_1(T'[x \leftarrow t']) = \sigma_1(T')$ par le second lemme de substitutivité. L'hypothèse de récurrence sur t a la forme :

$$\sigma(\Gamma) \vdash \sigma_2(t) \in (x:\sigma_1(T))\sigma_1(T')$$

Or, $\sigma(\Gamma) \vdash \sigma_1(T) \in Prop$ car T est de type *Prop*. Donc, par 3.3, on a $\sigma(\Gamma) \vdash \sigma_2(t) \in \sigma_1(T')$.

Cas 2 : T' de niveau 0 donc t de niveau 1.

Cas 2.1 : T de niveau 0 ou informatif de niveau 1.

On a alors $\sigma_1(t\ t') = (\sigma_1(t)\ \sigma_i(t'))$ et $\sigma_0(T'[x \leftarrow t']) = \sigma_0(T')[x \leftarrow \sigma_i(t')]$ par le premier lemme de substitutivité. Les hypothèses de récurrence ont la forme :

$$\begin{cases} \sigma(\Gamma) \vdash \sigma_1(t) \in (x:\sigma_i(T))\sigma_0(T') \\ \sigma(\Gamma) \vdash \sigma_i(t') \in \sigma_{i-1}(T) \end{cases}$$

On peut déduire par les règles de typage : $\sigma(\Gamma) \vdash (\sigma_1(t)\ \sigma_i(t')) \in \sigma_0(T')[x \leftarrow \sigma_i(t')]$.

Cas 2.2 : T logique de niveau 1 (de type *Prop*).

On a alors $\sigma_1(t\ t') = \sigma_1(t)$ et $\sigma_0(T'[x \leftarrow t']) = \sigma_0(T')$ par le second lemme de substitutivité. Donc, la première hypothèse de récurrence donne $\sigma(\Gamma) \vdash \sigma_1(t) \in \sigma_0(T')$.

– La règle de conversion, la règle de formation de type inductif et la règle de formation de constructeur sont triviales.

– Règle de formation de l'élimination :

$$\frac{\Gamma \vdash c : (I\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q : (\vec{x}:\vec{A})(I\ \vec{p}\ \vec{x}) \to s' \quad \overset{(\forall i = 1\ldots m)}{\Gamma \vdash f_i : C_i\{(I\ \vec{p}), Q, (\mathsf{Constr}(i,I)\ \vec{p})\}}}{\Gamma \vdash \mathsf{Elim}(c,Q)\{f_1|\ldots|f_m\} : (Q\ \vec{a}\ c)}$$

On veut montrer que $\sigma(\Gamma) \vdash \sigma_{i+1}(\mathsf{Elim}(c, Q)\{f_1| \dots |f_m\}) \in \sigma_i(Q\ \vec{a}\ c)$.

On a alors $\sigma_{i+1}(\mathsf{Elim}(c, Q)\{f_1| \dots |f_m\}) = \mathsf{Elim}(\sigma_2(c), \sigma_i(Q))\{\sigma_{i+1}(f_1)| \dots |\sigma_{i+1}(f_m)\}$ et $\sigma_i(Q\ \vec{a}\ c) = (\sigma_i(Q)\ \sigma_j(\vec{a})\ \sigma_2(c))$.

On peut parler de $\sigma_2(c)$ car c ne peut pas être logique de niveau 2. En effet, si c est logique de niveau 2, alors I est logique de niveau 1, donc il s'agit d'une élimination de type $NoDep_{Prop,Prop}$. L'élimination est alors logique de niveau 2 et l'extraction faible ne s'applique pas à des termes logiques de niveau 2.

Les hypothèses de récurrence sont :

$$\begin{cases} \sigma(\Gamma) \vdash \sigma_2(c) \in (\sigma_1(I)\ \sigma_j(\vec{p})\ \sigma_j(\vec{a})) \\ \sigma(\Gamma) \vdash \sigma_i(Q) \in (x : \sigma_{j-1}(\vec{A}))(\sigma_1(I)\ \sigma_j(\vec{p})\ \sigma_j(\vec{x})) \to \sigma_0(s') \\ \sigma(\Gamma) \vdash \sigma_{i+1}(f_i) \in C_i\{(\sigma_1(I)\ \sigma_j(\vec{p})), \sigma_i(Q), (\mathsf{Constr}(i, \sigma_1(I))\ \sigma_j(\vec{p}))\}_\sigma \quad \forall i = 1 \dots m \end{cases}$$

On peut déduire par les règles de typage :

$$\sigma(\Gamma) \vdash \mathsf{Elim}(\sigma_2(c), \sigma_i(Q))\{\sigma_{i+1}(f_1)| \dots |\sigma_{i+1}(f_m)\} \in (\sigma_i(Q)\ \sigma_j(\vec{a})\ \sigma_2(c))$$

□

3.3 Inversion de l'extraction faible

Nous souhaitons maintenant inverser la fonction d'extraction faible. Par inverser, nous entendons, étant donnés une spécification et un terme faiblement extrait, retrouver un terme de preuve. Il faut noter que nous ne considérons dans la suite que des termes faiblement extraits sous forme normale η-longue (un terme est en forme η-longue si tous ses sous-termes sont appliqués à autant d'arguments que nécessaire c'est-à-dire pour tout $t : A_1 \to \dots \to A_n \to B$, t apparaît appliqué à n arguments). En fait, nous verrons que nous savons retrouver un terme de preuve partiel avec certains *trous* qui représentent des preuves logiques qui, si elles peuvent être prouvées, permettent la reconstitution totale du terme de preuve. Ces preuves logiques représentent les propriétés que le terme faiblement extrait doit vérifier pour valider sa spécification. Nous avons donc une façon de conserver les preuves sous un aspect plus réduit et nous avons en quelque sorte une représentation canonique des preuves.

Nous allons donc d'abord montrer comment inverser l'extraction faible. Nous verrons que la preuve d'inversion de l'extraction faible donne un algorithme de reconstruction de preuves à partir de programmes. Nous verrons ensuite que, malheureusement, les termes faiblement extraits ne correspondent pas vraiment à des programmes naturels. Les termes de programmes faiblement extraits ne sont que des termes artificiels utiles à la théorie mais non utilisables en pratique. Nous montrerons comment introduire des programmes naturels et utilisables en pratique en perdant une partie du déterminisme. Mais, nous montrerons comment ajouter, quand nécessaire seulement, des annotations dans les programmes naturels et obtenir ainsi une méthode correcte et complète (au sens d'une certaine notion de complétude faible à définir) de reconstruction de preuves.

3.3.1 Déterminisme de l'inversion

Nous allons d'abord donner la définition explicite de notre notion d'inversion. Un programme p sera inversible si, étant donnés un contexte Γ et une spécification S, où S est prouvable et p est cohérent avec S dans Γ, on sait reconstruire de façon déterministe un terme de preuve pour p, avec des *trous* correspondants à des preuves logiques. Si celles-ci sont vérifiées manuellement, alors le terme de preuve a le type S et p est certifié correct. Bien sûr, la preuve recherchée doit suivre la structure imposée par le programme pour garder la correspondance preuve-programme et assurer la correction du programme donné et non d'un autre.

Avant d'énoncer le théorème d'inversion de l'extraction faible, nous allons apporter un certain nombre de restrictions pour pouvoir en faire la preuve. Ces restrictions sont au nombre de deux. Nous allons dans un premier temps donner des raisons intuitives à ces restrictions. Nous insisterons à l'intérieur de la preuve elle-même sur les points précis où ces restrictions interviennent.

1. Nous voulons interdire un certain nombre de situations pour lesquelles nous ne saurons pas passer du terme faiblement extrait au terme de preuve correspondant. Typiquement, prenons un terme T dont le type est de la forme $(x : nat)(P : (y : nat)(x > y) \rightarrow Set)(P\ 1\ h_1) \rightarrow (P\ 2\ h_2)$. La signification de ce terme est que $1 = 2$ et que $h_1 = h_2$. Son extraction faible est $\bar{T} : (x : nat)(P : nat \rightarrow Set)(P\ 1) \rightarrow (P\ 2)$. La signification de ce terme faiblement extrait n'est plus seulement que $1 = 2$. Le lien entre h_1 et h_2 est perdu. Par contre, prenons un terme *racine* de type $(x : nat)(x > 0) \rightarrow nat$. Son extraction faible est $\overline{racine} : nat \rightarrow nat$. Supposons qu'on considère une proposition faiblement extraite $(racine\ x)$ alors on peut reconstruire un terme de preuve $(racine\ x\ h_1)$ où h_1 est une variable existentielle de type $x > 0$.

   En fait, nous souhaitons pouvoir définir des types parlant de preuves logiques (comme celui de *racine*) mais pas des arités parlant de preuves logiques (comme le type de T). Cela signifie que nous considérons un PTS sans les règles $(Prop, Type)$ et $(Prop, TypeSet)$ (voir remarque 1 section 3.1.3).

   Remarques : en se plaçant dans un PTS sans les règles $(Prop, Type)$ et $(Prop, TypeSet)$, on aura toujours les points suivants :

   - $\sigma_i((x : A)B) = (x : \sigma_j(A))\sigma_i(B)$ quelque soit le niveau de B.
   - si $\sigma_0(A) = s$ où s est une sorte alors $A = s$.
   - si $\sigma_i(A) = (x : B)C$ alors $A = (x : B')C'$.
   - si $\sigma_i(A) = [x : B]C$ et si A de niveau 1 alors $A = [x : B']C'$.

2. Par contre, si on sait engendrer dans le cas de *racine* une obligation de preuve h_1 pour $x > 0$, il existe plusieurs possibilités pour h_1. On veut en fait identifier toutes les preuves d'une même proposition logique. Nous avons donc besoin d'une *proof-irrelevance* des preuves logiques. La règle de conversion entre deux termes doit être étendue pour devenir une règle de conversion entre termes typés.

   Définition 42 *Egalité étendue aux termes typés (notée $=_e$) :*
   Si $B =_{\beta\iota} B'$ et si l et l' sont des termes logiques de niveau 2 alors $B[x \leftarrow l] =_e B'[x \leftarrow l']$.

   De façon générale, pour comparer deux termes, on les mettra en forme normale et on comparera ces formes. On aura d'une part $[x_1 : T_1] \ldots [x_n : T_n](c\ u_1 \ldots u_k)$ et d'autre part $[x'_1 : T'_1] \ldots [x'_n : T'_n](c'\ u'_1 \ldots u'_k)$. En effet, si deux termes sont égaux en ce sens faible alors ils ont exactement la même structure d'abstractions et de produits. Pour comparer les deux termes, on compare les T_i récursivement, on compare les têtes c et c', puis chaque u_i. Si u_i est une preuve logique alors u'_i aussi et u_i et u'_i sont considérés comme égaux, sinon on les compare récursivement. c et c' peuvent être des éliminations et alors on compare récursivement à l'intérieur.

Proposition 7 *Proof-irrelevance des preuves logiques :*
Si A et A' sont deux termes de types égaux modulo l'égalité $=_e$, vérifiant les conditions 1 et 2 précédentes et n'étant pas des preuves logiques, alors $\sigma_i(A) = \sigma_i(A') \Rightarrow A = A'$.

Preuve. Nous considérons des termes en forme η-longue. Les formes générales de A et A' sont donc les suivantes :
$A = [x_1 : A_1] \ldots [x_n : A_n](y_1 : B_1) \ldots (y_m : B_m)(c\ t_1 \ldots t_p)$.
$A' = [x'_1 : A'_1] \ldots [x'_n : A'_n](y'_1 : B'_1) \ldots (y'_m : B'_m)(c'\ t'_1 \ldots t'_q)$.

Comme les types de A et A' sont les mêmes modulo l'égalité $=_e$ et comme A et A' sont en forme η-longue, il y a nécessairement autant de x_i que de x'_i et $\sigma(A_i) = \sigma(A'_i)$ donc $A_i = A'_i$ par hypothèse de récurrence.

Comme $\sigma((x : A)B) = (x : \sigma(A))\sigma(B)$ en général et comme A et A' sont de même type modulo l'égalité $=_e$, il y a nécessairement autant de y_i que de y'_i et $\sigma(B_i) = \sigma(B'_i)$ donc $B_i = B'_i$ par hypothèse de récurrence.

Comme $\sigma(A) = \sigma(A')$, on déduit donc maintenant que $\sigma(c\ t_1 \ldots t_p) = \sigma(c'\ t'_1 \ldots t'_q)$. Donc $\sigma(c) = \sigma(c')$.

Trois cas se présentent alors :

- soit c et c' sont élémentaires (variables, constructeurs) et alors nécessairement $c = c'$.
- soit $c = ind$ et $c' = ind'$. Alors, $\sigma(ind) = \sigma(ind')$. Or, σ appliqué à un type inductif se réduit en l'application de σ sur chacune des sous-parties du type inductif. L'hypothèse de récurrence peut être appliquée sur chacune de ces sous-parties et donc $ind = ind'$ et $c = c'$.
- soit $c = \mathsf{Elim}(t, P)\{f_i\}$ et $c' = \mathsf{Elim}(t', P')\{f'_i\}$.
  Alors, $\mathsf{Elim}(\sigma(t), \sigma(P))\{\sigma(f_i)\} = \mathsf{Elim}(\sigma(t'), \sigma(P'))\{\sigma(f'_i)\}$. Or $\sigma(t) : \sigma(ind)$ et $\sigma(t') : \sigma(ind')$. On peut appliquer l'hypothèse de récurrence sur ind et ind' et donc $ind = ind'$. On peut alors appliquer l'hypothèse de récurrence sur t et t', P et P', f_i et f'_i. On obtient alors $t = t'$, $P = P'$ et $f_i = f'_i$. Donc, $c = c'$.

Dans les trois cas, on peut donc déduire que $c = c'$. Il reste donc maintenant à comparer deux termes de types identiques et de σ identiques $(c\ t_1 \ldots t_p)$ et $(c\ t'_1 \ldots t'_q)$. Le type de la tête est donc identique dans les deux cas. Supposons $p \leq q$ (le problème est symétrique). Il y a nécessairement correspondance entre t_i et t'_i pour $i \leq p$. Ensuite, comme les termes sont en forme η-longue, il ne reste pas de produits après l'application de tous les t_i. Donc nécessairement $p = q$. Finalement, $A = A'$. □

Nous pouvons maintenant énoncer le théorème d'inversion de l'extraction faible qui correspond à la notion intuitive suivante : étant donné un programme bien typé cohérent avec une spécification bien typée dans un environnement, on peut trouver un terme de preuve de cette spécification dans un autre environnement (contenant des lemmes logiques) tel que l'extraction faible de ce terme est le programme et l'extraction faible de la spécification est le type du programme. Nous nous plaçons toujours dans un modèle respectant les deux conditions énumérées précédemment et permettant d'assurer la *proof-irrelevance* des preuves logiques. Dans la suite, tous les jugements sont donc restreints à ce modèle.

Notation : on notera $\vdash_r$ les jugements restreints à ce modèle.

Théorème 1 *Inversibilité de l'extraction faible :*

Si $\Gamma \vdash t \in T$ *et si* Γ' *est un environnement bien formé tel que* $\sigma(\Gamma') = \Gamma$ *alors* $\exists t', \mathcal{L}$ *contexte logique bien formé et* S' *tels que* $\Gamma', \mathcal{L} \vdash_r t' : S'$ *avec* $\sigma(t') = t$*,* $\sigma(S') = T$*.*

Preuve. Par récurrence sur le typage de $\in$. Chaque étape décrit la construction du terme de preuve t'. A chaque étape, on connaît Γ' tel que $\sigma(\Gamma') = \Gamma$. Dans toute la preuve, les termes désignés par un type L sont des termes logiques de niveau 2, et $\mathcal{L}$ représente un contexte logique.

L'hypothèse de récurrence sur les jugements de contextes bien formés est : si Γ est bien formé dans $\in$ alors $\exists \Gamma'$ bien formé dans le typage usuel tel que $\sigma(\Gamma') = \Gamma$.

- Contexte :

  $$\frac{}{[]\ bien\ form\acute{e}}$$

  évident

- Contexte :

  $$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ bien\ form\acute{e}} \quad s \in \mathcal{S} = \{Prop, Set, Type, TypeSet\}$$

  Par hypothèse de récurrence, $\exists T', \mathcal{L}$ et S' tels que $\Gamma', \mathcal{L} \vdash_r T' : S'$ avec $\sigma(T') = T$ et $\sigma(S') = s$. Donc, $S' = s$. Donc, $\Gamma', \mathcal{L}[x : T']$ *bien formé.*

- Contexte :

  $$\frac{\Gamma\ bien\ form\acute{e}}{\Gamma \vdash Prop \in Type}$$

  évident

- Contexte :

  $$\frac{\Gamma\ bien\ form\acute{e}}{\Gamma \vdash Set \in TypeSet}$$

  évident

- Variable :

  $$\frac{\Gamma\ bien\ form\acute{e} \quad x : T \in \Gamma}{\Gamma \vdash x \in T}$$

  On prend comme terme de preuve la variable x de type Γ'_x. Comme $\sigma(\Gamma') = \Gamma$, on a $\sigma(\Gamma'_x) = \Gamma_x = T$. Donc, $\Gamma' \vdash_r x : \Gamma'_x$ avec $\sigma(x) = x$ et $\sigma(\Gamma'_x) = T$.

  L'environnement cherché est Γ', le terme de preuve x et son type Γ'_x.

- Formation de l'application :

  $$\frac{\Gamma \vdash t \in (x : A)B \quad \Gamma \vdash u \in A}{\Gamma \vdash (t\ u) \in B[x \leftarrow u]} \quad \text{A de niveau 0 ou informatif de niveau 1}$$

  Par hypothèse de récurrence, $\exists t', \mathcal{L}$ et T' tels que $\Gamma', \mathcal{L} \vdash_r t' : T'$ avec $\sigma(t') = t$ et $\sigma(T') = (x : A)B$. De plus, $\exists u', \mathcal{L}'$ et A' tels que $\Gamma', \mathcal{L}' \vdash_r u' : A'$ avec $\sigma(u') = u$ et $\sigma(A') = A$.

  Nécessairement, $T' = (x : A'')B'$ avec $\sigma(A'') = A$ et $\sigma(B') = B$. Il n'y a pas de produits logiques en tête de T' du fait de la condition 1 qui impose des arités non dépendantes de preuves logiques. Donc, $\Gamma', \mathcal{L} \vdash_r t' : (x : A'')B'$ et $\sigma(A') = \sigma(A'') = A$. Pour pouvoir appliquer la règle de typage standard de l'application, on a besoin de $A' = A''$. On a donc besoin ici de la *proof-irrelevance* des preuves logiques donc de la proposition 7. On déduit donc de $\sigma(A') = \sigma(A'')$ que $A' = A''$. Donc, $\exists \mathcal{L}''$ qui est la réunion de $\mathcal{L}$ et $\mathcal{L}'$ tel que $\Gamma', \mathcal{L}'' \vdash_r (t'\ u') : B'[x \leftarrow u']$. $(t'\ u')$ est donc le terme de preuve recherché avec les conditions voulues à savoir :

$$\begin{array}{rcl} \sigma(t'\ u') & = & (\sigma(t')\ \sigma(u')) \\ & = & (t\ u) \\ \sigma(B'[x \leftarrow u']) & = & \sigma(B')[x \leftarrow \sigma(u')] \\ & = & B[x \leftarrow u] \end{array}$$

– Formation du produit :

$$\frac{\Gamma \vdash T_1 \in s_1 \quad \Gamma[x : T_1] \vdash T_2 \in s_2}{\Gamma \vdash (x : T_1)T_2 \in s_2} \quad (s_1, s_2) \in \mathcal{R} = \mathcal{S} \times \mathcal{S} \setminus \{(Prop, Type), (Prop, TypeSet)\}$$

Par hypothèse de récurrence, $\exists t', \mathcal{L}$ et S' tels que $\Gamma'[x : A], \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t') = T_2$, $\sigma(S') = s_2$ et $\sigma(A) = T_1$. Comme $s_2 \in \mathcal{S}$, $S' = s_2$. Donc, $\Gamma'[x : A], \mathcal{L} \vdash_r t' : s_2$.

Pour pouvoir exhiber un terme de preuve, on doit commuter les variables logiques de $\mathcal{L}$ dans le contexte. Pour cela, on définit un vecteur $\vec{l'} : \vec{L'}$ avec $L'_i = (x : A)L_i$ si $\mathcal{L} = [\vec{l} : \vec{L}]$. On substitue dans t' les l_i par $(l'_i\ x)$ et on obtient un terme t''. On a alors : $\Gamma'[\vec{l'} : \vec{L'}; x : A] \vdash_r t'' : s_2$.

Par application de la règle standard de typage du produit, on a le jugement recherché : $\Gamma'[\vec{l'} : \vec{L'}] \vdash_r (x : A)t'' : s_2$. On a bien les conditions voulues à savoir :

$$\begin{array}{rcl} \sigma((x : A)t'') & = & (x : \sigma(A))\sigma(t'') \\ & = & (x : T_1)T_2 \end{array}$$

$$\sigma(s_2) = s_2$$

– Formation de l'abstraction :

$$\frac{\Gamma \vdash (x : T_1)T_2 \in s \quad \Gamma[x : T_1] \vdash t \in T_2}{\Gamma \vdash [x : T_1]t \in (x : T_1)T_2} \quad s \in \mathcal{S}$$

Par hypothèse de récurrence, $\exists t', \mathcal{L}$ et S' tels que $\Gamma'[x : A], \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t') = t$, $\sigma(S') = T_2$ et $\sigma(A) = T$.

Pour pouvoir exhiber un terme de preuve, on doit commuter les variables logiques $\vec{l}$ dans le contexte. Pour cela, on définit un vecteur $\vec{l'} : \vec{L'}$ avec $L'_i = (x : A)L_i$ si $\mathcal{L} = [\vec{l} : \vec{L}]$. On note $(\vec{l'\ x})$ pour le vecteur $((l'_1\ x), \ldots, (l'_n\ x))$. On a alors : $\Gamma'[\vec{l'} : \vec{L'}; x : A] \vdash_r t'[\vec{l} \leftarrow (\vec{l'\ x})] : S'[\vec{l} \leftarrow (\vec{l'\ x})]$.

Par application de la règle standard de typage de l'abstraction, on a le jugement recherché : $\Gamma'[\vec{l'} : \vec{L'}] \vdash_r [x : A]t'[\vec{l} \leftarrow (\vec{l'\ x})] : (x : A)S'[\vec{l} \leftarrow (\vec{l'\ x})]$. On a bien les conditions voulues à savoir :

$$\begin{array}{rcl} \sigma([x : A]t'[\vec{l} \leftarrow (\vec{l'\ x})]) & = & [x : \sigma(A)]\sigma(t') \\ & = & [x : T_1]t \\ \sigma((x : A)S'[\vec{l} \leftarrow (\vec{l'\ x})]) & = & (x : \sigma(A))\sigma(S'[\vec{l} \leftarrow (\vec{l'\ x})]) \\ & = & (x : T_1)T_2 \end{array}$$

– Règle 3.3 :

$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \rightarrow U}{\Gamma \vdash t \in U}$$

Par hypothèse de récurrence, on a $\exists t', \mathcal{L}$ et S' tels que $\Gamma', \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t') = t$ et $\sigma(S') = A \rightarrow U$. Donc, $S' = (x : A')B'$ avec $\sigma(A') = A$, $A' : Prop$ et $\sigma(B') = U$. Donc, $\Gamma', \mathcal{L} \vdash_r t' : (x : A')B'$.

On est donc exactement dans le même cas que l'application et on sait reconstruire le terme de preuve. On a $\exists \mathcal{L}'$, x' tel que $\Gamma', \mathcal{L}' \vdash_r x' : A'$. Donc, $\exists \mathcal{L}''$ la réunion de $\mathcal{L}$ et de $\mathcal{L}'$ tel que $\Gamma', \mathcal{L}'' \vdash_r (t'\ x') : B'[x \leftarrow x']$ avec $\sigma(t'\ x') = t$ et $\sigma(B'[x \leftarrow x']) = U$. B' peut dépendre de x mais l'extraction faible supprime cette dépendance.

– Règle 3.4 :

$$\frac{\Gamma \vdash t \in U \quad \Gamma \vdash A \in Prop}{\Gamma \vdash t \in A \rightarrow U}$$

Par hypothèse de récurrence, on a $\exists t', \mathcal{L}$ et S' tels que $\Gamma', \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t') = t$ et $\sigma(S') = U$.

Or, $\mathcal{L} = [l_1 : L_1; \ldots; l_i : L_i; x : A'; l_{i+1} : L_{i+1}; \ldots; l_n : L_n]$ avec $\sigma(A') = A$. On fait des commutations à l'intérieur de $\mathcal{L}$, pour obtenir $\mathcal{L}' = [l_1 : L_1; \ldots; l_i : L_i; l'_{i+1} : L'_{i+1}; \ldots; l'_n : L'_n; x : A']$ où $L'_i = (x : A')L_i$.

On a alors $\Gamma', \mathcal{L}' \vdash_r t' : S'$.

Donc, $\Gamma'[l_1 : L_1; \ldots; l_i : L_i; l'_{i+1} : L'_{i+1}; \ldots; l'_n : L'_n] \vdash_r [x : A']t' : A' \rightarrow S'$ avec $\sigma([x : A']t') = \sigma(t') = t$ et $\sigma(A' \rightarrow S') = A \rightarrow U$.

– Formation du type inductif :

$$\frac{\begin{array}{c}(\forall i = 1 \ldots m)\\ \Gamma[p_j : P_j] \vdash A \in s_1 \quad \Gamma[p_j : P_j; X : A] \vdash C_i \in s_2 \quad \mathit{constructeur}(C_i, X)\end{array}}{\Gamma \vdash \mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : A)\{c_1 : C_1 | \ldots | c_m : C_m\} \in (p_1 : P_1) \ldots (p_n : P_n)A} \quad \not\exists P_j : Prop$$

Par hypothèse de récurrence, $\exists \mathcal{L}, t'_i$ et S'_i tels que $\Gamma'[p'_j : P'_j; X' : A'], \mathcal{L} \vdash_r t'_i : S'_i$ avec $\sigma(t'_i) = C_i$, $\sigma(S'_i) = s_2$, $\sigma(P'_j) = P_j$ et $\sigma(A') = A$. De plus, $\exists \mathcal{L}', t''$ et S' tels que $\Gamma'[p'_j : P'_j], \mathcal{L}' \vdash_r t'' : S'$ avec $\sigma(t'') = A$ et $\sigma(S') = s_1$.

On a donc $\Gamma'[p'_j : P'_j; X' : A'; \vec{l} : \vec{L}] \vdash_r t'_i : S'_i$ où $[\vec{l} : \vec{L}]$ est la réunion de $\mathcal{L}$ et de $\mathcal{L}'$. Comme $\sigma(t'') = \sigma(A')$, grâce à la *proof-irrelevance* des preuves logiques, on a $t'' = A'$. Donc, comme on l'a déjà vu dans des cas précédents, on n'a qu'à commuter les $\vec{l}$ et on applique la règle de typage usuelle du type inductif. On obtient bien un terme de preuve qui vérifie les bonnes propriétés.

– Formation du constructeur :

$$\frac{\Gamma \vdash Ind \equiv \mathsf{Ind}[\vec{p} : \vec{P}](X : A)\{c_1 : C_1 | \ldots c_m : C_m\} \in B \quad 1 \leq i \leq m}{\Gamma \vdash \mathsf{Constr}(i, Ind) \in (p_1 : P_1) \ldots (p_n : P_n)C_i[X \leftarrow (Ind\ p_1 \ldots p_n)]}$$

Par hypothèse de récurrence, $\exists ind', \mathcal{L}$ et S' tels que $\Gamma', \mathcal{L} \vdash_r ind' : S'$ avec $\sigma(ind') = Ind$ et $\sigma(S') = B$.

Or, $B = (\vec{p} : \vec{P})A$, donc $\sigma(S') = (\vec{p'} : \vec{P'})A'$ avec $\sigma(p'_i) = p_i$ et $\sigma(A') = A$.

Donc, $\Gamma', \mathcal{L} \vdash_r ind' : (\vec{p'} : \vec{P'})A'$. Donc, $ind' = \mathsf{Ind}[\vec{p} : \vec{P}](X : A')\{c_i' : C_i'\}$. Donc, on a le jugement recherché : $\Gamma', \mathcal{L} \vdash_r \mathsf{Constr}(i, ind') : (\vec{p'} : \vec{P'})C_i'[X \leftarrow (ind'\ \vec{p'})]$ avec $\sigma(\mathsf{Constr}(i, ind')) = \mathsf{Constr}(i, Ind)$ et $\sigma((\vec{p'} : \vec{P'})C_i'[X \leftarrow (ind'\ \vec{p'})]) = (\vec{p} : \vec{P})C_i[X \leftarrow (Ind\ \vec{p})]$.

– Formation de l'élimination :

$$\frac{\Gamma \vdash c \in (I\ \vec{p}\ \vec{a}) \quad \Gamma \vdash Q \in (\vec{x} : \vec{A})(I\ \vec{p}\ \vec{x}) \to s' \quad \overset{(\forall i = 1 \ldots m)}{\Gamma \vdash f_i \in C_i\{(I\ \vec{p}), Q, (\mathsf{Constr}(i, I)\ \vec{p})\}_\sigma}}{\Gamma \vdash \mathsf{Elim}(c, Q)\{f_1 | \ldots | f_m\} \in (Q\ \vec{a}\ c)}$$

Par hypothèse de récurrence, $\exists c_1, \mathcal{L}_1$ et S_c tels que $\Gamma', \mathcal{L}_1 \vdash_r c_1 : S_c$ avec $\sigma(c_1) = c$ et $\sigma(S_c) = (I\ \vec{p}\ \vec{a})$; $\exists Q_1, \mathcal{L}_2$ et S_Q tels que $\Gamma', \mathcal{L}_2 \vdash_r Q_1 : S_Q$ avec $\sigma(Q_1) = Q$ et $\sigma(S_Q) = (\vec{x} : \vec{A})(I\ \vec{p}\ \vec{x}) \to s'$; $\exists f_i', \mathcal{L}^i$ et S_i tels que $\Gamma', \mathcal{L}^i \vdash_r f_i' : S_i$ avec $\sigma(f_i') = f_i$ et $\sigma(S_i) = C_i\{(I\ \vec{p}), Q, (\mathsf{Constr}(i, I)\ \vec{p})\}_\sigma$.

On a donc $S_Q = (\vec{x_1} : \vec{A'})(I_1\ \vec{p_1}\ \vec{x_1}) \to s'$ avec $\sigma(A_i') = A_i$, $\sigma(I_1) = I$, $\sigma(\vec{p_1}) = \vec{p}$ et $\sigma(\vec{x_1}) = \vec{x}$; $S_c = (I_2\ \vec{p_2}\ \vec{a_1})$ avec $\sigma(I_2) = I$, $\sigma(\vec{p_2}) = \vec{p}$ et $\sigma(\vec{a_1}) = \vec{a}$; $S_i = C_i'\{(I_3\ \vec{p_3}), Q_2, (\mathsf{Constr}(i, I_4)\ \vec{p_4})\}_\sigma$ avec $\sigma(C_i') = C_i$, $\sigma(I_3) = I$, $\sigma(\vec{p_3}) = \vec{p}$, $\sigma(Q_2) = Q$, $\sigma(I_4) = I$ et $\sigma(\vec{p_4}) = \vec{p}$.

Donc, par application de la propriété de *proof-irrelevance*, on a $I_1 = I_2 = I_3 = I_4$, $\vec{p_1} = \vec{p_2} = \vec{p_3} = \vec{p_4}$ et $Q_1 = Q_2$.

Donc, $\exists \mathcal{L}_3$ réunion de $\mathcal{L}_1$, $\mathcal{L}_2$ et $\mathcal{L}^i$ tel que $\Gamma', \mathcal{L}_3 \vdash_r \mathsf{Elim}(c_1, Q_1)\{f_i'\} : (Q_1\ \vec{a_1}\ c_1)$ avec les bonnes propriétés.

$\square$

Nous avons donc une preuve de l'inversibilité de la fonction d'extraction faible. Cette preuve est constructive car, pour chaque cas, elle donne la méthode explicite de construction du terme de preuve. Cependant, il faut bien noter que nous ne construisons en réalité qu'un terme de preuve partiel. En effet, pratiquement, nous ne savons pas retrouver les preuves logiques (dans la preuve du théorème, nous les avons toutes identifiées). Donc, la méthode reconstruit un terme de preuve partiel avec des *trous* correspondant à des obligations de preuve.

De plus la preuve est faite par récurrence sur le typage $\in$. Or, à cause des règles 3.3 et 3.4, ce typage n'est pas déterministe. Dans la suite, nous allons donc définir un autre typage $\in_2$ équivalent à $\in$ mais déterministe. Nous montrerons que nous savons passer de $\in$ à $\in_2$ et réciproquement. La preuve précédente est alors toujours valide dans le système de typage $\in_2$. Ainsi, si nous considérons des programmes dans le système de typage $\in_2$, nous avons une méthode de reconstruction de preuve déterministe.

Puisque l'indéterminisme de $\in$ provient des règles 3.3 et 3.4, nous allons créer un nouveau système de typage en supprimant ces deux règles et en les remplaçant par une seule et unique règle.

Nous définissons donc $\in_2$ en remplaçant 3.3 et 3.4 par 3.5. Pour cela, nous définissons une notion tout d'abord d'équivalence logique entre deux termes puis le système $\in_2$.

Définition 43 *Equivalence logique :*

Deux types A et B sont logiquement équivalents (noté $A \equiv_L B$) si et seulement si il existe $\vec{L}$ et $\vec{M}$ logiques de niveau 1, et C tels que $A = \vec{L} \to C$ et $B = \vec{M} \to C$.

Remarques :

1. Si $A \equiv_L B$ et $\Gamma \vdash M \in A$, alors $\Gamma \vdash M \in B$. Ce qui est trivial par une série d'applications des règles 3.3 et 3.4.

2. $\equiv_L$ est réflexive, symétrique et transitive.
3. $L \rightarrow A \equiv_L A$ si L est logique de niveau 1.

Nous donnons maintenant la définition du jugement de typage $\in_2$. Nous donnons uniquement la règle 3.5 qui remplace les règles 3.2, 3.3 et 3.4 car les autres sont identiques à celles de $\in$.

Définition 44

$$\frac{\Gamma \vdash t \in_2 A \quad \Gamma \vdash u \in_2 B \quad A \equiv_L (x:M)N \quad B \equiv_L M}{\Gamma \vdash (t\ u) \in_2 N[x \leftarrow u]} \tag{3.5}$$

Nous pouvons maintenant montrer l'équivalence de $\in$ et de $\in_2$. Pour cela, il faut uniquement montrer qu'on peut déduire 3.5 de 3.3 et 3.4 et inversement. En fait, on peut montrer que le type d'un terme dans les deux systèmes est presque le même, c'est-à-dire le même à des produits logiques près.

Proposition 8 *Equivalence de $\in$ et de $\in_2$:*
$\Gamma \vdash t \in S$ si et seulement si $\exists S', \Gamma \vdash t \in_2 S'$ avec $S \equiv_L S'$.

Preuve.

– Condition nécessaire : $\in \Rightarrow \in_2$.

Par récurrence sur le typage $\in$. Les règles qui posent problème sont les règles 3.3 et 3.4.

– Règle 3.3 :

$$\frac{\Gamma \vdash A \in Prop \quad \Gamma \vdash t \in A \rightarrow U}{\Gamma \vdash t \in U}$$

Par hypothèse de récurrence, $\exists T$ tel que $\Gamma \vdash t \in_2 T$ et $T \equiv_L A \rightarrow U$. La remarque 3 sur $\equiv_L$ donne $A \rightarrow U \equiv_L U$. La transitivité de $\equiv_L$ donne $T \equiv_L U$. On a exactement ce qu'on cherche, à savoir le même terme que dans $\in$ avec un type équivalent à des produits logiques près.

– Règle 3.4 :

$$\frac{\Gamma \vdash t \in U \quad \Gamma \vdash A \in Prop}{\Gamma \vdash t \in A \rightarrow U}$$

Par hypothèse de récurrence, $\exists T$ tel que $\Gamma \vdash t \in_2 T$ et $T \equiv_L U$. La remarque 3 sur $\equiv_L$ donne $A \rightarrow U \equiv_L U$. La transitivité de $\equiv_L$ donne $T \equiv_L A \rightarrow U$. On a exactement ce qu'on cherche, à savoir le même terme que dans $\in$ avec un type équivalent à des produits logiques près.

– Condition suffisante : $\in_2 \Rightarrow \in$.

Par récurrence sur le typage $\in_2$. La règle qui pose problème est la règle 3.5.

$$\frac{\Gamma \vdash t \in_2 A \quad \Gamma \vdash u \in_2 B \quad A \equiv_L (x:M)N \quad B \equiv_L M}{\Gamma \vdash (t\ u) \in_2 N[x \leftarrow u]}$$

Par hypothèse de récurrence, on a $\exists A_1, B_1$ tels que $\Gamma \vdash t \in A_1$, $\Gamma \vdash u \in B_1$, $A_1 \equiv_L A$ et $B_1 \equiv_L B$. Par transitivité de $\equiv_L$, $A_1 \equiv_L (x:M)N$ et $B_1 \equiv_L M$. Finalement, par la première remarque sur $\equiv_L$, $\Gamma \vdash (t\ u) \in N[x \leftarrow u]$.

□

Nous pouvons en déduire qu'étant donnés une spécification S et un contexte satisfaisant les restrictions 1 et 2, si p est un programme typable dans $\in$ et compatible avec S alors le type de p peut être calculé avec $\in_2$ de façon déterministe et ce type est compatible avec S. On peut alors grâce au théorème 1 retrouver une preuve de S par une méthode déterministe.

Il faut cependant noter une difficulté, à savoir que la règle de conversion rend toujours $\in_2$ non déterministe. Mais, en fait, dans le système que nous considérons, cette règle est systématiquement repoussée dans l'application et dans l'élimination à la fin. Le typage dans $\in_2$ est alors déterministe.

3.3.2 Résultats de complétude sur l'inversion

L'inversion de l'extraction faible peut maintenant se faire de façon déterministe. Si on part d'une preuve $t : S$ et qu'on applique la méthode de reconstruction de preuve à $\sigma(t)$, par le théorème 1, nous obtenons $\Gamma', \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t) = \sigma(t')$, $\sigma(S) = \sigma(S')$ et $\mathcal{L}$ un contexte logique bien formé. Le terme de preuve engendré est partiel. Il contient un certain nombre de *trous* qui correspondent aux preuves logiques qui ne peuvent être retrouvées automatiquement. Le contexte logique contient des variables existentielles correspondant à ces lemmes logiques.

Ce théorème entraîne certains résultats sur les possibilités offertes par la méthode de synthèse de preuves sous-jacente. Nous pouvons considérer plusieurs aspects. Tout d'abord, que se passe-t-il si l'on applique la méthode d'inversion sur un terme qui est l'extraction faible d'une preuve (c'est-à-dire qui est exactement la trace d'une preuve) et sur sa spécification correspondante ? Nous pouvons montrer que les lemmes logiques engendrés dans une telle situation sont prouvables. Nous avons alors une notion de complétude faible pour cette méthode. La méthode d'inversion définie par la preuve du théorème 1 peut être vue comme un foncteur F qui, à partir d'une dérivation de $\Gamma \vdash t \in S$ construit Γ', t', S' tels que $\Gamma' \vdash_r t' : S'$.

Notation : soit φ une dérivation $\Gamma \vdash_r t : S$, on note $\sigma(\varphi)$ la dérivation correspondante dans σ, à savoir $\sigma(\Gamma) \vdash \sigma(t) \in \sigma(S)$. On se place dans $\in$ et non pas dans $\in_2$, car il contient les règles utilisées pour la démonstration du théorème 1 et la preuve de complétude suit cette preuve. De plus, on a montré l'équivalence des deux jugements de typage.

Proposition 9 *Complétude faible de l'inversion de l'extraction faible :*
Si $\varphi \equiv \Gamma \vdash_r t : S$ alors $F(\sigma(\varphi)) \equiv \Gamma', \mathcal{L} \vdash_r t' : S'$ avec t et t', S et S' identiques modulo l'identification des preuves logiques d'une même proposition, Γ' prouvable dans Γ et avec $\mathcal{L}$ contenant des lemmes logiques dont il existe une preuve dans Γ.

Preuve. On fait une récurrence sur le typage de t. On passe sur les règles de formation de contexte.

- Le cas variable est évident.
- Produit :

$$\frac{\Gamma \vdash_r A : s_1 \quad \Gamma[x : A] \vdash_r B : s_2}{\Gamma \vdash_r (x : A)B : s_2}$$

On a alors :

$$\frac{\sigma(\Gamma) \vdash \sigma(A) \in s_1 \quad \sigma(\Gamma)[x : \sigma(A)] \vdash \sigma(B) \in s_2}{\sigma(\Gamma) \vdash (x : \sigma(A))\sigma(B) \in s_2}$$

Par hypothèse de récurrence, $\exists \Gamma', t', S', \mathcal{L}$ tels que $\Gamma'[x : A'], \mathcal{L} \vdash_r t' : S'$ avec $\sigma(\Gamma') = \sigma(\Gamma)$, $\sigma(A') = \sigma(A)$, $\sigma(t') = \sigma(B)$, $\sigma(S') = s_2$ et $\mathcal{L} = [\vec{l} : \vec{L}]$.

F construit alors le jugement $\Gamma'[\vec{l'} : \vec{L'}] \vdash_r (x : A')t'' : s_2$ où $L'_i = (x : A')L_i$ et $t'' = t'[l_i \leftarrow (l'_i\ x)]$. Par hypothèse de récurrence, les L_i sont prouvables dans Γ'. La seule transformation qui est faite est la commutation des L_i en L'_i et cette commutation ne change pas la prouvabilité. Donc, les L'_i sont prouvables dans Γ.

– Abstraction :

$$\frac{\Gamma \vdash_r (x : A)B : s \quad \Gamma[x : A] \vdash_r t : B}{\Gamma \vdash_r [x : A]t : (x : A)B}$$

On doit distinguer deux cas : soit $A : Prop$, soit A n'est pas de type $Prop$.

- A n'est pas de type $Prop$. On a exactement le même comportement que pour le produit car on retombe sur la règle d'abstraction au niveau de $\in$.
- $A : Prop$. On a alors :

$$\frac{\sigma(\Gamma)[x : \sigma(A)] \vdash \sigma(t) \in \sigma(B)}{\sigma(\Gamma)[x : \sigma(A)] \vdash \sigma(t) \in \sigma(A) \rightarrow \sigma(B)}$$

  Par hypothèse de récurrence, $\exists \Gamma', t', S', \mathcal{L}$ tels que $\Gamma'[x : A'], \mathcal{L} \vdash_r t' : S'$ avec $\sigma(\Gamma') = \sigma(\Gamma)$, $\sigma(A') = \sigma(A)$, $\sigma(t') = \sigma(t)$, $\sigma(S') = \sigma(B)$ et $\mathcal{L} = [\vec{l} : \vec{L}]$.
  F construit alors le jugement $\Gamma', \mathcal{L}' \vdash_r [x : A']t' : A' \rightarrow S'$ où $\mathcal{L}'$ est un contexte logique résultant de commutations à l'intérieur de $\mathcal{L}$. Par hypothèse de récurrence, les L_i sont prouvables. La seule transformation qui est faite est la commutation à l'intérieur de $\mathcal{L}$ et cette commutation ne change pas la prouvabilité. Donc, $\mathcal{L}'$ est un contexte logique prouvable.

– Application :

$$\frac{\Gamma \vdash_r t : (x : A)B \quad \Gamma \vdash_r u : A}{\Gamma \vdash_r (t\ u) : B[x \leftarrow u]}$$

On doit distinguer deux cas : soit $A : Prop$, soit A n'est pas de type $Prop$.

- A n'est pas de type $Prop$.
  On a alors :

$$\frac{\sigma(\Gamma) \vdash \sigma(t) : (x : \sigma(A))\sigma(B) \quad \sigma(\Gamma) \vdash \sigma(u) : \sigma(A)}{\sigma(\Gamma) \vdash (\sigma(t)\ \sigma(u)) : \sigma(B)[x \leftarrow \sigma(u)]}$$

  Par hypothèse de récurrence, $\exists \Gamma', t', \mathcal{L}$ et S' tels que $\Gamma', \mathcal{L} \vdash_r t' : S'$ avec $\sigma(t') = t$ et $\sigma(S') = (x : \sigma(A))\sigma(B)$. De plus, $\exists \Gamma'', u', \mathcal{L}'$ et A' tels que $\Gamma'', \mathcal{L}' \vdash_r u' : A'$ avec $\sigma(u') = u$ et $\sigma(A') = A$. $S' = (x : A'')B'$ avec $\sigma(A'') = \sigma(A)$ et $\sigma(B') = \sigma(B)$.
  F construit alors le jugement $\Gamma''', \mathcal{L}'' \vdash_r (t'\ u') : B'[x \leftarrow u']$ où Γ''' est la réunion de Γ et Γ', et $\mathcal{L}''$ est la réunion de $\mathcal{L}$ et de $\mathcal{L}'$. Par hypothèse de récurrence, $\mathcal{L}$ et $\mathcal{L}'$ sont prouvables donc $\mathcal{L}''$ aussi.
- $A : Prop$. On a alors :

$$\frac{\sigma(\Gamma) \vdash \sigma(t) \in (x : \sigma(A))\sigma(B)}{\sigma(\Gamma) \vdash \sigma(t) \in \sigma(B)}$$

  Par hypothèse de récurrence, $\exists \Gamma', t', S', \mathcal{L}$ tels que $\Gamma', \mathcal{L} \vdash_r t' : S'$ avec $\sigma(\Gamma') = \sigma(\Gamma)$, $\sigma(t') = \sigma(t)$, $\sigma(S') = (x : \sigma(A))\sigma(B)$ et $\mathcal{L} = [\vec{l} : \vec{L}]$.
  F construit alors le jugement $\Gamma', \mathcal{L}' \vdash_r (t'\ x') : B'[x \leftarrow x']$ où $S' = (x : A')B'$, $x' : A'$ est une variable logique avec $\sigma(A') = \sigma(A)$ et $\mathcal{L}'$ est un contexte logique résultant de

la réunion de $\mathcal{L}$ et du contexte dans lequel x' est bien typé. Or, $\sigma(A) = \sigma(A')$ entraîne $A = A'$. Donc, $x' : A$. Or, on sait qu'il existe une preuve de A qui est u. Donc, la variable logique x' peut bien être instanciée. Le lemme logique créé à cette étape est prouvable. Par hypothèse de récurrence, $\mathcal{L}$ est prouvable. Donc, $\mathcal{L}'$ est un contexte logique prouvable.

- Les cas type inductif, constructeur et élimination sont sur le même modèle.

□

Ensuite, on peut se demander ce qui se passe si l'on applique la méthode à une spécification prouvable et à un terme quelconque cohérent avec cette spécification (quelconque au sens où il n'est pas nécessairement la trace d'une preuve). Naturellement, si le programme est faux, les lemmes logiques engendrés ne sont pas prouvables. Mais, on ne peut pas assurer que, si le programme est correct, alors les lemmes engendrés sont prouvables. Nous montrons sur des exemples que des programmes cohérents avec une spécification prouvable peuvent engendrer des lemmes logiques non prouvables. En bref, si le programme n'est pas une trace de preuve, on ne peut rien garantir sur la prouvabilité des lemmes logiques engendrés.

Exemples :

- Si le programme n'est pas une trace de preuve, on peut alors avoir au niveau de la preuve des comportements de la forme suivante, où $A : Prop$:

$$\cfrac{\cfrac{u : S}{[x : A]u : A \rightarrow S}}{([x : A]u\ x) : S}$$

Ce qui donne dans σ :

$$\cfrac{\cfrac{\sigma(u) \in \sigma(S)}{\sigma(u) \in A \rightarrow \sigma(S)}}{\sigma(u) \in \sigma(S)}$$

Au passage à la deuxième dérivation, rien ne garantit alors que le lemme logique engendré soit prouvable. Le problème vient en fait des règles 3.3 et 3.4.

- Prenons la spécification $(x : nat)(x > 0) \rightarrow \{y : nat | y < x\}$ avec une pseudo-syntaxe *Coq*. Un programme annoté cohérent avecs cette spécification peut être :

```
[x:nat]<{y:nat|y<x}>Match x with
                     0
                     [n:nat]n
          end
```

Un lemme logique improuvable est alors engendré :$(0 < 0)$.

Un programme qui engendre uniquement des lemmes prouvables serait :

```
[x:nat]<(x>0)->{y:nat|y<x}>Match x with
                     0
                     [n:nat]n
          end
```

Nous avons donc une méthode de reconstruction déterministe d'un terme de preuve à partir d'un programme, valide par rapport à une spécification, avec des *trous* correspondant aux preuves logiques qui ne peuvent être retrouvées. Ces preuves doivent être faites manuellement. Elles symbolisent les propriétés logiques que le programme doit vérifier pour valider sa spécification.

Nous pouvons faire une remarque intéressante. Si l'on considère un point de vue d'implantation, on peut estimer que nous avons plus ou moins une représentation canonique des preuves. En effet, au lieu de conserver les preuves sous forme de termes du Calcul des Constructions Inductives, nous pouvons imaginer de les sauvegarder sous forme d'un terme faiblement extrait augmenté d'une liste de termes de preuve correspondant aux propriétés logiques.

3.3.3 Programmes plus réalistes : une méthode heuristique

Les programmes faiblement extraits ne sont pas des programmes très naturels. Nous aimerions pouvoir écrire des programmes contenant moins de spécifications et retrouver ces spécifications en utilisant l'unification. Cela signifie que nous aimerions que le programmeur puisse écrire des programmes sous la même forme que les programmes extraits de ${F_\omega}^{Ind}$, tout en gardant les propriétés qui viennent d'être vues sur les termes faiblement extraits. C'est le but d'une tactique qui a été implémentée dans le système Coq, la tactique Program (voir annexe A). Nous décrirons dans cette partie, non pas l'implantation d'une telle tactique, mais une méthode heuristique et donc non complète qui permet de retrouver des preuves à partir de programmes extraits de ${F_\omega}^{Ind}$, en s'inspirant fortement de ce qui a été fait sur l'extraction faible.

Le fait de vouloir utiliser l'unification rend le problème non déterministe. Cependant, dans les cas pratiques, nous pouvons utiliser des heuristiques qui pallient à ce problème. Nous allons décrire ces heuristiques sur des exemples. Naturellement, comme cette méthode est heuristique, elle peut échouer par manque d'informations. La section 3.3.4 remédie à ce problème et permet de rendre la méthode valide et complète (au sens d'une certaine notion de complétude faible à préciser).

Pour distinguer les cas dans lesquels la méthode nécessite des heuristiques, nous pouvons regarder les cas dans lesquels la méthode déterministe sur les termes faiblement extraits nécessite réellement la présence de spécifications. Nous pouvons distinguer deux cas : l'application et l'élimination. En fait, nous en distinguons aussi un troisième qui est celui d'une application dont la tête est une élimination.

Nous allons détailler un exemple de chacun de ces trois cas pour illustrer le type d'heuristiques utilisées. Il s'agit en fait uniquement de suivre la procédure déterministe vue pour les programmes faiblement extraits mais en prenant des décisions pour rendre l'unification déterministe (i.e. à une seule solution).

On continue à demander une certaine notion de correction aux programmes qui assure leur compatibilité par rapport à l'extraction de [PM89a]. Pour cela, nous considérons un système de typage noté $\in$ un peu différent de celui présenté précédemment. Les règles d'oubli de l'information logique dans les types sont remplacées par une règle de conversion plus forte (voir 3.6). Nous donnons uniquement la règle 3.6. Les autres règles ne changent pas par rapport à $\in$.

Définition 45

$$\frac{\Gamma \vdash T \in s \quad \Gamma \vdash T' \in s' \quad \Gamma \vdash t \in T \quad \varepsilon_i(T) =_{\beta\iota} \varepsilon_i(T')}{\Gamma \vdash t \in T'} \quad \textit{pour } i = 0,1 \tag{3.6}$$

Cas de l'application

Supposons qu'on veuille prouver $\forall n.\exists m.(S\ m) = n \vee n = m = 0$. Un programme extrait satisfaisant cette spécification est :

```
[n:nat](nat_rec (sig nat) (exist nat 0) [y:nat][H:(sig nat)](exist nat y))
```

où `nat_rec` est le principe de récurrence standard sur les entiers et dont le type est $(P : nat \rightarrow Set)(P\ 0) \rightarrow ((n : nat)(P\ n) \rightarrow (P\ (S\ n))) \rightarrow (n : nat)(P\ n)$. Attention, dans le programme ci-dessus `nat_rec` n'est pas vu comme une variable de preuve mais comme une variable de programme (donc extraite) et son type est donc $(P : Set)P \rightarrow (nat \rightarrow P \rightarrow P) \rightarrow nat \rightarrow P$.

Cet exemple est en l'occurrence un exemple où l'application code une élimination. Cependant, nous avons choisi cet exemple car il illustre le problème de retrouver des spécifications par unification et illustre bien le cas général.

Le terme de preuve que l'on souhaiterait retrouver est :

```
[n:nat](nat_rec [n0:nat]{m:nat|(S m)=n0 \/ n0=m=0}
                (exist nat [m:nat]((S m)=0 \/ 0=m=0) 0 P1)
                [y:nat][H:{m:nat|(S m)=y \/ y=m=0}]
                  (exist nat [m:nat]((S m)=(S y) \/ (S y)=m=0) y P2))
```

où `P1` et `P2` sont des preuves de $(S\ 0) = 0 \vee 0 = 0 = 0$ et $(S\ y) = (S\ y) \vee (S\ y) = y = 0$.

Considérons uniquement le cas de l'application (l'abstraction de début est triviale). Nous cherchons le prédicat dont l'extraction est *nat*. Or, nous connaissons `nat_rec` et son type de preuve. Nous savons donc que le prédicat recherché P est de type $nat \rightarrow Set$. Pour instancier P, nous utilisons la spécification. Nous unifions la tête du type de `nat_rec` (à savoir $(P\ n)$) avec la spécification (à savoir $\exists m.(S\ m) = n \vee n = m = 0$).

L'indéterminisme de la situation apparaît car plusieurs unificateurs sont possibles :

1. $P = [n0]\exists m.(S\ m) = n0 \vee n0 = m = 0$.
2. $P = [n0]\exists m.(S\ m) = n0 \vee n = m = 0$.
3. $P = [n0]\exists m.(S\ m) = n \vee n0 = m = 0$.
4. $P = [n0]\exists m.(S\ m) = n \vee n = m = 0$.

Une heuristique est donc nécessaire pour choisir un unificateur. L'heuristique utilisée est de garder l'unificateur le plus *liant*, à savoir le premier dans le cas de notre exemple. Dans une grande majorité des cas pratiques, cette unification fonctionne bien. Mais, si le prédicat choisi n'est pas le bon, la méthode a toutes les chances d'échouer.

Nous engendrons ainsi deux sous-buts :

$$\begin{cases} \exists m.(S\ m) = 0 \vee 0 = m = 0 \\ \forall n.\exists m.((S\ m) = n \vee n = m = 0) \rightarrow \exists m.(S\ m) = (S\ n) \vee (S\ n) = m = 0 \end{cases}$$

associés aux deux sous-programmes `(exist nat 0)` et
`[y:nat][H:{m:nat|(S m)=y\/y=m=0}](exist nat [m:nat]((S m)=(S y)\/(S y)=m=0) y)`.
La méthode peut ainsi être itérée et le terme de preuve complet sauf les parties `P1` et `P2` peut être retrouvé. Donc, on obtient deux lemmes logiques à prouver correspondant à `P1` et `P2` :

$$\begin{cases} (S\ 0) = 0 \vee 0 = 0 = 0 \\ (S\ y) = (S\ y) \vee (S\ y) = y = 0 \end{cases}$$

Nous voyons sur cet exemple que l'heuristique principale du cas applicatif consiste à retrouver les bons prédicats qui sont en arguments de l'application. Ceci a une autre conséquence que l'exemple précédent. En effet, nous pouvons supposer que la spécification du terme applicatif est un peu plus compliquée que celle de l'exemple précédent au sens où elle peut contenir en tête un certain nombre de produits logiques, c'est-à-dire elle peut avoir la forme $(\vec{l} : \vec{L})S$. Le problème est alors de décider à quoi correspondent ces produits logiques. Soit ils font partie intégrante de la spécification de l'application (c'est-à-dire le type de preuve de la tête de l'application commence par des produits logiques) soit ce sont des hypothèses logiques à rajouter dans le contexte (c'est-à-dire le terme de preuve correspondant à l'application commence par des λ-abstractions logiques).

L'heuristique choisie est d'introduire comme hypothèses logiques toutes celles qui ont peu de chance d'être modifiées par la suite. On distingue alors deux cas. Soit la tête du type de l'application est liée soit non. Dans le premier cas, on a généralement un prédicat. Aucune décision n'est plus valable qu'une autre. L'heuristique choisie qui semblait le mieux fonctionner en pratique est d'introduire les hypothèses dont la tête du type de l'application ne dépend pas. Ce cas correspond exactement à la même situation que pour l'élimination (nous en montrerons un exemple dans le paragraphe suivant). Dans le second cas, on introduit toutes les hypothèses logiques.

Cas de l'élimination

Nous prenons pour exemple une étape de la preuve d'un algorithme de division euclidienne. La spécification est $(b > r) \to (n = b * q + r) \to \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r)$. Un programme possible correspondant à cette spécification peut être :

```
<nat*nat>Match (inf b (S r)) with
           ((S q),0)
           (q,(S r))
end
```

Le principe de recherche du prédicat d'élimination est le même que dans le cas précédent. Le terme sur lequel on fait une élimination est un booléen donc de type *bool*. On cherche à appliquer l'élimination sur les booléens de type $(P : bool \to Set)(P\ true) \to (P\ false) \to (b : bool)(P\ b)$.

Mais, la spécification ne dépend pas du terme $(inf\ b\ (S\ r))$ sur lequel on veut faire l'élimination. On n'a donc qu'un seul unificateur entre $(P\ b)$ et la spécification qui est :

$$P = [c : bool](b > r) \to (n = b * q + r) \to \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r)$$

qui est sans aucun intérêt.

Ceci signifie que, dans des cas comme celui-ci, où il n'y a pas de dépendance de la spécification par rapport au terme sur lequel on fait la récurrence, il faut introduire artificiellement une dépendance. Une solution peu coûteuse a été choisie. Si Sp est la spécification, t le terme sur lequel on fait la récurrence et T_t son type, on transforme Sp en $(t = t) \to Sp$ en marquant qu'on ne veut éliminer que la seconde occurrence de t. L'unificateur choisi est alors $P = [x : T_t](t = x) \to Sp$. Cette heuristique n'est appliquée que dans le cas d'un type non récursif. En effet, dans le cas d'un type récursif, cela peut être dangereux au sens où l'on risque d'engendrer des choses non prouvables. Prenons un exemple avec $T_t = nat$. L'unificateur est alors $[x : nat](t = x) \to Sp$. L'élimination va engendrer deux sous-buts : $(t = 0) \to Sp$ et $[y : nat]((t = y) \to Sp) \to (t = (S\ y)) \to Sp$. Et rien n'assure que le second sous-but soit prouvable.

Dans notre cas, nous obtenons :

$$P = [c : bool]((b < (S\ r)) = c) \to (b > r) \to (n = b * q + r) \to \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r)$$

Nous engendrons ainsi deux sous-buts :

$$\left\{\begin{array}{l}((inf\ b\ (S\ r)) = true) \rightarrow (b > r) \rightarrow (n = b * q + r) \rightarrow \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r) \\ ((inf\ b\ (S\ r)) = false) \rightarrow (b > r) \rightarrow (n = b * q + r) \rightarrow \exists q.\exists r.((S\ n) = b * q + r) \wedge (b > r)\end{array}\right.$$

associés aux deux sous-programmes `((S q),0)` et `(q,(S r))`.

Dans le cas de l'élimination, nous avons donc une méthode heuristique pour déterminer le prédicat d'élimination. De plus, nous avons exactement le même problème que pour l'application, à savoir les produits logiques en tête de la spécification de l'élimination. En effet, là-aussi, il faut décider si ces produits font partie intégrante du prédicat d'élimination ou pas. Nous prenons le même genre de décision que pour l'application. A savoir, nous introduisons toutes les hypothèses logiques qui ne risquent pas d'être modifiées par l'élimination. Il s'agit donc des produits logiques qui ne dépendent pas du terme sur lequel porte l'élimination. Encore une fois, cette décision n'est motivée que par la pratique et n'a aucune justification théorique.

Exemple : prenons l'exemple d'une spécification de la forme : $(n, m : nat)(le\ n\ m) \rightarrow \{(lt\ n\ m)\} + \{(n = m)\}$ à savoir pour deux entiers n et m, si $n \leq m$ alors soit $n < m$ soit $n = m$. Un programme possible pour une telle spécification est :

```
[n:nat]<nat->bool>Match n with
            [m:nat]<bool>Match m with
                           true
                           [_:nat][_:bool]false
                    end
            [_:nat][H:nat->bool][m:nat]
                <bool>Match m with
                      (except bool)
                      [n1:nat][_:bool](H n1)
                end
       end
```

La spécification correspondant à la première élimination sur m est : $(le\ 0\ m) \rightarrow \{(lt\ 0\ m)\} + \{(0 = m)\}$. Le problème est de savoir si le produit logique $(le\ 0\ m)$ doit être d'abord introduit dans le contexte et l'élimination appliquée ensuite sur la nouvelle spécification, ou si l'élimination doit être appliquée directement sur cette spécification. Dans ce cas précis, l'élimination est appliquée directement sur la spécification telle quelle. En effet, on fait une élimination sur m et le produit dépend de m. Il vaut donc mieux appliquer l'élimination aussi sur le produit logique qui dépend de m. On retrouve le même genre de problème pour l'autre élimination sur m.

Cas d'une élimination en tête d'une application

Il reste un cas particulier de l'application à traiter : le cas d'une élimination en tête d'une application. Pour ce cas, nous ne donnerons pas d'exemple car il est difficile de trouver un exemple simple de ce cas précis. Cependant, nous pouvons assez facilement expliquer le problème.

Nous avons vu dans le cas précédent que nous pouvions retrouver le prédicat d'élimination à partir de la spécification. Mais, lorsque l'élimination est en tête d'une application, nous perdons cette possibilité. Aussi, nous aimerions nous ramener au cas précédent.

Pour cela, nous utilisons la généralisation. Ceci correspond à l'application de la règle d'inférence suivante :

$$\frac{\Gamma \vdash \mathsf{Elim}(c, P)\{f_1|\ldots|f_n\} : (x : A)B \quad \Gamma \vdash a : A}{\Gamma \vdash (\mathsf{Elim}(c, P)\{f_1|\ldots|f_n\}\ a) : B[x \leftarrow a]}$$

C'est-à-dire, étant donnés le programme $(\mathsf{Elim}(c, P)\{f_1|\ldots|f_n\}\ a)$ et la spécification $B[x \leftarrow a]$, nous généralisons la spécification par rapport à l'argument a et nous lui associons comme programme l'élimination seule. On est donc ainsi ramené au cas précédent. Cette technique est donc un artifice (non déterministe, puisqu'il y a plusieurs façons de généraliser, exactement comme il peut y avoir plusieurs unificateurs dans le cas de l'application), pour se ramener à un cas connu.

3.3.4 Annotations de programmes: une méthode correcte et complète

La méthode décrite précédemment n'est donc qu'heuristique. Elle joue sur la possibilité de tomber dans des cas où l'unification d'ordre supérieur fonctionne et où on retrouve le bon unificateur (ce qui semble dans la pratique être assez courant). Ceci dit, pour assurer le déterminisme de cette méthode, il faut permettre au programmeur d'ajouter dans le programme des informations logiques que la méthode puisse utiliser pour être dirigée. Nous introduisons donc une notion d'annotations de programmes et nous montrons que, dans le cadre des programmes annotés, nous avons une méthode valide et complète.

Présentation des annotations

Pour introduire des annotations, nous ajoutons une nouvelle syntaxe.

Syntaxe : S est une annotation pour le programme p se note $p :: S$.

Les annotations sont alors définies par la règle d'inférence qui permet leur introduction.

Définition 46 *Annotations :*

$$\frac{\Gamma \vdash p \in S \quad \varepsilon_i(S) =_{\beta\iota} \varepsilon_i(S')}{\Gamma \vdash (p :: S) \in S'} \quad \textit{pour } i = 0, 1$$

La signification de cette définition est la suivante. S est une annotation pour p si p est cohérent avec S. De plus, si p est aussi cohérent avec S', alors $p :: S$ est cohérent avec S' et $\varepsilon(S) = \varepsilon(S')$. Les annotations sont donc des spécifications logiques qui marquent des programmes et permettent de préciser leur spécification. Le but est d'utiliser ces annotations pour indiquer explicitement la spécification d'un programme quand la méthode heuristique échoue ou risque d'échouer.

Il faut noter qu'une annotation peut contenir des variables libres. Ces variables libres peuvent être des variables de programmes mais aussi des variables logiques. Ceci signifie qu'il faut pouvoir faire référence à des variables logiques à l'intérieur même des programmes et donc autoriser les abstractions sur des variables logiques. Cela correspond à la règle déjà existante du jugement $\in$ sur les abstractions logiques, à savoir :

$$\frac{\Gamma[x : L] \vdash p \in S}{\Gamma \vdash [x : L]p \in (x : L)S} \quad \text{si } L : Type$$

Ces abstractions sur des variables logiques peuvent donc être utilisées pour faire référence à des variables logiques utilisées dans des annotations. Mais, une utilisation ad-hoc de ces abstractions est apparu lors de l'implantation. Elles peuvent permettre de guider la méthode heuristique.

Comme nous l'avons vu précédemment, un problème de la méthode heuristique est de décider si des introductions logiques doivent être faites ou non. Typiquement, dans certains cas, certaines introductions ne sont pas faites alors qu'il serait nécessaire qu'elles le soient. Ajouter des abstractions logiques dans un programme peut alors permettre de forcer la méthode à introduire une hypothèse logique à un moment donné.

Exemple 1 :

Nous reprenons l'exemple de la division euclidienne. La spécification est $(b > r) \rightarrow (n = b * q + r) \rightarrow \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$ et le programme :

```
<nat*nat>Match (inf b (S r)) with
            ((S q),O)
            (q,(S r))
end
```

Dans ce programme, on sous-entend qu'on connaît la spécification du terme `inf`. Supposons maintenant que `inf` soit uniquement défini comme une fonction booléenne mais qu'on ne connaisse pas sa spécification. On ne peut alors pas retrouver le type de preuve de `inf` et on ne peut pas retrouver le bon prédicat d'élimination.

On peut alors utiliser une annotation pour préciser la spécification de `inf` et le programme devient :

```
<nat*nat>Match (inf b (S r)) :: {(b<=(S r))} + {(b>(S r))} with
            ((S q),O)
            (q,(S r))
end
```

Exemple 2 : si l'on cherche des exemples plus compliqués d'annotations, on peut à chaque niveau de sous-programme (à chaque niveau de l'arbre de syntaxe abstraite) mettre comme annotation le type de preuve de ce sous-terme. On peut par exemple reprendre l'exemple de la division euclidienne et le surannoter (sans intérêt dans ce cas mais pour montrer le genre d'annotations qui apparaissent).

```
[b,a:nat]<nat*nat>Match a with
   <nat,nat>(O,O) :: :: { (diveucl O b) }
   ([n:nat][h:nat*nat]
      <nat*nat>let (q,r:nat) = h in
       <nat*nat>if (inf_dec b (S r)) :: :: { {(le b (S r))}+{(gt b (S r))} }
                 then <nat,nat>((S q),O)
                 else <nat,nat>(q,(S r)))
      :: :: { (n:nat)(diveucl n b)->(diveucl (S n) b) }
          end :: :: { (diveucl a b) }.
```

Validité

Nous souhaitons avoir une propriété permettant d'assurer la validité de la méthode utilisée. La méthode heuristique peut soit réussir soit échouer. Elle réussit si elle engendre un ensemble de lemmes logiques prouvables (en fait, ce sont des lemmes logiques dont il existerait une preuve dans le terme de preuve correspondant). Bien entendu, nous souhaitons que, dans ce cas, la spécification initiale soit réellement prouvable. Or, le théorème 1 assure que l'inversion de l'extraction faible engendre une preuve de la spécification initiale. A fortiori, la méthode déterministe ne montre que des choses prouvables. La méthode heuristique suit le même cheminement que la méthode déterministe, nous sommes donc assurés de la validité de la méthode heuristique si elle réussit.

Par contre, la méthode peut aussi échouer. Nous considérons deux situations distinctes d'échec. Si la méthode heuristique n'arrive pas à exploiter le programme, soit il manque des informations de spécifications dans le programme (il faut donc ajouter des annotations dans le programme), soit le programme est faux. La méthode peut aussi choisir de mauvaises heuristiques et engendrer des lemmes non prouvables. Il faut alors revenir en arrière et ajouter des informations de spécifications dans le programme. Nous en arrivons ainsi au problème de la complétude de la méthode.

Complétude

L'échec de la méthode heuristique sur un programme correct provient d'un manque d'informations logiques dans le programme. Par contre, la méthode sur les programmes faiblement extraits est déterministe. Donc, nous pouvons affirmer que la méthode heuristique réussit toujours sur des programmes *suffisamment annotés* qui sont des traces de preuve, c'est-à-dire au pire si le programme annoté est le programme faiblement extrait. Par contre, pour d'autres programmes, nous ne pouvons rien affirmer.

Alors, par le déterminisme de l'inversion de l'extraction faible, nous pouvons affirmer la proposition suivante de complétude faible :

Proposition 10 *Si S est prouvable, si le programme associé p est suffisamment annoté pour être la trace d'une preuve de S, alors les lemmes logiques engendrés par la méthode heuristique sont prouvables.*

Cela signifie que la méthode heuristique sur des programmes faiblement extraits est complète au même titre que la méthode déterministe, c'est-à-dire que toute spécification qui peut être prouvée de façon interactive et donner lieu à un programme, peut être prouvée par la méthode de synthèse de preuves. En effet, si les programmes considérés sont les programmes faiblement extraits alors les heuristiques fonctionnent toujours. La notion vague de *programme suffisamment annoté* correspond donc au pire à la notion de terme faiblement extrait. Dans la pratique, les annotations les plus courantes sont typiquement des prédicats d'élimination. En effet, comme les éliminations correspondent au niveau des programmes à des structures récursives ou des analyses par cas, le problème est de retrouver les spécifications de ces parties délicates. Nous retombons sur le problème connu de la logique de Hoare avec les programmes impératifs quant à la recherche d'invariants de boucle.

Conclusion

Dans ce chapitre, nous avons défini une fonction d'extraction faible pour laquelle nous savons montrer le déterminisme de l'inversion. Ceci signifie qu'étant donnés une spécification et un programme faiblement extrait cohérent avec cette spécification, nous pouvons engendrer des lemmes logiques qui s'ils sont prouvables assurent la correction du programme.

Nous avons donc ainsi une représentation condensée des preuves et non plus des programmes *exécutables*. Nous avons surtout une méthode de synthèse de preuves à partir de programmes, plus intéressante pour un programmeur que la synthèse de programmes à partir de preuves car censée demander moins d'expertise.

Mais, les programmes faiblement extraits ne sont pas des programmes naturels pour un programmeur. Nous avons donc montré comment écrire des programmes plus naturels dans un style fonctionnel (programmes extraits) et comment, en utilisant des annotations sur ces programmes, nous gardions une méthode de synthèse de preuves valide et complète. Les annotations représentent pour le programme des informations sur les propriétés de sous-programmes (comme des invariants par exemple).

Cependant, les programmes considérés ne sont pas encore sous une forme complètement naturelle. Nous avons considéré des programmes extraits mais pas des programmes extraits optimisés (problème des types singletons). De plus, nous n'avons pas de récursion générale. Dans le chapitre suivant, nous nous proposons d'améliorer les programmes considérés pour nous rapprocher le plus possible de programmes fonctionnels naturels.

Chapitre 4

Optimisations et Implantation

Le chapitre précédent décrit une méthode correcte et complète de synthèse de preuves à partir de programmes. Les programmes considérés sont des programmes correspondant à des programmes extraits. Ce ne sont pas des programmes optimisés, ils sont même assez lourds. En effet, nous avons considéré que nous inversions une fonction d'extraction (pour des raisons que nous avons déjà expliquées) mais une fonction non optimisée. Dans le chapitre 2, nous avons présenté une fonction d'extraction qui optimise l'extraction des types singletons. Dans ce chapitre, nous allons entre autres ajouter cette considération. En fait, le but de ce chapitre est de montrer comment écrire des programmes dans un style fonctionnel le plus naturel possible et comment retrouver les preuves correspondantes. En effet, les programmes extraits ne sont pas optimisés car ils conservent une lourdeur due aux nécessités de la preuve sous-jacente. Le problème de considérer des programmes plus naturels que les programmes extraits est de savoir retrouver la correspondance au niveau de la preuve. Bien entendu, les programmes considérés sont encore écrits sous forme de λ-termes et nous ne sommes pas encore à l'étape d'écriture de programmes ML. Dans les optimisations sur l'écriture des programmes, nous proposons plusieurs aspects. Tout d'abord, l'introduction de programmes récursifs équivalents au `let rec` de ML, c'est-à-dire l'introduction de la récursivité générale et non plus uniquement de la récursion primitive. Nous détaillerons ensuite le problème des types inductifs isomorphes dont nous avons déjà parlé brièvement dans 3.1.3. Puis, nous détaillerons le processus d'inversion de l'extraction des types singletons qui entraîne une grande amélioration dans l'écriture des programmes et demande des manipulations assez compliquées pour retrouver un terme de preuve. La partie suivante concerne d'autres aspects similaires de transformations sur les programmes pour retrouver la preuve. On peut illustrer un des problèmes sur un exemple : il est très fréquent que l'extraction génère des sous-programmes de la forme `if b then true else false` qui se condensent trivialement en `b` pour le programmeur. Le problème est donc de comprendre comment *inverser* cette optimisation. Dans le même ordre d'idée, il est difficile de retrouver des processus de réécriture. Le problème est de savoir comment les retrouver et dans quelle mesure cela est possible. Enfin, nous pouvons optimiser l'écriture des analyses par cas récurrentes en jouant sur l'emplacement des hypothèses de récurrence.

4.1 Récursivité générale

La seule possibilité de récursion dans le Calcul des Constructions Inductives est la récursion primitive. Il semble naturel d'autoriser l'écriture de programmes récursifs généraux. Mais, pour pouvoir prouver de tels programmes, il faut pouvoir en assurer la terminaison. Nous utilisons pour cela un principe de récurrence bien fondée qui peut être réalisé par un programme récursif. Un principe de récurrence bien fondée a la forme :

$$\forall P \forall R (bien\ fond\acute{e}\ R) \rightarrow (\forall x (\forall y\ (R\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x)) \rightarrow \forall a\ (P\ a)$$

Ce qui signifie que, si on veut montrer $\forall a (P\ a)$ pour une propriété P quelconque, alors il faut montrer pour un ordre donné bien fondé $(\forall x (\forall y\ (R\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x))$. On a bien une récurrence généralisée puisque, pour prouver $(P\ x)$, il faut prouver $(P\ y)$ pour tous les y plus petits que x au sens de l'ordre bien fondé. La bonne fondation de l'ordre assure la terminaison.

Ainsi, l'introduction de programmes récursifs généraux n'est qu'un sucre syntaxique. Elle consiste uniquement en l'ajout d'une syntaxe au-dessus de ce principe de récurrence bien fondée. Tout programme récursif est alors transformé en l'application du précédent principe.

Cependant, le problème de ceci est la détermination d'un ordre bien fondé. En effet, le programmeur définit un programme récursif sans se soucier de l'ordre bien fondé sous-jacent. D'autre part, il semble difficile de retrouver de façon automatique cet ordre. L'aide du programmeur est donc nécessaire à cette étape. Il doit fournir explicitement l'ordre sous-jacent. Naturellement, le programmeur doit aussi prouver manuellement que l'ordre qu'il fournit est bien fondé. On peut envisager d'automatiser la preuve de bonne fondation d'un ordre. [Lec94] propose une méthode d'automatisation de preuves constructives de bonne fondation qui pourrait peut-être être utilisée ici.

Exemple : Algorithme de division euclidienne.

L'algorithme de la division euclidienne est récursif. On note `<P>rec nom-du-programme {nom-de-l'ordre} ...` pour la syntaxe ML `let rec nom-du-programme ... = ...` où `P` est le type du résultat. Le programme est le suivant :

```
[b:nat]<nat*nat>rec div { lt }
        [a:nat]<nat*nat>if (inf b a) then
                <nat*nat>let (q,r:nat) = (div (minus a b)) in
                               <nat,nat>((S q),r)
                         else <nat,nat>(0,a).
```

La spécification correspondante est : $\forall b.(b > 0) \Rightarrow \forall a.\exists q \exists r.(a = b * q + r) \wedge (b > r)$.

Les notations `if ... then ... else ...` et `let ... = ... in ...` ont le même sens qu'en ML et où `lt` représente l'ordre strict sur les entiers naturels, `div` et `minus` la division et la soustraction sur les entiers naturels, et `inf` la décidabilité de l'ordre sur les entiers naturels.

En terme de `Match`, on a la correspondance suivante :

`<T>if b then t1 else t2 = <T>Match b with t1 t2 end` et
`<T>let (x1:T1;...;xn:Tn)=t1 in t2 = <T>Match t1 with [x1:T1]...[xn:Tn]t2`.

4.2 Isomorphisme des types inductifs

Nous reprenons dans cette partie la remarque de la section 3.1.3 sur les types inductifs isomorphes. Nous avions vu dans cette section sur le typage des programmes que, si un terme de programme était bien typé de type *bool*, alors il était aussi de type (*sumbool* A B) pour A et B quelconques. En fait, nous souhaitons identifier, au niveau du typage, des types dont l'extraction est la même (par extraction, nous entendons extraction de [PM89a] et pas extraction faible). De façon usuelle, la notion d'égalité sur deux types inductifs est très stricte. Deux types inductifs sont égaux s'ils ont le même nom et les mêmes constructeurs, le tout modulo la $\beta\iota$-équivalence (voir chapitre 1). Cette idée d'isomorphisme est donc beaucoup plus forte.

La raison peut s'expliquer sur un exemple. Si nous reprenons l'exemple présenté sur l'extraction dans le chapitre 2, nous avons une spécification $(n : nat)\{m : nat|(S\ m) = n\} + \{0 = n\}$ pour une fonction qui calcule le prédécesseur sur les entiers naturels. Cette fonction est une fonction qui, dans un cas, renvoie une valeur, dans l'autre cas ($n = 0$) ne renvoie rien (car il n'y a pas de prédécesseur pour 0). Cette fonction utilise le comportement des exceptions, c'est-à-dire dans le premier cas elle renvoie une valeur, dans le second elle échoue. Le type inductif codant ce comportement peut se noter :

$$Exc = \mathsf{Ind}[A : Set](X : Set)\{value : A \to X|error : X\}$$

Le programme correspondant à la fonction s'écrit alors :

```
[n:nat]<(Exc nat)>Match n with
          (error nat)
          [y:nat][H:(Exc nat)](value nat y)
       end
```

La spécification a pour type extrait : $nat \to (sumor\ nat)$. On considère bien entendu l'extraction optimisée sur les types singletons. Le programme a pour type : $nat \to (Exc\ nat)$. On veut que le programme et la spécification soient cohérents. Donc, on veut que le programme soit aussi de type $nat \to (sumor\ nat)$. Or, l'extraction de $sumor$ est bien la même que celle de Exc (qui est elle-même en l'occurrence). En effet, l'extraction de $sumor$ est :

$$\varepsilon_1(sumor) := \mathsf{Ind}[\bar{A} : Set](\bar{X} : Set)\{\overline{inleft} : A \to X|\ \overline{inright} : X\}$$

L'intérêt d'un tel comportement est d'écrire des programmes avec des types beaucoup plus naturels pour le programmeur puisque généralement il considérera des types extraits dans son programme, beaucoup plus simples que des types de preuves.

4.3 Problème des types singletons

Dans cette section, nous considérons des programmes extraits optimisés c'est-à-dire des programmes dans lesquels les types singletons sont extraits. Pour les définitions d'un type singleton et de l'extraction sur les types singletons, on se réfèrera au chapitre 2 section 2.2. Nous allons reprendre cette extraction et décrire une méthode heuristique de reconstruction de preuve associée.

Si nous supposons que Ind est un type singleton sur A, alors :

$$\varepsilon_1(\mathsf{Ind}[p_1 : P_1; \ldots; p_n : P_n](X : I)\{c_1 : C_1\}) = [p_i : \varepsilon_0(P_i)]A(p_i) \text{ pour } P_i : TypeSet$$

$$\varepsilon_2(\mathsf{Elim}(c, P)\{f_1\}) = (\varepsilon_2(f_1)\ \varepsilon_2(c))$$

$$\varepsilon_2(Constr(i, ind, \vec{p})) = (([p_i : \varepsilon_0(P_i)][x : A]x)\ \varepsilon_1(\vec{p}))$$

Le problème du type inductif est similaire à celui évoqué dans la méthode déterministe du chapitre précédent. Les problèmes délicats sont les cas constructeur et élimination. Le problème se divise en deux étapes : dans un premier temps, comment se rendre compte que l'on est dans le cas d'une construction de type singleton et ensuite comment retrouver cette construction elle-même.

En effet, dans les deux cas, alors que la structure du programme n'est ni un constructeur ni une élimination, il faut savoir que l'on cherche un constructeur ou une élimination. Pratiquement, on se rend compte qu'on cherche un constructeur ou une élimination si on a un type singleton dans la spécification. En fait, on peut fonctionner non pas a posteriori c'est-à-dire juste au moment où on tombe sur un type singleton dans la spécification d'une application mais plutôt a priori. En effet, si, lorsqu'on introduit une variable informative, on a dans le programme une variable de type A et dans la spécification une variable de type un type singleton sur A, on sait que forcément dans la suite on voudra faire une élimination sur cette variable (ou du moins, ça ne coûte rien de la faire). Aussi, au lieu d'attendre l'apparition de cette variable dans la suite du programme, on peut tout de suite faire l'élimination.

Malheureusement, ceci ne peut concerner que les cas où on a une abstraction de type A dans le programme et un produit de type un type singleton sur A dans la spécification. Ceci ne concerne pas des cas comme une abstraction sur $A \rightarrow B$ dans le programme et un produit sur $A \rightarrow sing(B)$ dans la spécification ou $A \rightarrow B$ dans le programme et $sing(A) \rightarrow B$ dans la spécification, où $sing(A)$ signifie type singleton sur A.

Montrons maintenant comment reconstruire le bon terme de preuve. La stratégie globale est la suivante : les éliminations sur des types singletons sont faites au plus tôt (comme nous l'avons expliqué plus haut), les constructeurs manquants de type singleton sont rajoutés au dernier moment quand nécessaire. Pour le cas constructeur, c'est exactement la même chose que pour le cas constructeur de la méthode heuristique (on sait qu'on cherche le premier constructeur; il n'y en a qu'un puisqu'il s'agit d'un type singleton). Reconstruire une élimination est plus délicat. En effet, il faut retrouver le prédicat d'élimination P. Comme dans le cas de la méthode heuristique, il s'agit de trouver l'unificateur le plus liant modulo d'éventuelles introductions logiques.

Nous avons déjà souligné que les seules éliminations qui peuvent être faites a priori sont celles correspondant à une variable de type A dans le programme et une variable de type $sing(A)$ dans la spécification. Cela signifie qu'une application dans un programme peut toujours correspondre à une élimination dans la preuve. Nous allons détailler les cinq cas possibles pour une application $(p\ q)$; p' et q' sont les termes de preuves correspondants.

1. $p' : sing(A) \rightarrow B$ et $q' : A$.

   Ce cas est particulier car il ne correspond pas à une élimination mais à un constructeur manquant. Le terme de preuve qui doit être reconstruit est $(p'\ (\mathsf{Constr}(1, sing(A))\ \vec{arg}_1\ q'\ \vec{arg}_2))$ où $\vec{arg}_1$ et $\vec{arg}_2$ sont les arguments de $\mathsf{Constr}(1, sing(A))$ autres que q'.

2. $p' : A \rightarrow B$ et $q' : sing(A)$.

   Ce cas est simple et correspond à une élimination sur q'. Le terme de preuve qui peut être construit a la forme suivante : $(p'\ {<}P{>} Match\ q'\ with\ [\vec{x} : \vec{X}]x_i\ end)$ où $x_i : A$. Ce terme est en fait transformé en $({<}Q{>} Match\ q'\ with\ [\vec{x} : \vec{X}](p'\ x_i)\ end)$. Cette transformation revient à ramener l'élimination, au plus tôt possible, pour essayer de simplifier au maximum la preuve engendrée.

3. $p' : sing(A) \rightarrow sing(B)$ et $q' : A$.

   Ce cas est une composition de 1 et 2.

4. $p' : sing(A \rightarrow B)$ et $q' : A$.

   Ce cas correspond à une élimination sur p'. Le terme de preuve qui peut être construit a la forme suivante : $(({<}P{>} Match\ p'\ with\ [\vec{x} : \vec{X}]x_i\ end)\ q')$ où $x_i : A$. Ce terme est en fait transformé en $({<}Q{>} Match\ p'\ with\ [\vec{x} : \vec{X}](x_i\ q')\ end)$. Cette transformation revient encore une fois à ramener l'élimination au plus tôt possible pour essayer de simplifier au maximum la preuve engendrée.

5. $p' : A \rightarrow sing(B)$ et $q' : A$.

   Ce cas correspond à une élimination sur $(p\ q)$. En fait, il n'entre pas précisément dans le cadre de l'optimisation que nous présentons dans cette section mais plutôt dans le cadre de la section 4.4.1. Nous indiquons tout de même le terme de preuve engendré mais on se référera à la section 4.4.1 pour des explications. Le terme de preuve construit est :($< P >$ $Match\ (p'\ q')\ with\ [\vec{x} : \vec{X}]x_i\ end$) où $x_i : B$.

Nous avons donc une méthode qui permet de reconstruire de façon heuristique un terme de preuve prenant en compte des types singletons alors que le programme considéré ne contient aucune référence à de tels types. L'intérêt de cette optimisation est de pouvoir écrire des programmes beaucoup plus proches du style ML que sinon. L'intelligence de l'optimisation réside dans la possibilité de réengendrer les transformations nécessaires sur la preuve. Nous allons illustrer sur un exemple cet intérêt.

Exemple : La fonction d'Ackermann.

Cette fonction est définie comme suit :

$$\begin{aligned} ack(0,n) &= n+1 \\ ack(n+1,0) &= ack(n,1) \\ ack(n+1,m+1) &= ack(n, ack(n+1,m)) \end{aligned}$$

Cette définition peut s'exprimer en terme de définition inductive comme un prédicat ternaire avec la syntaxe *Coq* suivante :

```
Inductive Ack : nat->nat->nat->Prop :=
     Ack0 : (n:nat)(Ack 0 n (S n))
   | AcknO : (n,p:nat)(Ack n (S O) p)->(Ack (S n) O p)
   | AckSS : (n,m,p,q:nat)(Ack (S n) m q)->(Ack n q p)
                           ->(Ack (S n) (S m) p).
```

Cette syntaxe correspond au type inductif suivant :

$$\begin{aligned} Ack \equiv Ind\ (X : nat \rightarrow nat \rightarrow nat \rightarrow Prop)& \\ & Ack0 : (n : nat)(X\ 0\ n\ (S\ n)) \\ & \mid AcknO : (n,p : nat)(X\ n\ (S\ O)\ p) \rightarrow (X\ (S\ n)\ O\ p) \\ & \mid AckSS : (n,m,p,q : nat)(X\ (S\ n)\ m\ q) \rightarrow (X\ n\ q\ p) \rightarrow (X\ (S\ n)\ (S\ m)\ p)\} \end{aligned}$$

Supposons maintenant qu'on veuille prouver $\forall n.\forall m.\exists p.Ack(n,m,p)$. Une fonction ML qui correspond à cette spécification est :

```
let rec ack n m = match n with
         0    -> (fun m -> m+1)
       | n'+1 -> (match m with
                          0    -> ack n' 1
                        | m'+1 -> ack n' (ack (n'+1) m')) ;;
```

Ce programme s'écrit, si nous considérons des programmes sans types singletons, dans une forme ${F_\omega}^{Ind}$ par :

```
[n:nat]<nat->nat>Match n with
  [m:nat](S m)
  [y:nat][H:nat->nat][m:nat]
    <nat>Match m with
         (H (S 0))
         [m':nat][H':nat](H H') end end
```

Si nous ne considérions pas des programmes sans types singletons, le programme aurait une forme beaucoup moins agréable (plus éloignée du style ML) :

```
[n:nat]<nat->(sig nat)>Match n with
  [m:nat](exist nat (S m))
  [y:nat][H:nat->(sig nat)][m:nat]
    <(sig nat)>Match m with
               <(sig nat)>Match (H (S 0)) with
                          [x:nat](exist nat x) end
               [m':nat][H':(sig nat)]
                  <(sig nat)>Match H' with
                     [x:nat]<(sig nat)>Match (H x) with
                              [y:nat](exist nat y)
                              end end end end
```

Le premier programme est beaucoup plus agréable et c'est celui que nous considérons dans la suite. Nous allons illustrer les différents points de ce programme relatifs au problème des types singletons. Nous donnons pour chaque cas le sous-programme et la sous-spécification qui lui est associée. La spécification de départ est notée $(n, m : nat)\{p : nat|(Ack\ n\ m\ p)\}$.

- Programme : `(S m)`, Spécification : $\{p : nat|(Ack\ 0\ m\ p)\}$.

  C'est le cas où il faut rajouter un constructeur. Le terme de preuve qui peut être reconstruit est `(exist (Ack 0 m) (S m) L1)` où $L1$ est un terme de preuve logique qui correspond à un lemme logique.

- Programme : `(H (S 0))`, Spécification : $\{p : nat|(Ack\ (S\ y)\ 0\ p)\}$.

  Ce cas correspond à une élimination et plus précisément au cas 5. L'hypothèse H a pour type de preuve $(m : nat)\{p : nat|(Ack\ y\ m\ p)\}$. $(H\ (S\ 0))$ est donc de type $\{p : nat|(Ack\ y\ (S\ 0)\ p)\}$ ce qui n'est pas exactement la conclusion. En effet, par application de $Ackn0$, $\{p : nat|(Ack\ y\ (S\ 0)\ p)\}$ se réduit en $\{p : nat|(Ack\ (S\ y)\ 0\ p)\}$ qui correspond bien à la spécification. Le terme de preuve partiel doit donc être le suivant, avec _ correspondant à une partie que nous ne détaillons pas car non nécessaire à la compréhension :

  ```
  <_>Match (H (S 0)) with
     [x:nat][p:(Ack y (S 0) x))]x
  end
  ```

– Programme : `[H':nat](H H')`,
Spécification : $\{p : nat|(Ack\ (S\ y)\ m'\ p)\} \rightarrow \{p : nat|(Ack\ (S\ y)\ (S\ m')\ p)\}$.
Ce cas correspond au cas simple d'une élimination qui peut être faite a priori. On fait donc une élimination sur H' dont le type de preuve est $\{p : nat|(Ack\ (S\ y)\ m'\ p)\}$. On obtient un élément x de type nat qu'on substitue à H' dans le reste du terme avec la propriété sur x que $(Ack\ (S\ y)\ m'\ x)$. On a le terme de preuve partiel suivant :

```
<_>Match H' with
   [x:nat][p:(Ack (S y) m' x)](H x)
end
```

– Programme : `(H x)`, Spécification : $\{p : nat|(Ack\ (S\ y)\ (S\ m')\ p)\}$.
Ce cas correspond à une élimination et en particulier au cas 5 encore une fois. L'hypothèse H a pour type de preuve $(m : nat)\{p : nat|(Ack\ y\ m\ p)\}$. $(H\ x)$ a donc pour type de preuve $\{p : nat|(Ack\ y\ x\ p)\}$ ce qui n'est pas la conclusion. Mais, par application de $AckSS$, $\{p : nat|(Ack\ y\ x\ p)\}$ avec l'hypothèse $(Ack\ (S\ y)\ m'\ x)$ sur x se réduit en $\{p : nat|(Ack\ (S\ y)\ (S\ m')\ p)\}$. Le terme de preuve partiel doit donc avoir la forme suivante, où _ correspondant à une partie que nous ne détaillons pas car non nécessaire à la compréhension :

```
<_>Match (H x) with
   [x':nat][p':(Ack y x x')]x'
end
```

Il faut bien noter que nous avons toujours parlé de terme de preuve partiel c'est-à-dire que nous construisons petit à petit le terme de preuve. Il se peut que par la suite des sous-parties de ce terme de preuve partiel soient modifiées. Typiquement, dans le cas `[H':nat](H H')`, nous engendrons un terme de preuve qui est modifié dans le cas suivant `(H x)`.

4.4 Transformations de programmes optimisés

Dans cette section, nous détaillons des transformations de preuve dans la même optique que le problème des types singletons mais dans d'autres situations. Pour présenter chacun des points développés dans la suite, nous montrons comment l'écriture de programmes extraits peut être optimisée et donc comment nous pouvons inverser ces optimisations.

4.4.1 Compression d'expressions

Une forme très courante de programmes extraits engendrés par l'extraction est la structure `if b then true else false`. Un programmeur n'écrit jamais de telles trivialités et le programme se réduit à `b`. Le but de l'optimisation présentée dans cette section est de retrouver la preuve correspondant à `b` dans un tel contexte. En fait, on peut avoir un programme correct pour une spécification Q mais dont la preuve p ne soit pas une preuve de Q mais de Q' avec $\varepsilon(Q) = \varepsilon(Q')$. Il faut alors retrouver comment transformer $p : Q'$ en $p' : Q$.

Prenons tout d'abord un exemple pour illustrer ce problème des programmes extraits. Supposons qu'on veuille montrer la spécification suivante de décidabilité de l'ordre sur les entiers naturels : $(n, m : nat)\{(n \leq m)\} + \{(n > m)\}$. La preuve est une analyse par cas sur n. Si $n = 0$ alors $(0 \leq m)$ pour tout m. Sinon, $n = n' + 1$, on fait une analyse par cas sur m. Si $m = 0$, alors $(n' + 1 > 0)$ pour tout n'. Sinon, $m = m' + 1$, et on sait que $(m : nat)\{(n' \leq m)\} + \{(n' > m)\}$.

Si on appelle H cette propriété, $(H\ m')$ est de type $\{(n' \leq m')\} + \{(n' > m')\}$ et on veut montrer $\{(n'+1 \leq m'+1)\} + \{(n'+1 > m'+1)\}$. Pour cela, on distingue deux cas sur H. Soit $\{(n' \leq m')\}$ et alors $\{(n'+1 \leq m'+1)\}$, soit $\{(n' > m')\}$ et alors $\{(n'+1 > m'+1)\}$. Le programme extrait de cette preuve est :

```
[n:nat]<nat->bool>Match n with
        [m:nat] true
        [n':nat][H:nat->bool][m:nat]
           <bool>Match m with
                   false
                   [m':nat][H':nat]
                      <bool>Match (H m') with
                             true
                             false end end end
```

On voit bien apparaître un `<bool>Match (H m') with true false end`. Le programme naturel que l'on aimerait écrire est plutôt le suivant :

```
[n:nat]<nat->bool>Match n with
        [m:nat] true
        [n':nat][H:nat->bool][m:nat]
           <bool>Match m with
                   false
                   [m':nat][H':nat](H m') end end
```

C'est donc ce que nous proposons ici. Le problème est donc de savoir encore une fois comment se rendre compte quand une telle situation se produit. En fait, c'est très simple. Cette situation correspond au cas où le type du terme de preuve ne correspond pas à la spécification mais est convertible avec elle. Il faut alors faire une élimination sur le terme à savoir reconstruire par exemple le `if b then true else false` à partir du `b` (le `if` est en fait un `Match` sur les booléens).

C'est donc exactement le problème que nous avons rencontré dans la section précédente avec l'exemple d'Ackermann. Par exemple, nous avions un terme $(H\ (S\ 0))$ dont le type était $\{p : nat|(Ack\ y\ (S\ 0)\ p)\}$ alors que la spécification était $\{p : nat|(Ack\ (S\ y)\ 0\ p)\}$. Mais, nous savions par une propriété $Ackn0$ que $\{p : nat|(Ack\ y\ (S\ 0)\ p)\}$ se réduisait en $\{p : nat|(Ack\ (S\ y)\ 0\ p)\}$. Il suffisait donc de faire une élimination sur $(H\ (S\ 0))$.

4.4.2 Eliminations

Cette section est encore une inversion d'une optimisation qui peut être faite sur les programmes extraits. Considérons un programme récursif général et supposons qu'une analyse par cas apparaisse dans ce programme. Si une hypothèse (généralement l'hypothèse de récurrence) apparaît dans chacun des cas de l'élimination alors le programme peut être transformé en enlevant cette hypothèse de chacun des cas de l'élimination pour la laisser apparaître une seule fois, avant l'élimination. C'est la façon courante d'écrire des programmes. Si le programme est `<A->B>Match n with [x:A]t1...[x:A]tm end`, alors un programme plus naturel est `[x:A]<B>Match n with t1...tm end`[1].

Dans notre optique, nous voulons considérer des programmes dont l'hypothèse de récurrence est à l'extérieur de l'élimination donc de la forme `[x:A]<B>Match n with t1...tm end` et transformer

1. Noter cependant que cette optimisation ne peut être faite que dans le cas d'éliminations non récursives.

de tels programmes sous la forme `<A->B>Match n with [x:A]t1...[x:A]tm end`. En effet, au niveau de la preuve, il est important de distinguer l'hypothèse de récurrence pour chaque cas de l'élimination. De façon générale, l'hypothèse de récurrence dépend de la variable sur laquelle l'élimination est faite et donc cette hypothèse change dans chaque cas. Il est donc important de la déplacer à l'intérieur de l'élimination si l'on souhaite arriver à faire la preuve.

Prenons l'exemple du tri rapide pour illustrer ce problème. La spécification du tri rapide (ou de n'importe quel autre algorithme de tri d'ailleurs) est : $\forall l.\exists m.(triee\ m) \wedge (permut\ l\ m)$, si *triee* et *permut* sont deux prédicats, l'un signifiant qu'une liste est triée, l'autre que la première liste est une permutation de la deuxième. Une fonction ML qui vérifie cette spécification est la suivante, si `partition` est une fonction prédéfinie qui divise une liste en deux à l'aide d'un pivot :

```
let rec quick l = match l with
                    [] -> []
                  | a::m -> let (l1,l2) = (partition a m) in
                              (quick l1)@[a]@(quick l2) ;;
```

Cette fonction s'écrit dans ${F_\omega}^{Ind}$ sous la forme suivante. On utilise la possibilité d'écrire des programmes récursifs généraux vue au 4.1. La fonction *mil* prend un élément a et deux listes l_1 et l_2 et concatène le tout en $l_1@[a]@l_2$, la fonction *ltl* représente l'ordre strict d'inclusion sur les listes grâce à leur longueur.

```
<list>rec quick (: ltl :)
   [l:list]<list>Match l with
       (* nil *) nil
  (* cons a m *) [a:A][m:list][h:list]
                   <list>Match (partition a m) with
                      [l1,l2:list](mil a (quick l1) (quick l2))
                   end
           end
```

Comme nous l'avons vu au 4.1, cette syntaxe n'est qu'une macro pour l'application d'un principe de récurrence bien fondée `well-founded-induction`. Le programme s'écrit donc en fait :

```
[x:list](well_founded_induction list list
  [l:list][quick:list->list]
    <list>Match l with
       (* nil *) nil
  (* cons a m *) [a:A][m:list][h:list]
                   <list>Match (partition a m) with
                      [l1,l2:list](mil a (quick l1) (quick l2))
                   end
    end
```

L'hypothèse *quick* correspond au niveau preuve à $\forall y.(ltl\ y\ l) \rightarrow \exists m.(triee\ m) \wedge (permut\ y\ m)$. Elle dépend de l. Il faut donc la scinder en deux, suivant les deux cas de l'élimination sur l, c'est-à-dire une hypothèse *quick*1 dans le cas *nil* qui correspond au niveau preuve à $\forall y.(ltl\ y\ nil) \rightarrow \exists m.(triee\ m) \wedge (permut\ y\ m)$ et une hypothèse *quick*2 dans le cas $(cons\ a\ m)$ qui correspond au niveau preuve à $\forall y.(ltl\ y\ (cons\ a\ m)) \rightarrow \exists m.(triee\ m) \wedge (permut\ y\ m)$.

De façon interne, le programme doit donc être transformé en :

```
[x:list](well_founded_induction list list
  [l:list]<(list->list)->list>Match l with
       (* nil *) [quick1:list->list]nil
  (* cons a m *) [a:A][m:list][h:list][quick2:list->list]
                  <list>Match (partition a m) with
                     [l1,l2:list](mil a (quick2 l1) (quick2 l2))
                  end
    end
```

4.4.3 Problèmes de réécriture

Dans une preuve, on peut avoir besoin de faire de la réécriture. Par réécriture, nous entendons ici quelque chose de très simple : si deux termes p et q sont égaux au sens de l'égalité de Leibniz, alors on peut réécrire l'un en l'autre.

Donc, dans une preuve, si l'on veut montrer $(P\ p)$ alors que l'on a en hypothèse $(P\ q)$, on peut réécrire p en q et montrer $p = q$. Au niveau du programme extrait, on garde trace d'une telle manipulation. Or, dans la réalité, on ne souhaite pas inclure de réécriture dans un programme.

Ceci signifie qu'au niveau du passage programme vers preuve, il faut être capable de se rendre compte d'une réécriture. En fait, c'est encore une fois très simple. De la même façon que précédemment, si la spécification à prouver est $(P\ p)$ et si le type du terme de preuve retrouvé à partir du programme est $(P\ q)$ alors, on réécrit p en q et on aura comme lemme logique $p = q$.

Cette optimisation très simple permet donc un pas supplémentaire vers l'écriture de programmes plus naturels.

4.5 Implantation

La méthode heuristique et les optimisations présentées dans ce chapitre ont été implantées dans le système *Coq* depuis la version 5.8 [DFH+93] à la version 5.10 actuelle [CCF+94] sous la forme d'une tactique appelée `Program` (voir annexe A). Le passage de l'étude théorique à l'implantation n'est pas simple. En pratique, plusieurs problèmes apparaissent. Les problèmes d'optimisations que nous avons dans les paragraphes précédents sont typiquement en majorité apparus lors de l'implantation; notamment, les problèmes de déplacements des hypothèses à l'intérieur des éliminations ou les problèmes des réécritures. Cependant, d'autres points (particulièrement liés à l'implantation) dont nous n'avons pas parlé jusqu'ici doivent être mis en évidence.

4.5.1 Constantes versus Variables

Dans la présentation qui a été faite jusqu'ici, nous n'avons jamais parlé de constantes. Théoriquement parlant, elles ne posent pas de problèmes. Mais, en pratique, une constante a un statut particulier. Il s'agit d'une macro pour un λ-terme plus compliqué. On peut y faire référence uniquement par son nom et son type, ou bien on peut vouloir avoir accès au terme caché sous la constante (on dit qu'on expanse la constante). Les variables, elles, ne posent pas ce problème. Elles n'ont aucune structure cachée. Une première idée naïve peut être de mettre sur le même plan variables et constantes. Mais, cela ne s'avère pas une bonne méthode en général. Ainsi, une constante qui apparaît à l'intérieur d'un programme correspond à l'idée intuitive de sous-programme. Si ce sous-programme est suffisamment spécifié alors il n'est pas nécessaire de connaître sa structure sous-jacente et donc inutile de l'expanser. S'il n'est pas suffisamment spécifié, alors la seule solution est d'expanser la constante.

Le gros problème est donc de décider quand expanser une constante. En fait, la décision prise dans la pratique est de toujours essayer la méthode sans expanser les constantes (c'est-à-dire les considérer de premier abord comme des variables) et, si la méthode échoue, d'expanser alors la constante concernée à l'étape i.

Il y a tout de même une restriction à ce choix. Il n'est valable que pour des constantes qui sont des sous-termes du programme global. Si le programme lui-même est réduit à une constante alors il paraît ridicule de ne pas directement expanser la constante. En fait, cela correspond juste à la situation où on définit au préalable une fonction et où on donne ensuite sa spécification.

Ce problème de dualité entre constantes et variables est donc un problème technique qu'il a fallu résoudre dans la pratique, bien qu'il ne soit pas apparu en théorie.

4.5.2 Couplage avec la tactique Propre

`ProPre` [MS92] est une tactique du système *Coq* qui permet de définir des fonctions sous forme équationnelle. A partir d'une définition équationnelle, ProPre sait retrouver les récurrences structurelles et donc recontruire le λ-terme typé correspondant à la fonction définie. Ceci permet une définition beaucoup plus agréable des fonctions. `Program` se place en aval de `ProPre`. A partir d'un terme défini sous forme de récurrences structurelles, `Program` sait reconstruire les sous-spécifications logiques et donc un terme de preuve partiel pour le programme résultant de `ProPre` avec des obligations de preuve laissées à l'utilisateur.

Exemple : supposons qu'on veuille définir une fonction qui calcule la longueur d'une liste. La fonction présentée est une fonction récursive terminale et prend donc deux arguments (une liste et un entier) et renvoie un entier. La fonction standard de calcul de la longueur revient à l'appel de cette fonction sur 0. Une définition de cette fonction avec ProPre a la syntaxe suivante :

```
Recursive Definition length_l : list -> nat -> nat :=
  nil n => n
| (cons m x) n => (length_l x (S n)).
```

Cette définition est équivalente au programme :

```
[l:list]<nat->nat>Match l with
              [n:nat]n
              [a:A][m:list][H:nat->nat](H (S n))
        end.
```

Typiquement, `length_l` peut alors être utilisé comme programme vérifiant la spécification $(l : list)(n : nat)\{m : nat|(plus\ n\ (length\ l)) = m\}$.

4.5.3 Types mutuellement inductifs

Des types mutuellement inductifs peuvent maintenant être définis dans le système *Coq* (comme les arbres et les forêts par exemple). Les opérateurs de récursion correspondant sont un `Case` (sur le modèle du `Match` non récursif des types inductifs) et un opérateur de point fixe. Il a donc fallu adapter `Program` à ce type de structure (la théorie n'est pas faite). Ceci n'est pas très simple notamment pour le point fixe. On pourra trouver des exemples de définitions par point fixe dans l'annexe A.

Conclusion

Dans ce chapitre, nous avons décrit une liste d'optimisations sur l'écriture des programmes dont on souhaite prouver la correction. Ces optimisations permettent de se rapprocher d'un style de programmation plus proche de ML. En outre, nous avons ajouté une syntaxe qui permet d'écrire des programmes récursifs généraux, basée sur l'utilisation d'un principe de récurrence bien fondée. D'autre part, nous avons introduit des transformations internes sur les programmes et les preuves qui permettent une plus grande souplesse d'écriture. Nous pouvons considérer, grâce aux optimisations sur les types singletons, des programmes qui ne font pas d'éliminations inutiles pour le programmeur. Il n'est plus nécessaire d'écrire des programmes triviaux dans certains cas comme `if b then true else false`. Les éliminations dans les programmes récursifs peuvent s'écrire de façon naturelle c'est-à-dire avec l'hypothèse de récurrence à l'extérieur de l'élimination. De plus, les problèmes de réécriture sont complètement écartés des programmes et se retrouvent uniquement dans des manipulations de preuves. Enfin, nous avons insisté sur les difficultés posées par l'implantation.

Chapitre 5

Autres travaux en preuves de programmes

Le but de ce chapitre est de présenter d'autres techniques permettant de développer des programmes dont la correction vis-à-vis d'une spécification est assurée et qui peuvent être comparées avec ce que nous avons présenté.

Dans un premier temps, nous allons décrire des méthodes dans des cadres non typés. Ces méthodes [Hoa69, MW71, MR92, CG91] sont plutôt vouées aux programmes impératifs. La raison est que des méthodes comme la logique de Hoare [Hoa69] sont fondamentales dans les concepts de base qu'elles proposent et qu'il y a de nombreuses comparaisons avec les idées que nous avons présentées.

Ensuite, dans le cadre de la théorie des types, il existe différentes approches pour assurer la correction de programmes. On peut les diviser en deux groupes : des méthodes qui développent des preuves et extraient des programmes à partir de ces preuves (synthèse de programmes) [PM89b, PM89a, And93, And94a, And94b, Ber94, HN88, Con86] et des méthodes qui construisent en même temps preuve et programme [Pol92, Pol94, BM92]. Le premier aspect nous intéresse particulièrement car l'approche que nous avons choisie dérive directement de cet aspect là. On ne présentera pas à nouveau l'approche [PM89b] mais on citera d'autres systèmes [Ber94, HN88, Con86, And93] dans lesquels le même genre de choses a été fait. L'aspect [And93], suit cependant un cheminement différent même si l'idée de base est la même.

5.1 Logique de Hoare

5.1.1 Contexte

Le propos de la logique de Hoare [Hoa69] est de montrer qu'un programme donné suit un comportement décrit par le programmeur. L'idée de base est qu'une des propriétés les plus importantes d'un programme est de calculer la fonction qu'il est censé implémenter. Cette fonction peut être spécifiée par des assertions générales sur les variables du programme après exécution. Les programmes considérés sont des programmes écrits dans des langages classiques avec affectation et possibilités d'effets de bords.

Les assertions décrivent des propriétés générales sur les variables et les relations qu'elles peuvent avoir entre elles. Les assertions peuvent être écrites en termes de formules du calcul des prédicats du premier ordre. Dans la plupart des cas, la validité des résultats d'un programme dépend des valeurs initiales des variables. Ces valeurs initiales peuvent être spécifiées exactement de la même façon que les valeurs résultats.

On introduit une notation pour indiquer la relation entre une précondition P sur les valeurs initiales des variables, un programme Q et une postcondition sur les valeurs finales R, à savoir $P\{Q\}R$. L'interprétation de cette notation est : si l'assertion P est vraie avant l'exécution du programme Q, alors l'assertion R est vraie après l'exécution du programme Q. La spécification est donc représentée par le couple (P, R). Un programme satisfait cette spécification s'il vérifie $P\{Q\}R$.

5.1.2 Méthode

Le principe est d'utiliser un raisonnement déductif. En fait, pour chaque structure possible du programme Q, on donne une règle d'inférence qui transforme l'expression $P\{Q\}R$. On a une preuve de $P\{Q\}R$ si on arrive à dériver $P\{Q\}R$ uniquement à partir d'axiomes et de règles d'inférence.

Pour un langage simple à la PASCAL, les règles d'inférence sont les suivantes :

$$R_1 \quad \vdash P[x \leftarrow f]\{x := f\}P$$

$$R_2(x, y) \quad Si\ \vdash P\{Q\}R\ et\ \vdash R \Rightarrow S\ alors\ \vdash P\{Q\}S$$

$$R_3(x, y) \quad Si\ \vdash P\{Q\}R\ et\ \vdash S \Rightarrow P\ alors\ \vdash S\{Q\}R$$

$$R_4(x, y) \quad Si\ \vdash P\{Q_1\}R_1\ et\ \vdash R_1\{Q_2\}R\ alors\ \vdash P\{Q_1; Q_2\}R$$

$$R_5(x) \quad Si\ \vdash P \wedge B\{Q\}P\ alors\ \vdash P\{while\ B\ do\ Q\}\neg B \wedge P$$

avec x et y représentant les prémisses des règles d'inférences.

Exemple :

On reprend un exemple de [Hoa69]. Il s'agit de montrer un algorithme de division euclidienne sur des entiers naturels. Cet exemple est une version impérative d'un algorithme de division similaire à l'exemple de division que nous avons donné en introduction.

Le programme Q a la forme suivante (pour la division de x par y) :

```
r:=x; q:=0;
while y<=r do
   r:=r-y;
   q:=1+q
```

Une spécification de ce programme peut être : $(true, \neg(y \leq r) \wedge (x = r + y * q))$, $true$ signifiant qu'il n'y a pas de précondition. Cette spécification rappelle la spécification de la division dans notre cadre vue dans l'introduction, à savoir $\forall a, b(b > 0) \rightarrow \exists q, r(a = b * q + r) \wedge (b > r)$.

La preuve formelle que ce programme vérifie sa spécification est décrite dans la figure 5.1. On remarque que deux lemmes doivent être vérifiés pour que la preuve fonctionne. Ces lemmes doivent préalablement avoir été prouvés pour assurer la correction du programme. On retrouve ici le problème des lemmes logiques laissés à l'utilisateur dans notre méthode de synthèse de preuves.

Ligne	Preuve	Règle utilisée
1	$true \Rightarrow x = x + y * 0$	Lemme 1
2	$x = x + y * 0\{r := x\}x = r + y * 0$	R_1
3	$x = r + y * 0\{q := 0\}x = r + y * q$	R_1
4	$true\{r := x\}x = r + y * 0$	R_3(2,1)
5	$true\{r := x; q := 0\}x = r + y * q$	R_4(4,3)
6	$x = r + y * q \wedge y \leq r \Rightarrow x = (r - y) + y * (1 + q)$	Lemme 2
7	$x = (r - y) + y * (1 + q)\{r := r - y\}x = r + y * (1 + q)$	R_1
8	$x = r + y * (1 + q)\{q := 1 + q\}x = r + y * q$	R_1
9	$x = (r - y) + y * (1 + q)\{r := r - y; q := 1 + q\}x = r + y * q$	R_4(7,8)
10	$x = r + y * q \wedge y \leq r\{r := r - y; q := 1 + q\}x = r + y * q$	R_3(9,6)
11	$x = r + y * q\{while\ (y \leq r)\ do\ \ldots\}\neg(y \leq r) \wedge (x = r + y * q)$	R_5(10)
12	$true\{Q\}\neg(y \leq r) \wedge (x = r + y * q)$	R_4(5,11)

FIG. 5.1 - *Preuve de la division euclidienne en logique de Hoare*

5.1.3 Conclusion et comparaison

Cette méthode n'assure que la correction d'un programme s'il termine mais pas sa terminaison. En fait, l'expression $P\{Q\}R$ doit plutôt se lire "si Q termine, et si P est vraie alors après l'exécution de Q, R est vraie". Cependant, il est facilement possible d'adapter les axiomes de sorte qu'ils ne puissent être utilisés sur des programmes qui ne terminent pas.

Les programmes considérés sont des programmes impératifs. Les spécifications considérées ne sont que des formules du calcul des prédicats du premier ordre. De cette méthode de preuve, on peut déduire une méthode de synthèse de programmes. Etant donnée une spécification (P, R), prouver cette spécification revient à construire un programme Q tel que $P\{Q\}R$ est vraie.

Le problème majeur de cette méthode réside dans les invariants de boucle. En effet, dans l'exemple qu'on a vu, il restait deux lemmes à prouver. Le deuxième lemme correspond à un invariant sur la boucle while. Dans le cas de l'exemple, l'invariant est simple. Dans le cas général, les invariants de boucle ne sont ni faciles à énoncer, ni faciles à prouver.

On peut faire de nombreux points de comparaison entre cette approche et celle présentée dans les chapitres précédents. Les spécifications en logique de Hoare sont très similaires à celles que nous considérons. En effet, nous considérons des spécifications sous la forme $\forall x(P\ x) \rightarrow \exists y(Q\ x\ y)$ et Hoare considère des spécifications sous la forme $P\{R\}Q$. Hoare distingue la précondition et la postcondition. Dans notre approche, il n'y a pas distinction de deux formules différentes du fait de la puissance d'expression inhérente au Calcul des Constructions Inductives mais pourtant on retrouve une partie $(P\ x)$ correspondant à la précondition et une partie $(Q\ x\ y)$ correspondant à la postcondition. Ensuite, la méthode de Hoare fonctionne par analyse sur la structure du programme et modifications successives de la spécification jusqu'à arriver à des axiomes. C'est exactement le cheminement de la synthèse de preuves. On considère un programme et pour retrouver son terme de preuve à partir de sa spécification, on fait une analyse sur sa structure et on génère des sous-spécifications et des sous-programmes suivant le cas. Enfin, le point de comparaison le plus flagrant semble être le problème des invariants de boucle avec la méthode de Hoare. On sait traditionnellement que c'est le problème principal de cette méthode. Cela correspond typiquement à nos problèmes pour retrouver les prédicats d'élimination. Dans la méthode de Hoare, l'intervention du programmeur est nécessaire. Dans la nôtre aussi, par le système d'annotations.

Pour conclure sur cette comparaison, on a donc entre la méthode de Hoare et notre approche des comportements très similaires. Cependant, les deux approches ne se situent pas dans le même domaine. L'une (celle de Hoare) concerne plutôt des programmes impératifs et éventuellement parallèles et considère des spécifications du calcul des prédicats. L'autre (la nôtre) se place dans le cadre de la programmation fonctionnelle avec des spécifications plus expressives (notamment grâce aux types dépendants) dans le Calcul des Constructions Inductives.

5.2 Synthèse de programmes

Dans cette partie, toutes les méthodes présentées sont des méthodes de synthèse de programmes à partir de preuves. Dans tous les cas, une spécification d'un comportement est décrite par une formule dans une certaine théorie et une preuve de cette spécification est faite. C'est le cadre de la méthode [PM89b] à la base de tout ce que nous avons présenté dans cette thèse.

On considère dans ces méthodes des spécifications sous la forme $\forall x.(P\ x) \rightarrow \exists y.(Q\ x\ y)$, c'est-à-dire étant donnés un objet d'entrée x et une preuve de $(P\ x)$, il existe un objet de sortie y et une preuve de $(Q\ x\ y)$. Une preuve d'une telle spécification doit être suffisamment constructive pour indiquer une méthode de calcul de l'objet de sortie. Cette méthode est la base d'un programme qui peut être extrait de la preuve. Chacune des différentes méthodes qui suivent a sa propre méthode d'extraction. De plus, la théorie dans laquelle s'exprime la preuve n'est pas la même dans les différents systèmes.

Cependant, toutes partent de la même considération de synthétiser des programmes à partir de preuves. La structure de la preuve d'un théorème détermine la structure du programme synthétisé. En particulier, une analyse par cas dans une preuve correspond à la formation d'une conditionelle ou d'un test dans le programme. L'utilisation d'un principe de récurrence dans une preuve correspond à une récursivité dans le programme. Si la preuve nécessite des sous-lemmes, le programme appelle des sous-programmes. Naturellement, différentes preuves d'une même spécification mènent à des programmes différents.

La première partie est consacrée à des méthodes dans le cadre du calcul des prédicats du premier ordre. Pour plus de détails sur cette partie, on peut consulter [Pot88]. Les parties suivantes portent sur des méthodes en théorie des types.

5.2.1 Méthodes non typées

La méthode Manna-Waldinger

La méthode de Manna-Waldinger [MW71, MR92] est appelée méthode des *tableaux déductifs*. Les spécifications considérées sont des formules du calcul des prédicats du premier ordre. Les preuves sont présentées sous forme de tableaux, les tableaux déductifs.

Un tableau est un ensemble de lignes. Chaque ligne est constituée soit d'une assertion, soit d'un but auquel peut être associée une expression décrivant le programme construit lors de la déduction du but. Chaque ligne correspond à l'application d'une règle d'inférence.

Les différents types de règles d'inférence sont :

- des règles de transformations sur les assertions et les buts ;
- des règles de découpage d'assertions et de buts ;
- une règle de récurrence basée sur un principe de récurrence bien fondée ;

– des règles de résolution correspondant à de l'analyse par cas.

On a une preuve de la spécification si au moins une instance d'un but est vraie ou si au moins une des assertions est fausse. A chaque application de règle d'inférence correspond une construction de programme. Lorsqu'un but est prouvé, on a dans la ligne correspondante une expression associée à ce but qui décrit le programme synthétisé.

Cette méthode synthétise donc des programmes à partir de preuves de formules du calcul des prédicats du premier ordre. Les programmes sont synthétisés dans un pseudo-langage fonctionnel. Le cadre n'est pas donc pas le même que celui considéré dans cette thèse. La méthode est elle aussi différente puisqu'il s'agit d'une analyse de règles d'inférence et d'une synthèse de programmes.

Autres travaux

Dans le même style de méthodes, on a la méthode développée dans [CG91]. La spécification est représentée par une combinaison d'une pré-formule et d'une post-formule qui sont toutes les deux des formules propositionnelles disjonctives.

La spécification est dite imprécise puisqu'elle spécifie uniquement le *quoi* du problème mais pas le *comment*. Comme on l'a déjà vu, une spécification est ambiguë et il existe plusieurs interprétations correctes de la spécification (donc plusieurs programmes). Les programmes sont construits par analyse de la spécification propositionnelle avec des if-then-else sur les atomes de la spécification. Tous les programmes candidats à la spécification sont ainsi engendrés et certains supprimés par une analyse de redondance à l'intérieur du programme.

Cette méthode est différente car il ne s'agit pas de synthèse mais plutôt de génération *automatique* de programmes. Les spécifications sont du domaine du calcul propositionnel donc d'un cadre beaucoup plus restreint que ce qui a été vu avant. On peut aussi citer [Coh92] qui synthétise des preuves à partir de programmes. Le cadre est celui de la logique du premier ordre.

5.2.2 Extraction de programmes en théorie des types

Les méthodes présentées sont toutes des méthodes de synthèse de programmes à partir de preuves.

"Pruning" de Berardi

Dans l'approche [PM89b], on utilise un marquage des termes pour distinguer termes calculatoires et termes non-calculatoires. Cette approche [Ber94] utilise la méthode dite du *pruning*. Le *pruning* consiste en un remplacement de certains sous-termes par des constantes dont le nom ne sert jamais. On a comme résultat que le *pruning* conserve le comportement observationnel d'un λ-terme simplement typé s'il ne modifie pas son type. On peut alors définir une fonction des λ-termes simplement typés dans les λ-termes simplement typés qui supprime le code redondant d'un λ-terme. On a un algorithme d'extraction. Le même genre d'idée est utilisé dans [Goa80].

Les systèmes PX et Nuprl

L'approche [PM89b] se place dans le cadre du Calcul des Constructions Inductives et plus particulièrement d'une de ses implantations le système *Coq* [DFH$^+$93]. D'autres systèmes de preuves offrent la possibilité d'extraire des programmes, en autres les systèmes PX [HN88] et NuPrl [Con86].

Transformation de preuves

Dans les méthodes d'extraction vues jusqu'ici, des preuves sont développées, des programmes extraits puis éventuellement les programmes extraits sont optimisés (voir chapitres 2 et 4). [And93] propose un autre point de vue. Il s'agit non plus d'appliquer des transformations sur les programmes extraits mais des transformations sur les preuves. Ces transformations sur les preuves peuvent s'appliquer directement à l'optimisation des programmes extraits comme la transformation d'un programme récursif en un programme récursif terminal plus efficace.

5.3 Construction de preuve-programme en simultané

Dans cette partie, les deux points présentés construisent simultanément preuve et programme et non preuve puis programme comme dans la section 5.2.2, ou programme puis preuve comme le point de vue développé dans cette thèse. Les deux approches semblent assez fortement liées avec, cependant, des présentations très différentes. On peut trouver une comparaison des deux approches dans [Pol94].

5.3.1 $\lambda\omega_L$: une logique de programmation

[PM89b] extrait d'une preuve non seulement un programme mais aussi une preuve de sa correction. Nous n'avons pas développé cet aspect dans notre présentation au chapitre 2. En fait, l'extraction consiste en deux fonctions, ε qui extrait le contenu non-calculatoire d'une preuve et $\mathcal{R}$ qui construit une preuve de la correction du programme extrait vis-à-vis de sa spécification. La preuve de correction du programme correspond à la notion de réalisabilité. La notion de réalisabilité [Tro73] est une interprétation du contenu calculatoire des preuves intuitionnistes comme programmes satisfaisant une spécification. La fonction $\mathcal{R}$ calcule en fait la formule de réalisabilité.

[Pol94] considère qu'il est plus pragmatique de construire en même temps et directement le programme et sa preuve de correction. L'avantage est que le programme en construction est visible à chaque étape de la preuve de correction et des choix sur sa conception peuvent être faits au fur et à mesure. En cela, il rejoint les motivations à la base de cette thèse.

Pour comparer les deux approches [PM89b] et [Pol94], il faut insister sur le fait que les deux systèmes logiques considérés ne sont pas les mêmes. [PM89b] considère un langage logique de développement de preuve qui est le Calcul des Constructions avec marquage des termes par *Prop* et *Set* pour différencier les termes calculatoires des termes non-calculatoires, et un langage de programmation dans lequel se situent les programmes extraits qui est le système F_ω. [Pol94] distingue lui-aussi un langage logique et un langage de programmation mais de façons différentes.

Le langage logique considéré est appelé $\lambda\omega_L$. Il est constitué de deux sous-systèmes appelés $\lambda\omega_s$ et $\lambda\omega_p$. $\lambda\omega_s$ est le langage de programmation, il s'agit d'une copie de F_ω avec la règle axiome $Set : TypeSet$. $\lambda\omega_p$ est une autre copie de $F_\omega$avec la règle axiome $Prop : Type$. Mais, on veut pouvoir exprimer des propriétés sur les programmes à travers $\lambda\omega_L$. Pour cela, on complète en rajoutant quatre règles qui entremêlent $\lambda\omega_s$ et $\lambda\omega_p$ pour pouvoir autoriser la formation de preuves dépendantes de programmes.

On a donc un système logique constitué de :

- $\lambda\omega_s$;

- $\lambda\omega_p$;

– toutes les dépendances pour avoir des dépendances des preuves vis-à-vis des programmes: (*TypeSet*,*Type*), (*TypeSet*,*Prop*), (*Set*,*Type*), (*Set*,*Prop*).

Les dépendances qui n'existent pas sont les dépendances des programmes vis-à-vis des preuves, ce qui est naturel puisqu'on veut extraire le contenu non-calculatoire. De plus, il n'y a pas dans $\lambda\omega_L$ les dépendances (*Set*,*TypeSet*) ni (*Prop*,*Type*) ce qui empêche la formation de types dépendants. Au niveau (*Set*,*TypeSet*), cela correspond exactement au principe suivant : le langage de programmation est F_ω(donc pas de types dépendants). Par contre, au niveau du langage logique, ne pas avoir (*Prop*,*Type*) signifie ne pas pouvoir coder des types de données tels que les entiers pairs ou les listes de longueur n.

$\lambda\omega_L$ est donc un système plus *pauvre* que le Calcul des Constructions au sens où il interdit les dépendances d'un type envers un terme. C'est ce qui permet de séparer programme et preuve de correction. Dans [PM89b], on ne peut pas séparer ε et $\mathcal{R}$ car ce sont des définitions mutuellement récursives du fait de la présence des types dépendants. En fait, le Calcul des Constructions est $\lambda\omega_L$ plus la règle (*Prop*,*Type*).

Donc, [Pol94] nous semble proche de [PM89a] dans le fait qu'il construit un programme et sa preuve de correction. Cependant, il est médian selon nous à travail de [PM89a] et à celui présenté dans cette thèse car il repose sur des motivations pratiques de développement de programmes et non pas de développement de preuves. Dans cette optique, on peut noter que [Pol94] introduit une façon d'écrire des programmes récursifs comme nous l'avons introduit dans le chapitre 4. Il autorise l'introduction d'un opérateur de récursion bien fondée qui permet d'écrire des programmes récursifs dont la terminaison est assurée par la bonne fondation de la récursion.

5.3.2 Les "deliverables"

La notion de *deliverables* a été introduite par R.Burstall et J.MacKinna [BM92]. Un deliverable est une paire constituée d'un programme et de sa preuve de correction. Dans l'idée, l'approche est donc similaire à la précédente.

La différence principale tient au langage de programmation considéré. Dans [Pol94], le langage de programmation est le système F_ω. Dans ce cas-ci, le langage de programmation est le λ-calcul simplement typé. Le langage logique considéré est le Calcul des Constructions Etendu (ECC) [Luo90, LP92]. Les paires programme-preuve de correction peuvent être représentées par les types sommes de ECC à l'intérieur même de ECC. Les paires peuvent donc être manipulées de façon externe au système. Cependant, ceci impose des restrictions sur la forme des programmes et des spécifications. On ne peut pas exprimer de relations entre les entrées et les sorties. Une notion de deliverables du second ordre est alors introduite pour pallier à ce problème.

Donc, de la même façon que pour le point précédent, cette approche nous semble se situer entre l'extraction de programmes et ce que nous avons décrit dans cette thèse.

5.4 Conclusion

Ce chapitre a brièvement présenté un certain nombre de méthodes de développement de programmes certifiés. Dans le cadre de la théorie des types, deux types principaux de méthodes nous semblent se détacher. D'une part, des méthodes qui synthétisent des programmes à partir de preuves. Un cas particulier de cette approche est l'extraction de programmes dans le Calcul des Constructions Inductives de [PM89a]. D'autre part, des méthodes qui construisent simultanément un programme et sa preuve de correction. Le point de vue présenté dans cette thèse nous semble appartenir à une troisième catégorie qui est la synthèse de preuves à partir de programmes. Ce point de vue rejoint cependant certaines motivations des méthodes de constructions simultanées de programmes et de leurs preuves de correction, à savoir certaines motivations à visée plus pratique pour permettre au programmeur l'accès direct au programme qu'il développe. Un avantage de la synthèse de preuves proposée est d'être très uniforme. En effet, la notion de synthèse fonctionne sur n'importe quelle formule de contenu calculatoire du Calcul des Constructions Inductives, ce qui fait qu'il s'agit d'une certaine méthode de construction de preuve ou plutôt de description de preuve.

Enfin, la synthèse de preuves à partir de programmes s'inspire fortement des idées de la logique de Hoare. Elles fonctionnent toutes les deux par une analyse de la structure du programme et par générations successives de sous-spécifications et de sous-programmes. Le problème des annotations dans les programmes considérés dans notre méthode se rattache aux problèmes bien connus des invariants de boucle en logique de Hoare. La différence principale entre les deux est cependant le domaine d'application : pour l'un, la programmation fonctionnelle typée, pour l'autre la programmation impérative.

Conclusion

Nous allons faire le point sur les résultats du travail présenté dans cette thèse. Nous avons défini une fonction d'extraction faible. Cette fonction est inversible et la preuve de son inversibilité donne un algorithme de reconstruction de preuves à partir de programmes. Cet algorithme a été mis en œuvre dans le système *Coq*. Enfin, nous conclurons sur les perspectives de ce travail.

Cadre

Cette thèse se place dans le cadre de la validation de programmes en théorie des types et plus particulièrement dans le Calcul des Constructions Inductives. L'isomorphisme de Curry-Howard identifie les notions de preuve et de programme. Ceci est mis en évidence en considérant qu'une preuve est un programme contenant certaines informations non calculatoires c'est-à-dire des informations concernant sa preuve de correction et non pas la façon de faire le calcul lui-même. Cette considération correspond à des techniques d'extraction de programmes à partir de preuves. Dans le cadre que nous considérons, le langage de preuve est le Calcul des Constructions Inductives et le langage de programmation est le système ${F_\omega}^{Ind}$.

Extraction faible

Dans cette thèse, nous avons considéré le point de vue inverse de l'extraction. Un programme peut être vu comme un squelette de preuve : il indique les principales étapes calculatoires de la preuve (les étapes de récurrence, les analyses par cas, etc). Dans cette thèse, nous proposons une méthode de synthèse de preuves à partir de programmes. Etant donnés un programme et sa spécification, nous engendrons une preuve automatique du programme moyennant des obligations de preuve restant à l'utilisateur.

Pour cela, une idée est d'inverser la notion d'extraction. Nous avons donc défini une fonction d'extraction faible qui peut être inversée. Cette fonction engendre des termes d'un nouveau langage qui est intermédiaire entre le Calcul des Constructions Inductives et le système ${F_\omega}^{Ind}$. Ses termes sont des programmes annotés avec des spécifications. Ils ont leur propre notion de typage qui n'est plus la notion standard de typage du Calcul des Constructions Inductives ni celle de ${F_\omega}^{Ind}$. L'intérêt de ces programmes annotés est qu'ils contiennent suffisamment d'informations pour qu'on puisse retrouver de façon automatique, à partir d'un programme annoté et de sa spécification, la preuve dont il est extrait.

Synthèse de preuves

Pour cela, l'inversibilité de la fonction d'extraction faible a été démontrée et la preuve constructive de cette inversibilité fournit un algorithme de reconstruction de preuve. La preuve reconstruite suit la structure du programme et est partielle. Des *trous* correspondant à des obligations de preuve restent. Ces obligations de preuve correspondent à des propriétés logiques que le programme doit vérifier pour valider sa spécification. Cette méthode de synthèse de preuves est valide et complète. L'inconvénient majeur vient du fait que les programmes annotés ne sont pas des programmes naturels pour le programmeur. Une méthode heuristique est donc mise en place sur des programmes plus naturels (c'est-à-dire des programmes de ${F_\omega}^{Ind}$ et non plus des programmes annotés). Telle quelle, cette méthode n'est plus complète car il n'est pas toujours possible de retrouver la spécification d'un sous-programme à partir de la spécification initiale du programme (typiquement, il est impossible de retrouver la spécification d'une récurrence). Cependant, des annotations peuvent être ponctuellement ajoutées (et non plus systématiquement) dans ces programmes pour guider la méthode heuristique. Celle-ci est alors complète pour des programmes suffisamment annotés, c'est-à-dire contenant autant d'informations de spécifications que nécessaire. D'autre part, les programmes ${F_\omega}^{Ind}$ considérés ne sont pas très naturels. Ils contiennent beaucoup d'informations redondantes du fait de la structure de preuve sous-jacente. Pour pallier ce problème, nous avons introduit un certain nombre d'*anti-optimisations*. Nous avons autorisé une écriture des programmes plus naturelle. Ceci a entraîné la conception de techniques de transformations des programmes pour retrouver la redondance nécessaire à la preuve sous-jacente. Finalement, la méthode heuristique proposée prend en compte des programmes de ${F_\omega}^{Ind}$ éventuellement annotés par des spécifications, écrits le plus naturellement possible, et engendre une preuve partielle de leur cohérence vis-à-vis d'une spécification donnée. Des obligations de preuve restent à résoudre par l'utilisateur.

Situation générale

Ce travail peut se comparer à d'autres méthodes de validation de programmes en théorie des types. Nous avons déjà indiqué qu'il prenait le point de vue inverse des méthodes d'extraction de programmes à partir de preuves. Le travail dont il nous semble le plus se rapprocher est celui de la logique de Hoare. La logique de Hoare considère des programmes impératifs et des spécifications sous forme de préconditions et de postconditions. La preuve qu'un programme valide une spécification est guidée là-aussi par la structure du programme. Un problème connu de cette approche est la recherche d'invariants. Dans de tels cas, l'intervention de l'utilisateur est nécessaire. Dans notre cas, elle est aussi nécessaire par l'utilisation des annotations. L'utilisation des annotations se fait généralement dans des cas de programmes récursifs ou utilisant des analyses par cas, ce qui correspond au problème des invariants en logique de Hoare. Ce qui différencie principalement notre approche de celle de la logique de Hoare est le cadre : dans notre cas, on considère des programmes fonctionnels typés et des spécifications dans une logique d'ordre supérieure ; en logique de Hoare, on considère des programmes impératifs et des spécifications dans une logique du premier ordre.

Aspect pratique

Ce travail a été mis en œuvre dans le système Coq. Il s'agit d'une tactique à l'aide de laquelle le développeur donne une spécification calculatoire et un programme. La tactique associe le programme à la spécification puis déstructure le programme et associe des sous-spécifications calculatoires à des sous-programmes jusqu'à n'obtenir plus que des lemmes logiques. Le passage de l'étude théorique à l'implantation pratique n'est pas direct. De nombreux problèmes se posent quant aux décisions à prendre sur les heuristiques. L'implantation actuelle est utilisable et utilisée par des personnes extérieurs aux concepteurs du système *Coq*. Une bibliothèque d'exemples a été développée. Il s'agit d'exemples de tris sur des listes, d'algorithmes sur des graphes, des arbres, d'algorithmes d'arithmétique simple .... Il s'agit d'exemples d'algorithmique standard et non pas d'exemples d'école. Il apparaît qu'il faut une certaine expertise pour utiliser cette tactique. Les lemmes engendrés varient beaucoup en complexité, de propriétés très simples à des lemmes demandant un développement manuel assez long. Une partie importante a été consacrée à faire évoluer la tactique pour qu'elle accepte de prendre en compte des programmes écrits de manière la plus naturelle possible. Le premier prototype prenait des termes extraits bruts, actuellement on peut écrire certaines fonctions sous formes d'équations en utilisant une autre tactique appelée ProPre due à [MS92].

Perspectives

On peut penser explorer d'autres méthodes de synthèse de preuve. En effet, il serait possible de travailler avec le programme f et la preuve à faire de $\forall x.(P\ x) \Rightarrow (Q\ x\ (f\ x))$. Cette formule exprime que f réalise $\forall x.(P\ x) \Rightarrow \exists y.(Q\ x\ y)$. Les techniques développées ne devraient pas être très différentes sauf qu'on peut à priori aller plus loin, car on peut parfois faire la preuve de "f réalise B" alors qu'on ne sait pas faire la preuve de B.

On pourrait aussi imaginer de prendre en compte des programmes partiels. Cela permettrait de faire uniquement des parties relativement simples ou du moins facilement mécanisables avec la méthode de synthèse de preuves et laisser les parties de preuve plus délicates directement à l'utilisateur. En l'état actuel, on donne un programme global pour une spécification.

Le langage de programmation considéré est encore un langage trop éloigné d'un langage de programmation usuel comme ML. Les programmes sont explicitement typés. Il faudrait pouvoir avoir un mécanisme d'inférence de types qui permette d'écrire des programmes non typés tout en gardant de façon interne la possibilité de typer ces termes. Cela permettrait de se rapprocher beaucoup plus d'un langage plus naturel.

Dans un aspect plus pratique, la mise en œuvre doit être améliorée: il faudrait enrichir le langage d'annotations (les annotations ne peuvent être placées qu'à certains endroits et pas d'autres, sans raison théorique), améliorer le traitement des erreurs, améliorer le langage de programmation (comme nous l'avons dit plus haut) et accepter un plus grand nombre de définitions sous la forme ProPre.

Enfin, dans le cadre de l'interface Centaur [BCD+88], il devrait être possible d'envisager de visualiser le programme extrait d'une preuve et utiliser la tactique pour que la modification de l'arbre de programme partiel modifie la structure de l'arbre de preuve. Ainsi, on pourrait par exemple imaginer transformer une preuve qui produit un programme inefficace en une preuve qui produit un programme plus efficace. Mais, il s'agit là non pas d'une conséquence directe de ce travail mais d'une idée à plus long terme.

Annexe A

Une implantation : la tactique Program

La méthode heuristique qui a été décrite dans cette thèse a été implémentée dans le système *Coq* [DFH+93] comme une tactique. Cette annexe décrit le fonctionnement pratique de cette implantation pour l'utilisateur *Coq*. Elle demande une certaine connaissance préalable du système. Pour cela, on peut se référer à [DFH+93] ou à la documentation de la version 5.10 du système *Coq* [CCF+94]. Les exemples présentés dans cette annexe peuvent être rejoués dans la version 5.10 du système.

Cette annexe se divise en trois parties. La première partie présente les principes généraux d'utilisation de la tactique. En effet, la synthèse de preuves à partir de programmes se fait à l'aide de plusieurs tactiques. Chacune des tactiques disponibles est décrite par sa syntaxe et sa sémantique. La deuxième partie décrit plus précisément le langage des programmes annotés que la tactique prend en compte. Elle donne non seulement la syntaxe pour les annotations elles-mêmes mais aussi des syntaxes qui permettent une écriture plus proche d'un langage à la CAML. Enfin, la dernière partie liste le développement d'un certain nombre d'exemples simples pour illustrer différents aspects de la tactique comme l'utilisation de programmes récursifs ou l'utilisation d'annotations.

A.1 Syntaxe des tactiques

L'utilisateur doit fournir deux éléments : la spécification (sous la forme d'un but de façon usuelle) et le programme (voir A.2). Le programme est alors associé au but et l'utilisateur peut utiliser différentes tactiques pour développer une preuve automatique.

Realizer

Tout d'abord, le programme est associé au but courant en utilisant la commande `Realizer`. Avec cette commande, le programme doit être donné dans une syntaxe précise décrite dans la section A.2. Il est alors associé au but courant s'il est cohérent avec celui-ci.

Show Program

La commande `Show Program` affiche le programme associé au but courant. `Show Program n` affiche le programme associé au n-ième sous-but.

Program

Une synthèse de preuve automatique peut alors être déclenchée. Le programme associé au but courant par l'utilisateur est déroulé suivant la méthode heuristique décrite au chapitre 3 et les sous-programmes sont associés aux sous-buts par la tactique `Program` elle-même. Si aucun programme n'est associé au but courant, la tactique `Program` s'arrête. Cette tactique engendre une séquence de tactiques `Intro`, `Apply` ou `Elim` suivant l'analyse de la syntaxe du programme en suivant la méthode heuristique. Par exemple, si le programme commence par une λ-abstraction, la tactique `Intro` est engendrée, éventuellement plusieurs fois si des introductions logiques préalables sont nécessaires.

La tactique `Program` engendre une liste de sous-buts qui peuvent être soit informatifs soit logiques. Des sous-programmes sont associés aux sous-buts informatifs. Les sous-buts logiques doivent être résolus par l'utilisateur.

Program_all

La tactique `Program_all` est équivalente à la composition : `Repeat (Program OrElse Auto)`. Elle répète la tactique `Program` sur chacun des sous-buts informatifs et essaie la tactique `Auto` sur les sous-buts logiques. Noter que le travail de la tactique `Program` est considéré terminé quand tous les sous-buts informatifs ont été résolus. Ceci signifie que les lemmes logiques restant éventuellement doivent être résolus manuellement par l'utilisateur.

Program_Expand

La tactique `Program_Expand` transforme le programme courant en le même programme dans lequel la constante de tête est expansée. Cette tactique permet d'expanser la constante de tête du programme courant. Ceci peut être utile pour forcer l'utilisation du contenu d'une constante.

A.2 Syntaxe des programmes

A.2.1 Programmes purs

Le langage pour exprimer les programmes est appelé Real[1]. Les programmes sont explicitement typés[2] et proches des termes extraits de preuves. Des expressions ont été ajoutées pour permettre une syntaxe plus agréable.

Cette forme de base constitue l'ensemble des programmes purs. En réalité, cette forme de programmes n'est pas suffisante comme nous l'avons expliqué dans le chapitre 3. En effet, il peut être nécessaire de rajouter des informations logiques dans le programme pour pouvoir engendrer la preuve automatique. Des programmes annotés sont donc introduits.

A.2.2 Programmes annotés

La notion d'annotation introduit dans un programme une assertion logique qui peut être utilisée dans la preuve. Le but de la tactique `Program` est d'engendrer, à partir d'une spécification et d'un programme, des sous-buts soit logiques soit associés à d'autres programmes. Cependant, trouver la bonne spécification pour un sous-programme n'est pas une chose simple. Par exemple, il est difficile de retrouver un invariant de boucle ou un ordre bien fondé dans un appel récursif.

1. Il correspond à F_ω plus définitions inductives.

2. Cette information n'est pas strictement nécessaire mais elle permet une détection d'erreur plus simple dans un premier temps.

Des annotations peuvent donc être ajoutées dans le programme (voir chapitre 3) et représentent des parties logiques nécessaires à la preuve qui ne peuvent être retrouvées automatiquement. Cela permet au système de prouver des programmes qu'il ne pourrait prouver autrement.

Pour cela, une syntaxe particulière est introduite : puisque les annotations sont des spécifications, elles suivent la même syntaxe que les termes usuels de *Coq*. Pour les différencier à l'intérieur d'un programme, elles sont indiquées entre { et } et précédées de :: ::.

Comme les annotations sont des termes *Coq*, elles peuvent contenir des abstractions sur des propositions logiques qui doivent être déclarées. Ceci nécessite donc l'introduction de λ-abstractions d'annotations qui sont notées entre [{ et }]. Les λ-abstractions d'annotations peuvent être vues comme des λ-abstractions usuelles ne concernant que les annotations et pas les programmes.

A.2.3 Programmes récursifs

Des programmes peuvent être définis récursivement en utilisant la syntaxe : <*type-du-résultat* > rec *nom-de-l'hypothèse-de-récurrence* :: :: { *ordre-bien-fondé-de-récursion* } et le corps du programme (voir A.3) qui doit toujours commencer par une abstraction [x:A] où A est le type des arguments de la fonction (et aussi sur lequel porte la relation d'ordre).

A.2.4 Abréviations

Deux abréviations ont été définies :

<P>let (p:X;q:Y)=Q in S est un sucre syntaxique pour <P>Match Q with [p:X][q:Y]S

et

<P>if B then Q else R est une abréviation pour l'élimination sur les expressions booléennes, c'est-à-dire une abréviation pour <P>Match B with Q R où B : *bool*.

A.2.5 Grammaire

La grammaire pour les programmes est la suivante :

pg::= identificateur
  | [x:pg]pg
  | (x:pg)pg
  | pg->pg
  | (pg pg ... pg)
  | (<pg>Match pg with pg-list end)
  | (<pg>Case pg of pg-list end)
  | pg :: :: { coqterm }
  | [{x:coqterm}]pg
  | <pg>let ($x_1^1,\ldots,x_1^{k_1}:A_1;\ldots;x_n^1,\ldots,x_n^{k_n}:A_n$) = pg in pg
  | <pg>if pg then pg else pg
  | <pg>rec identifier :: :: { terme *Coq* } [x:pg]pg

La référence à un identificateur du contexte *Coq* à l'intérieur d'un programme du langage Real est une référence à son contenu extrait.

A.3 Exemples

Les exemples fournis dans ce paragraphe peuvent être rejoués dans la version V5.10 du système *Coq*.

A.3.1 Fonction d'Ackermann

Nous reprenons un exemple déjà vu dans le chapitre 4. Donnons la spécification de la fonction d'Ackermann. Nous voulons prouver pour tout n et m, qu'il existe un p tel que $ack(n,m) = p$ avec :

$$\begin{aligned} ack(0,n) &= n+1 \\ ack(n+1,0) &= ack(n,1) \\ ack(n+1,m+1) &= ack(n, ack(n+1,m)) \end{aligned}$$

Un programme ML pour cette spécification peut être :

```
let rec ack = function
        0  -> (function m -> Sm)
      | Sn -> (function  0  -> ack n 1
                       | Sm -> ack n (ack Sn m))
```

Supposons que nous donnions cette définition comme un prédicat ternaire `(Ack n m p)` grâce à une définition inductive représentant $p = ack(n,m)$:

```
Coq < Inductive Ack : nat->nat->nat->Prop :=
     Ack0 : (n:nat)(Ack O n (S n))
   | Ackn0 : (n,p:nat)(Ack n (S O) p)->(Ack (S n) O p)
   | AckSS : (n,m,p,q:nat)(Ack (S n) m q)->(Ack n q p)->(Ack (S n) (S m) p).
```

Alors, le but est de prouver $\forall n, m. \exists p. (Ack\ n\ m\ p)$, et la spécification est donc :

`(n,m:nat){p:nat|(Ack n m p)}`. Le programme `Real` associé correspondant au programme ML ci-dessus est :

```
Coq < Fixpoint ack_func [n:nat] : nat -> nat :=
  <nat->nat>Case n of
     (* O *) [m:nat](S m)
 (* (S n) *) [n':nat]
                Fix ack_func2 {ack_func2/1 : nat -> nat :=
                    [m:nat]<nat>Case m of
              (* O *) (ack_func n' (S O))
            (* S m *) [m':nat](ack_func n' (ack_func2 m'))
                     end}
    end.
```

Le programme est associé en utilisant `Realizer ack_func`. Alors, en répétant la tactique `Program`, trois lemmes logiques sont engendrés et peuvent être résolus simplement en utilisant les propriétés Ack0, Ackn0 et AckSS.

```
3 subgoals
  n : nat
  m : nat
  ============================
   (Ack 0 m (S m))
subgoal 2 is:
 (Ack (S n') 0 n0)
subgoal 3 is:
 (Ack (S n') (S m') n0)
```

A.3.2 Division Euclidienne

Cet exemple montre l'utilisation de programmes récursifs. Donnons la spécification de l'algorithme de la division euclidienne. Nous voulons montrer que, pour a et b ($b > 0$), il existe q et r tels que $a = b * q + r$ et $b > r$.

Un programme ML suivant cette spécification peut être :

```
let div b a = divrec a where rec divrec a =
        if (b<=a) then let (q,r) = divrec (a-b) in (Sq,r)
                  else (0,a)
```

Supposons qu'on utilise une définition inductive pour coder la propriété qui doit être prouvée, c'est-à-dire $\exists q \exists r.\ (a = b * q + r \wedge\ b > r)$:

```
Coq < Inductive diveucl [a,b:nat] : Set
        := divex : (q,r:nat)(a=(plus (mult q b) r))->(gt b r)->(diveucl a b).
```

La décidabilité de la relation d'ordre doit être prouvée au préalable, en donnant la fonction associée de type `nat->nat->bool` :

```
Coq < Theorem le_gt_dec : (n,m:nat){(le n m)}+{(gt n m)}.
Coq < Realizer [n:nat]<nat->bool> Match n with
                      (* 0 *) [m:nat]true
                      (* S *) [n':nat][H:nat->bool][m:nat]
                              <bool> Case m of
                               (* 0 *) false
                               (* S *) [m':nat](H m') end end.
Coq < Program_all.
Coq < Save.
```

Alors, la spécification est `(b:nat)(gt b 0)->(a:nat)(diveucl a b)`. Le programme associé correspondant au programme ML ci-dessus est :

```
Coq < Realizer
      [b:nat](<nat*nat>rec div :: :: { lt }
              [a:nat]<nat*nat>if (le_gt_dec b a)
                        then (* le b a *)
                             <nat*nat>let (q,r:nat) = (div (minus a b))
                                       in <nat,nat>((S q),r)
                        else (* gt b a *) <nat,nat>(0,a)).
```

Où `lt` est la relation d'ordre bien fondé sous-jacente définie par :

```
Coq < Definition lt := [n,m:nat](gt m n).
```

Noter que la syntaxe des programmes récursifs a été expliquée plus haut. La construction `rec` nécessite quatre arguments: le type du résultat de la fonction (`nat*nat` car elle retourne deux entiers naturels) entre `<` et `>`, le nom de l'hypothèse de récurrence (qui peut être utilisé pour des appels récursifs), la relation d'ordre `lt` (en tant qu'annotation car c'est une spécification), et le programme lui-même qui doit commencer avec une λ-abstraction. La spécification de `le_gt_dec` est connue car c'est un lemme préalable.

Le terme `(le_gt_dec b a)` est vu par la tactique `Program` comme un terme de type `bool` qui satisfait la spécification `{(le a b)}+{(gt a b)}`. Les tactiques `Program_all` ou `Program` peuvent être utilisées et les lemmes logiques suivants sont obtenus :

```
6 subgoals
  b : nat
  H : (gt b O)
  a : nat
  ============================
   (well_founded nat lt)
subgoal 2 is:
 (lt (minus a0 b) a0)
subgoal 3 is:
 a0=(plus (mult (S q) b) r)
subgoal 4 is:
 (gt b r)
subgoal 5 is:
 a0=(plus (mult O b) a0)
subgoal 6 is:
 (gt b a0)
```

Les sous-buts 4, 5 et 6 sont résolus par `Auto` (si `Program_all` est utilisé, ils n'apparaissent pas puisque `Program_all` essaie `Auto`). Les autres doivent être résolus par l'utilisateur.

A.3.3 Tri par insertion

Cet exemple montre l'utilisation des annotations. Donnons la spécification d'un algorithme de tri. Nous voulons montrer que, pour une liste triée d'entiers naturels l et un entier naturel a, nous pouvons construire une autre liste triée l', contenant tous les éléments de l plus a.

Un programme ML implémentant l'algorithme de tri par insertion et suivant cette spécification peut être :

```
let sort a l = sortrec l where rec sortrec = function
        []    -> [a]
      | b::l' -> if a<b then a::b::l' else b::(sortrec l')
```

Supposons que nous donnions les définitions *Coq* suivantes.

Tout d'abord, la décidabilité de la relation d'ordre.

```
Coq < Definition inf_dec: nat->nat->bool :=
        [n:nat]<nat->bool> Match n with
                [m:nat] true
                [n':nat][H:nat->bool][m:nat]
                        <bool> Case m of
                               false
                               [m':nat](H m') end end.
```

La définition du type list :

```
Coq < Inductive list : Set := nil : list | cons : nat -> list -> list.
```

Nous définissons la propriété pour un élément x d'être dans une liste l comme la plus petite relation telle que : $\forall a \forall l\ (In\ x\ l) \Rightarrow (In\ x\ (a :: l))$ et $\forall l\ (In\ x\ (x :: l))$.

```
Coq < Inductive In [x:nat] : list->Prop
        := Inl  : (a:nat)(l:list)(In x l) -> (In x (cons a l))
        | Ineq : (l:list)(In x (cons x l)).
```

Une liste t' est équivalente à une liste t avec un élément supplémentaire y si et seulement si : $(\forall x\ (In\ x\ t) \Rightarrow (In\ x\ t'))$ et $(In\ y\ t')$ et $\forall x\ (In\ x\ t') \Rightarrow ((In\ x\ t) \vee y = x)$. La définition suivante implémente cette conjonction ternaire.

```
Coq < Inductive equiv [y:nat;t,t':list]: Prop :=
      equiv_cons :
        ((x:nat)(In x t)->(In x t'))
        -> (In y t')
        ->((x:nat)(In x t')->((In x t)\/<nat>y=x))
        -> (equiv y t t').
```

Définition de la propriété pour une liste d'être triée, définie inductivement :

```
Coq < Inductive sorted : list->Prop
        := sorted_nil  : (sorted nil)
        | sorted_trans : (a:nat)(sorted (cons a nil))
        | sorted_cons : (a,b:nat)(l:list)(sorted (cons b l)) -> (le a b)
                        -> (sorted (cons a (cons b l))).
```

Alors, la spécification est:
`(a:nat)(l:list)(sorted l)->{l':list|(equiv a l l')&(sorted l')}`.

Le programme Real correspondant au programme ML est :

```
Coq < Realizer
        [a:nat][l:list]
         <list>Match  l with
          (cons a nil)
          [b:nat][m:list][H:list]
          <list>if (inf_dec b a) :: :: { {(le b a)}+{(gt b a)} }
                then (cons b H)
                else (cons a (cons b m))
         end.
```

Noter que inf_dec a été défini comme un programme réalisant la décidabilité de la relation d'ordre sur les entiers naturels. Mais il n'a aucune spécification, donc une annotation est nécessaire pour donner sa spécification. Cette spécification est utilisée pour décider de la décidabilité de la relation d'ordre sur les entiers naturels et celle-ci doit ensuite être montrée en utilisant le programme inf_dec.

Supposons que Program_all est utilisé. Un certain nombre de lemmes logiques est engendré :

```
8 subgoals
  a : nat
  l : list
  H : (sorted nil)
  ============================
   (equiv a nil (cons a nil))
subgoal 2 is:
 (sorted (cons a nil))
subgoal 3 is:
 (sorted (cons n' l1))
subgoal 4 is:
 (sorted l1)
subgoal 5 is:
 (equiv a (cons n l1) (cons n x))
subgoal 6 is:
 (sorted (cons n x))
subgoal 7 is:
 (equiv a (cons n l1) (cons a (cons n l1)))
subgoal 8 is:
 (sorted (cons a (cons n l1)))
```

A.3.4 Tri rapide

Cet exemple montre l'utilisation de programmes utilisant des programmes précédemment prouvés. Donnons la spécification du tri rapide. Nous voulons montrer que pour une liste d'entiers naturels l, nous pouvons construire une liste l' triée, qui soit une permutation de la précédente.

Un programme ML suivant cette spécification peut être :

```
let rec quicksort l = function
   []   -> []
 | a::m -> let (l1,l2) = partition a m in
                 let m1 = quicksort l1 and
                 let m2 = quicksort l2 in m1@[a]@m2
```

Où partition est définie par :

```
let rec partition a l = function
        []   -> ([],[])
      | b::m -> let (l1,l2) = partition a m in
                 if a<b then (l1,b::l2)
                        else (b::l1,l2)
```

Supposons qu'on donne les définitions *Coq* suivantes.

Déclaration de la relation d'ordre.

```
Coq < Variable   inf : A -> A -> Prop.
Coq < Definition sup  := [x,y:A]~(inf x y).
Coq < Hypothesis inf_sup : (x,y:A){(inf x y)}+{(sup x y)}.
```

Définition de la concaténation de deux listes :

```
Fixpoint app [l:list] : list -> list
      := [m:list]<list>Case l of
               (* nil *) m
          (* cons a l1 *) [a:A][l1:list](cons a (app l1 m)) end.
```

Définition de la permutation de deux listes.

```
Coq < 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) (mil a m n))
   |permut_milc : (a:A)(l,m,n:list)
         (permut (app m n) l)->(permut (mil a m n) (cons a l)).
```

Les définitions `inf_list` et `sup_list` permettent de savoir si un élément est plus grand ou plus petit que tous les autres éléments d'une liste :

```
Coq < Variable R : A->Prop.
Coq < Inductive Rlist  : list -> Prop :=
    Rnil : (Rlist nil)
   |Rcons : (x:A)(l:list)(R x)->(Rlist l)->(Rlist (cons x l)).
Coq < Definition inf_list := [x:A](Rlist (inf x)).
Coq < Definition sup_list := [x:A](Rlist (sup x)).
```

Définition de la propriété d'une liste d'être triée.

```
Coq < 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 (mil a l m)).
```

Alors, le but à prouver est $\forall l \; \exists m \; (sort \; m) \wedge (permut \; l \; m)$ et la spécification est `(l:list){m:list|(sort m)&(permut l m)}`. Le programme Real correspondant au programme ML est :

```
Coq < Realizer
     <list>rec quick :: :: { ltl }
           [l:list]
           <list>Case l of
            (* nil *)  nil
           (* cons *)  [a:A][m:list][t:list]
                       <list>let (l1,l2:list) = (Partition a m) in
                       (mil a (quick l1) (quick l2))
           end.
```

Où `ltl` est la relation d'ordre bien fondée définie par :

```
Definition ltl := [l,m:list](gt (length m) (length l)).
```

Et `Partition_spec` est la définition correspondant à $\exists l_1 \exists l_2.\ (sup\_list\ a\ l_1) \wedge (inf\_list\ a\ l_2) \wedge (l \equiv l_1 @ l_2) \wedge (ltl\ l_1\ (a :: l)) \wedge (ltl\ l2\ (a :: l))$:

```
Coq < Inductive Partition_spec [a:A; l:list] : Set :=
      Split_intro : (l1,l2:list)(sup_list a l1)->(inf_list a l2)
                    ->(permut l (app l1 l2))
                    ->(ltl l1 (cons a l))->(ltl l2 (cons a l))
                    ->(Partition_spec a l).

Coq < Theorem Partition : (a:A)(l:list)(Partition_spec a l).
Coq < Realizer [a:A][l:list]
    <list*list> Match l with
    (* nil *) <list,list>(nil,nil)
  (* cons  *) [b:A][m:list][ll:list*list]
               <list*list>let (l1,l2:list) = ll in
                  <list*list>if (inf_sup a b)
                             then (* inf a b *) <list,list>(l1,(cons b l2))
                             else (* sup a b *) <list,list>((cons b l1),l2)
    end.
Program_all.
Simpl; Auto.
Save.
```

Alors, `Program_all` donne les lemmes logiques suivants (qui doivent être résolus par l'utilisateur) :

```
3 subgoals
  l : list
  ============================
   (well_founded list ltl)
subgoal 2 is:
 (sort (mil a0 x0 x))
subgoal 3 is:
 (permut (cons a0 m) (mil a0 x0 x))
```

Annexe B

Un exemple complet de division

Dans cette annexe, nous allons développer un exemple complet, pas à pas. Il s'agit d'un algorithme de division euclidienne qui a déjà été brièvement présenté dans l'annexe précédente. Ici, bien qu'utilisant la notation *Coq* pour illustrer les différentes étapes du développement, nous souhaitons insister sur l'illustration de la méthode sous-jacente.

Chaque étape de la preuve va donc être détaillée en deux temps. Dans un premier temps, une explication de la méthode ou de l'heuristique utilisée (c'est-à-dire quel terme de preuve est reconstruit et comment). Dans un second temps, la présentation du résultat de l'étape dans le système *Coq*.

La spécification d'un algorithme de division est la suivante : pour a et b ($b > 0$), il existe q et r tels que $a = b * q + r$ et $b > r$. Une telle propriété peut être codée en *Coq* par une définition inductive :

```
Coq < Inductive diveucl [a,b:nat] : Set
      := divex : (q,r:nat)(a=(plus (mult q b) r))->(gt b r)->(diveucl a b).
```

Le but est alors : $\forall b.(b > 0).\forall a.\exists q \exists r.\ (a = b * q + r \wedge\ b > r)$ qui se code en *Coq* par le but suivant, si *gt* est l'ordre décroissant sur les entiers naturels :

```
Coq < Theorem eucl_dev : (b:nat)(gt b 0)->(a:nat)(diveucl a b).
1 subgoal

  ============================
   (b:nat)(gt b 0)->(a:nat)(diveucl a b)
```

Un programme ML cohérent avec cette spécification peut être :

```
let div b a = divrec a where rec divrec a =
        if (b<=a) then let (q,r) = divrec (a-b) in (Sq,r)
                  else (0,a)
```

Ce programme peut s'écrire dans le langage Real de *Coq*, où *lt* est l'ordre croissant sur les entiers naturels qui est sous-jacent à l'algorithme, *le_gt_dec* la décidabilité de l'ordre sur les entiers naturels et *minus* la différence entre deux entiers naturels :

```
eucl_dev < Realizer [b:nat]
          (<nat*nat>rec div :: :: { lt }
              [a:nat]<nat*nat>if (le_gt_dec b a) then
                        <nat*nat>let (q,r:nat) = (div (minus a b)) in
                                <nat,nat>((S q),r)
                         else <nat,nat>(0,a)).
eucl_dev <
1 subgoal
  ============================
   (b:nat)(gt b 0)->(a:nat)(diveucl a b)
```

Essayons maintenant d'appliquer pas à pas la méthode heuristique. Le programme est une abstraction sur b de type nat. La spécification est un produit sur b de type nat. La preuve est donc une abstraction sur b de type nat.

Au niveau interactif, b est introduit dans le contexte et le nouveau programme associé est le programme initial sans l'abstraction sur b.

```
eucl_dev < Program.
1 subgoal
  b : nat
  ============================
   (gt b 0)->(a:nat)(diveucl a b)

eucl_dev < Show Program.
[Realizer (<nat*nat>rec div :: :: { lt }
            [a:nat]<nat*nat>if le_gt_dec b a then
                 <nat*nat>let (q,r:nat) = (div (minus a b)) in
                          <nat,nat>((S q),r)
                 else <nat,nat>(0,a))]
```

Nous avons maintenant un programme récursif. En référence au chapitre 4, un programme récursif n'est qu'une application d'un principe de récurrence bien fondée `well_founded_induction` de type $(A : Set)(R : A \to A \to Prop)(well-founded\ A\ R) \to (P : A \to Set)(x : A)((y : A)(R\ y\ x) \to (P\ y)) \to (P\ x) \to (a : A)(P\ a)$. La syntaxe ci-dessus n'est donc qu'une macro pour :

```
(well_founded_induction nat :: :: { lt } nat*nat
            [a:nat][div:nat->nat*nat]
                 <nat*nat>if le_gt_dec b a then
                  <nat*nat>let (q,r:nat) = (div (minus a b)) in
                           <nat,nat>((S q),r)
                 else <nat,nat>(0,a))
```

Dans un premier temps, la spécification contient un produit logique en tête qu'il faut introduire comme hypothèse dans le contexte.

Dans un deuxième temps, le type extrait de `well_founded_induction` est donc $(A : Set)(P : Set)((x : A)((A \to P) \to P)) \to A \to P$. Or, le nombre de produits de ce type et le nombre d'arguments informatifs de `well_founded_induction` dans le programme ne sont pas les mêmes. Le nombre de produits de ce type est 4 alors que le nombre d'arguments informatifs de `well_founded_induction` dans le programme est 3. Une heuristique dans un tel cas est de rajouter un argument à l'application. Donc, une variable a de type A est introduite et le programme pg est transformé en $(pg\ a)$. C'est la η-expansion du programme : pg est transformé en $[a : A](pg\ a)$ et de même au niveau de la preuve.

On a donc la situation suivante :

```
eucl_dev < Program.
1 subgoal
  b : nat
  H : (gt b O)
  a : nat
  ============================
   (diveucl a b)

eucl_dev < Show Program.
[Realizer (well_founded_induction nat :: :: { lt } nat*nat
            [a0:nat][div:nat->nat*nat]
              <nat*nat>if (le_gt_dec b a0) then
                 <nat*nat>let (q,r:nat) = (div (minus a0 b)) then
                          <nat,nat>((S q),r)
                 else <nat,nat>(O,a0)
            a)]
```

Nous avons donc une application dont il faut retrouver les bouts logiques. Regardons argument par argument. Nous avons le type de preuve de `well_founded_induction` qui est $(A : Set)(R : A \rightarrow A \rightarrow Prop)(well - founded\ A\ R) \rightarrow (P : A \rightarrow Set)(x : A)((y : A)(R\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x) \rightarrow (a : A)(P\ a)$.

Le premier argument nat est tel quel un argument de preuve du bon type à savoir Set. Il correspond à A.

Le deuxième argument lt fourni comme un argument logique est un argument de preuve correspondant à l'ordre R.

Au niveau preuve, nous attendons ensuite un argument de type $(well - founded\ nat\ lt)$. Cet argument n'apparaît naturellement pas dans notre programme et n'est pas retrouvable par unification (il s'agit d'une preuve logique donc d'un terme logique de niveau 2). Au niveau de la preuve, nous laissons un *trou* à cet emplacement qui correspond à un lemme logique que l'utilisateur devra résoudre.

L'argument suivant est de type $A \rightarrow Set$. Il s'agit du prédicat de récursivité. Pour le retrouver, nous devons faire une unification entre la tête du type de preuve et la spécification. Comme type de preuve, nous avons : $(P : nat \rightarrow Set)(x : nat)((y : nat)(lt\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x) \rightarrow (a : nat)(P\ a)$. La tête est donc $(P\ a)$. La spécification est $(diveucl\ a\ b)$. Par unification, on obtient $P \equiv (x : nat)(diveucl\ x\ b)$.

L'argument suivant ne peut être retrouvé donc va engendrer un sous-but informatif $(x : nat)((y : nat)(lt\ y\ x) \rightarrow (diveucl\ y\ b)) \rightarrow (diveucl\ x\ b)$ associé au programme représenté par le corps de la récursion.

Enfin, le dernier argument a est tel quel un argument de preuve de type nat. La situation est la suivante :

```
eucl_dev < Program.
2 subgoals
  b : nat
  H : (gt b O)
  a : nat
  ============================
   (well_founded nat lt)
subgoal 2 is:
 (x:nat)((y:nat)(lt y x)->(diveucl y b))->(diveucl x b)
eucl_dev < Show Program.
[]
```

Le premier sous-but est logique comme nous l'avions annoncé. Il est résolu par l'utilisateur en l'occurrence ici par un lemme pour lequel nous ne détaillerons rien.

```
eucl_dev < Exact lt_wf.
1 subgoal
  b : nat
  H : (gt b O)
  a : nat
  ============================
   (x:nat)((y:nat)(lt y x)->(diveucl y b))->(diveucl x b)

eucl_dev < Show Program.
[Realizer [a:nat][div:nat->nat*nat]
              <nat*nat>if (le_gt_dec b a) then
                 <nat*nat>let (q,r:nat) = (div (minus a b)) then
                          <nat,nat>((S q),r)
                 else <nat,nat>(O,a)]
```

Les deux étapes suivantes sont très simples car le programme est une suite d'abstractions qui ne posent aucun problème car correspondent exactement à la même chose au niveau de la preuve.

```
eucl_dev < Program.
1 subgoal
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  ============================
   ((y:nat)(lt y a0)->(diveucl y b))->(diveucl a0 b)

eucl_dev < Show Program.
[Realizer [div:nat->nat*nat]
              <nat*nat>if (le_gt_dec b a0) then
                 <nat*nat>let (q,r:nat) = (div (minus a0 b)) then
                          <nat,nat>((S q),r)
                 else <nat,nat>(O,a0)]
```

```
eucl_dev < Program.
1 subgoal
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  ============================
   (diveucl a0 b)

eucl_dev < Show Program.
[Realizer <nat*nat>if (le_gt_dec b a0) then
                 <nat*nat>let (q,r:nat) = (div (minus a0 b)) then
                          <nat,nat>((S q),r)
                 else <nat,nat>(O,a0)]
```

Nous avons maintenant une élimination sur `(le_gt_dec b x)`. Ce terme a pour terme de preuve lui-même avec `le_gt_dec` de type de preuve $(n, m : nat)(le\ n\ m) + (gt\ n\ m)$. Le prédicat d'élimination peut être trouvé par unification. Il s'agit de $[f : \{(le\ b\ a0)\} + \{(gt\ b\ a0)\}](diveucl\ a0\ b)$. On engendre deux sous-buts avec deux sous-programmes associés.

```
eucl_dev < Program.
2 subgoals
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  ============================
   (le b a0)->(diveucl a0 b)
subgoal 2 is:
 (gt b a0)->(diveucl a0 b)

eucl_dev < Show Program.
[Realizer <nat*nat>let (q,r:nat) = (div (minus a0 b)) then
                          <nat,nat>((S q),r)]
```

Nous avons à nouveau un élimination cette fois sur `(H0 (minus x b))`. Le terme de preuve de ce terme ne peut être retrouvé que partiellement. En effet, il faut une preuve logique de $(lt\ (minus\ a0\ b)\ a0)$ correspondant au deuxième argument de div. Comme il s'agit d'une preuve logique, c'est un *trou* qui reste dans la preuve et que l'utilisateur devra résoudre. Le prédicat d'élimination peut être trouvé par unification. Il s'agit de $[f : (diveucl\ (minus\ a0\ b)\ b)](diveucl\ a0\ b)$. On engendre un sous-but avec un sous-programme associé.

```
eucl_dev < Program.
3 subgoals
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  l : (le b a0)
  ============================
    (lt (minus a0 b) a0)
subgoal 2 is:
 (q:nat)
  (r:nat)(minus a0 b)=(plus (mult q b) r)->(gt b r)->(diveucl a0 b)
subgoal 3 is:
 (gt b a0)->(diveucl a0 b)

eucl_dev < Show Program 2.
[Realizer [q,r:nat]<nat,nat>((S q),r)]
```

Nous passon sur le premier sous-but qui est logique et considérons dans la suite le second. Nous passons sur les deux étapes suivantes qui sont des étapes triviales d'introductions d'hypothèses correspondant directement à des abstractions.

```
eucl_dev < 2:Program.
3 subgoals
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  l : (le b a0)
  ============================
   (lt (minus a0 b) a0)
subgoal 2 is:
 (r:nat)(minus a0 b)=(plus (mult q b) r)->(gt b r)->(diveucl a0 b)
subgoal 3 is:
 (gt b a0)->(diveucl a0 b)

eucl_dev < Show Program 2.
[Realizer [r:nat]<nat,nat>((S q),r)]

eucl_dev < 2:Program.
3 subgoals
  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
```

```
  l : (le b a0)
  ============================
   (lt (minus a0 b) a0)
subgoal 2 is:
 (minus a0 b)=(plus (mult q b) r)->(gt b r)->(diveucl a0 b)
subgoal 3 is:
 (gt b a0)->(diveucl a0 b)

eucl_dev < Show Program 2.
[Realizer <nat,nat>((S q),r)]
```

Nous avons maintenant un cas un peu plus compliqué. Dans un premier temps, nous introduisons les hypothèses logiques en tête de la spécification à savoir $(minus\ a0\ b) = (plus\ (mult\ q\ b)\ r)$ et $(gt\ b\ r)$.

Le programme `<nat,nat>((S q),r)` est une macro pour `(Constr(1,prod) nat nat (S q) r)` où *prod* est le type inductif codant le produit de deux éléments paramétré par les types de ces éléments. La spécification est le type inductif *diveucl* appliqué à des arguments.

Les deux types *prod* et *diveucl* sont isomorphes. On prend donc comme type inductif de preuve le type *diveucl*. On construit le terme de preuve suivant $(Constr(1, diveucl)\ a0\ b\ (S\ q)\ r\ ?_1\ ?_2)$ où $?_1$ correspond à la preuve logique de $a0 = (plus\ (mult\ (S\ q)\ b)\ r)$ et $?_2$ correspond à la preuve logique de $(gt\ b\ r)$. Pratiquement, cela revient à donner explicitement $(S\ q)$ et r comme témoins et à laisser deux lemmes logiques sur ces témoins. On a donc la situation suivante :

```
eucl_dev < 2:Program.
4 subgoals

  b : nat
  H : (gt b O)
  a : nat
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  l : (le b a0)
  ============================
   (lt (minus a0 b) a0)

subgoal 2 is:
 a0=(plus (mult (S q) b) r)
subgoal 3 is:
 (gt b r)
subgoal 4 is:
 (gt b a0)->(diveucl a0 b)

eucl_dev < Show 2.
subgoal 2 is:

  b : nat
  H : (gt b O)
  a : nat
```

```
  a0 : nat
  div : (y:nat)(lt y a0)->(diveucl y b)
  l : (le b a0)
  q : nat
  r : nat
  e : (minus a0 b)=(plus (mult q b) r)
  g : (gt b r)
  ============================
   a0=(plus (mult (S q) b) r)

eucl_dev < Show Program 2.
[]
```

Les trois premiers sous-buts sont donc des sous-buts logiques que l'utilisateur devra résoudre. En l'occurrence, ce sont des lemmes très simples. Le dernier lemme correspond au deuxième cas de l'élimination sur `(le_gt_dec b a0)`. Il est exactement similaire au cas que nous venons de voir sur le constructeur.

Bibliographie

[And93] Anderson (P.). – *Program Derivation by Proof Transformation.* – Thèse de PhD, Carnegie Mellon University, octobre 1993.

[And94a] Anderson (P.). – Program extraction in a logical framework setting. *In : 5th International Conference on Logic Programming and Automated Reasoning.*

[And94b] Anderson (P.). – Representing Proof Transformation for Program Optimization. *In : CADE'94.*

[Bar91] Barendregt (H.). – *Lambda Calculi with Types.* – Rapport technique n° 91-19, Catholic University Nijmegen, septembre 1991.

[BCD+88] Borras (P.), Clement (D.), Despeyroux (T.), Incerpi (J.), Kahn (G.), Lang (B.) et Pascual (V.). – Centaur: the system. *In : 3rd Symposium on Software Development Environments.*

[Ber94] Berardi (S.). – "pruning" simply typed λ-terms. – 1994.

[BM92] Burstall (R.) et McKinna (J.). – *Deliverables : a categorical approach to program development in type theory.* – Rapport technique n° 92-242, LFCS, octobre 1992. Aussi dans [NPP92].

[CCF+94] Cornes (C.), Courant (J.), Filliâtre (J.C.), Huet (G.), Manoury (P.), Paulin-Mohring (C.), Muñoz (C.), Murthy (C.), Parent (C.), Saïbi (A.) et Werner (B.). – *Coq V5.10 Reference Manual.* – Rapport technique, INRIA, 1994. Disponible en ftp anonyme sur ftp.inria.fr, à paraître.

[CG91] Cooke (D.) et Gates (A.). – On the development of a method to synthesize programs from requirements specifications. *International Journal of Software Engineering and Knowledge Enginerring*, vol. 1, n° 1, 1991, pp. 21–38.

[Coh92] Cohen (M.D.). – *Interactive Program Derivation.* – Thèse de PhD, University of Cambridge, 1992.

[Con86] Constable et al (R. L.). – *Implementing Mathematics with the Nuprl Proof Development System.* – Prentice-Hall, 1986.

[Coq85] Coquand (T.). – *Une théorie des constructions.* – Thèse de PhD, Université Paris VII, 1985.

[Coq89] Coquand (T.). – Meta-mathematical investigations of a Calculus of Constructions. – 1989. Dans [Hue89].

[Coq91] Coquand (T.). – *An introduction to type theory*, pp. 117–137. – INRIA, 1991.

[DB80] De Bruijn (N.G.). – A survey of the project AUTOMATH. *In : To H.B.Curry : Essays on Combinatory Logic , lambda-calculus and formalism*, éd. par Hindley (J.R.). – Seldin, J.P., 1980.

[DFH+93] Dowek (G.), Felty (A.), Herbelin (H.), Huet (G.), Murthy (C.), Parent (C.), Paulin-Mohring (C.) et Werner (B.). – *The Coq Proof Assistant User's Guide - Version 5.8.* – Rapport technique n° 154, Projet Formel - INRIA-Rocquencourt-CNRS-ENS Lyon, mai 1993.

[Dow91] Dowek (G.). – *Démonstration Automatique dans le Calcul des Constructions.* – Thèse de PhD, Université Paris 7, 1991.

[Gal91] Gallier (J.). – *Constructive Logic. Part I: A tutorial on Proof Systems and Typed λ-Calculi.* – Rapport technique, Digital, mai 1991.

[Gir72] Girard (J.Y.). – *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* – Thèse de PhD, Université Paris 7, 1972.

[Goa80] Goad (C.A.). – *Computational uses of the manipulation of formal proofs.* – Thèse de PhD, Standford University, 1980.

[HN88] Hayashi (S.) et Nakano (H.). – *PX : A Computational Logic.* – MIT Press, 1988, *Foundations of Computing.*

[Hoa69] Hoare (C.A.R.). – An Axiomatic Basis for Computer Programming. *Communications of the ACM*, vol. 12, n° 10, octobre 1969.

[How80] Howard (W.A.). – The formulaes-as-types notion of construction. *In : To H.B.Curry : Essays on Combinatory Logic , lambda-calculus and formalism*, éd. par Hindley (J.R.). – Seldin, J.P., 1980.

[Hue89] Huet (G.). – *The Calculus of Constructions, Documentation and users's guide, Version 4.10.* – Rapport technique, INRIA, 1989.

[Hue91] Huet (G.). – *Introduction au λ-calcul typé*, pp. 137–163. – INRIA, 1991.

[Hue92] Huet (G.). – *Constructive Computation Theory.* – Notes de cours, octobre 1992.

[Kri90] Krivine (J.L.). – *Lambda-calcul, types et modèles.* – MASSON, 1990.

[Laf91] Lafont (Y.). – *Calcul des séquents et déduction naturelle*, pp. 99–117. – INRIA, 1991.

[Lal90] Lalement (R.). – *Logique, Réduction, Résolution.* – MASSON, 1990.

[Lec94] Leclerc (F.). – Termination proof of term rewriting system with the multiset path ordering and derivation length. A complete development in the calculus of constructions. – 1994. A paraître dans TLCA'95.

[LP92] Luo (Z.) et Pollack (R.). – *LEGO Proof Development System : User's Manual.* – Rapport technique n° 92-211, LFCS, mai 1992.

[Luo90] Luo (Z.). – *An Extended Calculus of Constructions.* – Thèse de PhD, Department of Computer Science, University of Edinburgh, juin 1990.

[ML84] Martin-Löf (P.). – *Intuitionistic Type Theory.* – Bibliopolis, 1984, *Studies in Proof Theory.*

[MR92] Manna (Z.) et R. (Waldinger). – Fundamentals of Deductive Program Synthesis. *IEEE Transactions on Software Engineering*, vol. 18, n° 8, août 1992.

[MS92] Manoury (P.) et Simonot (M.). – *Des preuves de totalité de fonctions comme synthèse de programmes.* – Thèse de PhD, Université PARIS 7, décembre 1992.

[MW71] Manna (Z.) et Waldinger (R.J.). – Toward automatic program synthetis. *Communication ACM*, vol. 14, n° 3, mars 1971.

[NPP92] Nordström (B.), Petersson (K.) et Plotkin (G.) (édité par). – *Prooceedings of the 1992 worshop on types for proofs and programs.* – juin 1992.

[PC89] Pfenning (F.) et C. (Paulin-Mohring). – Inductively Defined Types in the Calculus of Constructions. *In : 5th International Conference on Mathematical Foundations of Programming Semantics*, pp. 209–228.

[PM89a] Paulin-Mohring (C.). – Extracting F_ω's programs from proofs in the Calculus of Constructions. *In : Sixteenth Annual ACM Symposium on Principles of Programming Languages.* – Austin, janvier 1989.

[PM89b] Paulin-Mohring (C.). – *Extraction de programmes dans le Calcul des Constructions.* – Thèse de PhD, Université Paris VII, 1989.

[PM93] Paulin-Mohring (C.). – Inductive Definitions in the System Coq - Rules and Properties. *In : Typed Lambda Calculi and Applications.* – Aussi rapport de recherche 92-49, LIP-ENS Lyon, Décembre 1992.

[PMW92] Paulin-Mohring (C.) et Werner (B.). – Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation-special issue on automated programing*, 1992.

[Pol92] Poll (E.). – *A programming logic for $F\omega$.* – Rapport technique n° 92/25, Eindhoven University of Technology, septembre 1992.

[Pol94] Poll (E.). – *A Programming Logic Based on Type Theory.* – Thèse de PhD, Technische Universiteit Eindhoven, 1994.

[Pot88] Potet (M.). – *Preuves et stratégies pour la synthèse déductive de programmes.* – Thèse de PhD, Institut National Polytechnique de Grenoble, 1988.

[Tro73] Troelstra (A.S.). – *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* – Springer-Verlag, 1973, *LNM*, volume 344.

Index