THÈSE DE DOCTORAT

présentée

À L'UNIVERSITÉ PARIS 7 - DENIS DIDEROT

pour l'obtention du Diplôme de

DOCTEUR DE L'UNIVERSITÉ PARIS 7
SPÉCIALITÉ : INFORMATIQUE FONDAMENTALE

présentée et soutenue publiquement
par

Bruno BARRAS

le 17 Novembre 1999

Titre :

Auto-validation d'un système de preuves avec familles inductives

---

Directeurs de thèse :

MM. Gérard HUET
Benjamin WERNER

Jury :

MM.	Guy COUSINEAU	Président
	Stefano BERARDI	Rapporteurs
	Pierre-Louis CURIEN	
Mme.	Thérèse HARDIN	
MM.	Hubert COMON	Examinateurs
	Herman GEUVERS	

Je voudrais remercier tout particulièrement Benjamin Werner pour l'encadrement de très grande qualité, sa disponibilité de tous les instants, et l'attention toute particulière qu'il a porté à mon travail depuis mon arrivée à l'INRIA. Mes nombreuses discussions avec lui ont toujours rencontré compréhension et conseils. Son soutien constant surtout lors de l'étape la plus décourageante de la thèse, la rédaction, m'ont été une aide précieuse et irremplaçable. J'ai aussi apprécié ses grandes qualités humaines qui font qu'il restera plus qu'un directeur de thèse pour moi.

Je suis très reconnaissant envers Stefano Berardi et Thérèse Hardin d'avoir accepté d'être rapporteurs de ma thèse, avec tout le travail que cela implique. Pierre-Louis Curien m'a fait l'honneur de lire avec beaucoup d'attention mon manuscrit, et j'ai eu à cœur de suivre ses remarques pour l'améliorer, tant sur le contenu que sur la forme.

Le projet Coq regroupe un grand nombre de talents, et cela n'est sûrement pas étranger à la façon dont Gérard Huet a su le diriger pendant de nombreuses années. Ses remarques m'ont toujours paru d'une grande sagesse, et j'espère que cela sera un tant soit peu contagieux.

J'ai eu l'occasion de connaître Guy Cousineau pour avoir enseigné sous sa direction. J'en garde un très bon souvenir, c'est pourquoi je suis heureux qu'il préside le jury de ma thèse.

Les conférences du groupe de travail TYPES m'ont permis de faire la connaissance d'Herman Geuvers, et de découvrir une personne très sympathique. Ses travaux m'ont souvent servi de référence pour mon travail de thèse.

Je me rejouis de la présence d'Hubert Comon dans mon jury, en portant un regard nouveau sur mon travail.

Mon travail d'implantation pour Coq a toujours été reçu avec grand intérêt par Christine Paulin. Son éternelle gentillesse m'a été réconfortante.

Mes pensées vont aussi à mes parents, pour tous les sacrifices qu'ils ont faits sans hésitation, m'incitant à poursuivre mes études, tout en m'ayant laissé une totale liberté pour m'engager dans la voie qui m'intéressait le plus.

La thèse est un travail de longue haleine, qui déborde bien souvent des horaires de travail, et la vie pourrait ressembler fort à une succession de buts que l'on résout à l'aide de tactiques. Mes amis de toujours (ou presque) m'ont heureusement à chaque fois ramené sur terre. Un grand merci à Thierry, Noam, Caroline, Natalie, Henri et enfin Juju, toujours 'zent pour aller s'écorcher les oreilles.

L'assiduité au très solennel séminaire "Preuves de Programmes Fonctionnels en Théorie des Types et Gueuletons" a aussi été très enrichissante. J'admire le talent de ses principaux animateurs : Christian et ses positions toujours polémiques, François pour la qualité des lieux de réunion, Magali en modératrice zélée et Alexandre, avec qui nous avons refait tant de fois le monde.

Por fin, dedico esas ultimas líneas a todos mis amigos de lengua nativa española, aunque casi todos tambien hablan francés : mi negrita Paula, Eduardo, Cristina, César y Gilberto. La mala cualidad de mi español no me permite agradecirlos como lo merecen. Sólo puedo decir que conocerlos me encantó, y mucho más...

Table des matières

Chapitre 1

Introduction

La logique est une branche de mathématiques dont l'objectif est de décrire de manière aussi rigoureuse que possible ce qu'est une démonstration d'un théorème en mathématiques. Habituellement, une démonstration est un texte rédigé en langage naturel, dans lequel on exprime et combine des arguments visant à convaincre de la vérité de la proposition. Le problème est que le langage naturel est peu fiable car il est parfois ambigu. Un des principaux travaux du logicien est d'essayer de définir avec précision le langage employé par les mathématiciens, et de montrer qu'il n'est pas possible de déduire de choses absurdes. Les preuves écrites dans ces formalismes ont l'avantage de ne pas êtres ambiguës, et peuvent être crues sans risque.

Ainsi, le but n'est pas de développer des théories, mais plutôt de vérifier que tout ce qui a été démontré l'a été fait dans les règles. C'est pourquoi la logique a pu avoir l'image d'un domaine assez stérile, puisqu'il n'apporte pas vraiment de résultat nouveau, réservé aux personnes maniaques, voire paranoïaques, estimant que les mathématiciens ne sont pas assez rigoureux. La découverte de paradoxes dans les premiers systèmes logiques[1] n'a pas redoré le blason de la logique. Une fois ces premiers paradoxes résolus, la logique n'avait toujours que très peu d'applications, car on considérait les mathématiciens suffisamment intelligents et respectables pour qu'on puisse leur faire confiance.

La grande revanche des logiciens est peut-être venue avec le développement considérable de l'informatique. Les programmes étant de plus en plus répandus, et à des positions de plus en plus critiques (ferroviaire, avionique, chirurgie et banque) les erreurs de programmation qu'ils comportent risquent d'avoir des conséquences tragiques. On ressent le besoin de faire des vérifications pour s'assurer qu'ils ont certaines bonnes propriétés. Pour reprendre les domaines d'applications déjà mentionnés, on pourra chercher à prouver l'impossibilité de collisions entre trains, ou bien garantir la confidentialité des échanges entre banques, etc. L'idée est de traduire les propriétés des programmes en une formule logique (cette étape s'appelle la *formalisation*) que l'on va chercher à prouver. Si la formule se trouve être démontrable, alors on est assuré que le programme possède la propriété attendue, dans la mesure où la propriété a été encodée correctement[2]. L'ensemble de ces propriétés forme ce

[1] Dans la théories des ensembles de Cantor, on peut former un ensemble paradoxal : l'ensemble de tous les ensembles qui ne s'appartiennent pas. Le paradoxe de Russell est que l'on peut prouver à la fois que cet ensemble s'appartient, et qu'il ne s'appartient pas.

[2] Cette traduction sera d'autant plus facile que le système logique est *expressif* et simple. Les logiques dites d'*ordre supérieur* sont en général plus proches du langage mathématique usuel. L'étape de formalisation est plus facile, car elle ne nécessite pas d'encodage complexe du problème. Ainsi, il est plus simple de se convaincre que ce qui est prouvé dans le système formel est une image fidèle de ce qui se passe dans la réalité.

qu'on appelle la *spécification* du programme.

1.1 Deux approches pour la vérification de programmes

Rédiger toutes les preuves dans un système logique pose quelques difficultés. D'une part, la construction de preuves formelles est souvent longue et fastidieuse car il faut donner beaucoup de détails, ou alors on élude certains détails, et la preuve risque de devenir trop compliquée, ou sujette à de multiples interprétations. En plus, la quantité et la complexité des programmes écrits fait que cette tâche échappe rapidement aux capacités humaines et il devient nécessaire de déléguer tout ou partie de ces vérifications à l'ordinateur. D'autre part, l'art de la démonstration nécessite beaucoup d'intuition, chose dont l'ordinateur est dépourvu. Il y a alors deux grandes orientations possibles, et qui sont parfois ressenties comme des frères ennemis dans le monde des méthodes formelles.

On peut d'une part chercher à résoudre une classe de problèmes restreinte, mais représentant une grande part des problèmes qui se posent concrètement. Pour cette classe, il est possible d'automatiser la résolution, la stratégie étant essentiellement d'énumérer toutes les possibilités. C'est la base du *model-checking*. L'avantage majeur de cette méthode est sa très grande automatisation. L'utilisateur se contente de fournir le programme annoté par les propriétés attendues, et le *model-checker* répond si les propriétés sont satisfaites. En particulier, il n'a pas à avoir une grande expertise dans le développement de preuves. La contrepartie, c'est que le domaine d'application est limité et lorsque l'on sort de ce cadre, l'outil n'est plus d'aucune utilité[3]. Nous rappelons que l'idée de base est d'énumérer tous les cas. Or, le nombre de possibilités croît en général de manière exponentielle avec le nombre de variables du problème. Il est absolument vital de factoriser massivement des cas. Cela devient encore plus complexe lorsque le problème a un nombre infini de cas. Par exemple, la proposition "pour tout entier naturel n, $2n$ est pair" ne peut pas se vérifier de manière exhaustive car il y a un nombre infini d'entiers. Le domaine d'application du *model-checking* a été étendu aux cas où l'on sait se ramener à des cas finis en faisant des suppositions simplificatrices, que l'on appelle heuristiques[4]. Enfin, nous mentionnons deux autres inconvénients directement liés au principal avantage. Du fait de l'automatisation, si le *model-checker* informe l'utilisateur que son programme est correct, cela ne lui dit pas pour autant *pourquoi* il marche. Il n'est pas uniquement question de satisfaire la soif de comprendre le problème dans ses moindres détails (ce qui est généralement une caractéristique du chercheur), il s'agit d'un problème de fiabilité : lorsque le *model-checker* donne son feu vert, on est obligé de lui faire confiance. Or, ces systèmes étant en général assez complexes car faisant appel à de nombreuses astuces de programmation (la factorisation de cas étant cruciale), ils risquent de comporter des erreurs. La deuxième conséquence néfaste de l'automatisation est que certains problèmes peuvent avoir un très grand nombre de cas, mais une preuve courte, qui est en général celle que l'humain expérimenté trouvera intuitive. Il se peut que le temps passé à chercher une preuve "humaine" soit largement compensé par le fait de n'avoir qu'une petite preuve à vérifier, plutôt qu'un grand nombre de cas.

L'autre approche pour prouver des propriétés de programme sur ordinateur, est de concevoir un système permettant de développer les preuves de théorème d'une manière proche du langage mathématique habituel. Cela concerne autant l'expressivité du système logique

[3] Si la correction du programme fait appel à un théorème très complexe (par exemple le théorème de Fermat), aucun système actuel ne sera capable de le certifier automatiquement.

[4] Par exemple en introduisant comme principe la récurrence sur les entiers, qui permet de traiter des formules quantifiées sur des entiers en étudiant le cas de 0 et le cas $n+1$ en supposant la propriété vraie pour n.

employé que la manière dont on cherche la preuve. Nous rappelons qu'un système expressif a l'avantage de ne pas obliger l'utilisateur à encoder son problème pour qu'il entre dans son domaine d'application, étape qui peut donner lieu à des erreurs particulièrement subtiles. La tâche de l'ordinateur est de vérifier que l'utilisateur construit une preuve correcte du théorème à démontrer, grâce à un *vérificateur de preuves*. Il est aussi très important de fournir un ensemble de commandes qui permettent à l'utilisateur de dire à l'ordinateur comment se prouve le théorème. On parle alors de *système d'aide à la preuve*. En d'autres termes, on laisse à l'humain le soin de concevoir l'architecture de la preuve, et l'ordinateur se charge de chercher les preuves fastidieuses et simples.

Parmi les premiers systèmes de preuves dans lesquels des preuves conséquentes ont été mécanisées, on peut citer Automath et LCF. La plupart des systèmes actuels ont largement réutilisé les idées de ces pionniers. Devant l'actuelle prolifération des systèmes de preuves, nous ne citerons que quelques uns : NQTHM, PVS, HOL, Isabelle [54], et du côté de la Théorie des Types, Coq [8], Alf et Lego. Même si dans cette thèse nous nous intéressons principalement à l'architecture du système Coq, il se dégage des principes généraux pertinents pour les autres systèmes.

Une caractéristique importante de ces systèmes est qu'ils donnent une idée plus précise de ce qu'est une preuve, même si certains systèmes ne les manipulent pas explicitement. Cette notion de preuve apporte beaucoup à la fiabilité du système car cela permet facilement de séparer les étapes de construction et de validation d'une preuve. En général, il est très facile d'écrire un programme qui prend en entrée une preuve et qui répond si celle-ci est correcte. Ce programme est le composant essentiel de ce que l'on appelle le *noyau* du système. Autour de ce noyau, des fonctions plus ou moins complexes (que l'on appelle *tactiques* depuis LCF) ont pour but de chercher une preuve, puis de la construire si cela est nécessaire[5]. Le point clé est que la fiabilité ne repose pas sur ces fonctions, mais uniquement sur le noyau.

Les deux options (model-checking ou système de preuves) sont inconciliables du fait du théorème d'incomplétude de Gödel. Informellement, ce théorème montre l'impossibilité d'écrire un algorithme qui permette de trouver une preuve pour toute formule vraie dans un système logique suffisamment fort, comme par exemple les très répandues théories des ensembles Z ou ZF. Cependant, il n'est pas interdit d'espérer faire cohabiter ces deux aspects qui ont chacun remporté un certain succès. Une idée pragmatique serait d'avoir un système de preuves vu comme une espèce de "shell". On y développerait les parties faisant appel à l'intuition ou qui méritent d'être clairement expliquées. La stratégie dominante serait de décomposer le problème en sous-problèmes indépendants, et lorsque l'on reconnaît qu'un de ces sous-problèmes rentre dans le champ d'application d'un model-checker, le système de preuve pourrait lui donner la main.

Il reste la question de savoir si l'on veut quand même la preuve correspondant à ce qu'a prouvé le model-checker. En effet, celle-ci risque d'être grosse. Mais d'un autre côté, on voudrait éviter de faire trop appel à des outils externes, car cela rend la preuve de cohérence du système particulièrement ardue. Un bon compromis à ce dilemne semble être atteint par la technique relativement ancienne de "réflexion calculatoire" que nous évoquerons. Le principe est en quelque sorte de formaliser le model-checker dans le système de preuve, et de prouver la correction de ses algorithmes.

[5] Pour des raisons d'efficacité, il est important de distinguer la recherche et la construction de la preuve. En effet, lorsque l'on recherche une preuve, on arrive parfois dans une "impasse" et il faut explorer d'autres possibilités. Il est inutile de construire les preuves partielles correspondant à ces impasses.

1.2 Vérification de l'implantation

Une fois que l'on est décidé à travailler avec un système de preuves, on peut observer qu'il y a deux tâches distinctes : d'une part rechercher un système logique adéquat, c'est un problème fondamental. La qualité essentielle d'un système logique est qu'il ne permette de prouver que des propositions qui correspondent à ce que l'on considère comme vrai intuitivement (par exemple "tout nombre impair est premier" ne devra pas être prouvable, si l'on a défini correctement les notions de nombre impair ou premier). C'est un problème qui dépasse le cadre des mathématiques, et on ne s'en préoccupera pas (ou alors très peu) dans cette thèse. Une propriété plus objective est qu'il soit cohérent, c'est-à-dire qu'il ne permette pas de prouver une proposition appelée absurde.

Ce besoin absolu d'avoir un système cohérent rentre en concurrence avec une autre exigence : l'expressivité. Nous avons déjà évoqué l'importance de l'expressivité pour que le confort d'utilisation du système soit plus grand, ce qui avait comme effet bénéfique de réduire les risques d'erreurs au moment de la formalisation du problème. Le problème est qu'un système plus expressif permet de prouver plus de théorèmes, et donc le risque d'incohérence est plus grand.

Un système logique comme la théorie des ensembles est une sorte de langage polyvalent, non spécialisé pour l'étude de propriétés des programmes. Pour faire de la vérification de programmes dans de tels systèmes, on code généralement les algorithmes par une fonction. Le prérequis est donc de formaliser la sémantique d'un langage de programmation, qui permet d'associer une fonction à chaque programme. Mais une fonction n'est pas la même chose qu'un algorithme. Il existe des fonctions que l'on peut spécifier, mais qui n'admettent pas d'algorithme[6]. Seules les fonctions *calculables* admettent un algorithme. D'autre part, on ne veut pas forcément identifier deux programmes qui rendent le même résultat pour la même entrée. Il est clair pour toute personne ayant déjà programmé que la rapidité avec laquelle l'ordinateur fera le calcul dépend de l'algorithme employé. Nous nous attacherons plutôt à des systèmes qui considèrent la notion de calcul comme essentielle. Les systèmes de la *Théorie des Types* incluent en général un langage de programmation de manière primitive. Ce langage ressemble à un langage ML purement fonctionnel.

L'autre direction de recherche, qui est l'objet principal de cette thèse est de réaliser une implantation sûre (mais on prouve aussi les propriétés du système, en excluant la preuve de cohérence), si possible efficace, d'un vérificateur de preuves pour un système logique donné[7] que l'on supposera cohérent. Ce travail va donc nous permettre d'étudier les interactions entre les formalismes logiques et l'architecture des systèmes de preuves, i.e. comment le formalisme influence l'implantation (de manière évidente), mais aussi comment l'implantation peut avoir des conséquences sur la façon de présenter un système. En l'occurrence, nous formaliserons le Calcul des Constructions Inductives, qui est le système logique de Coq. Comme ce vérificateur de preuve est censé garantir la validité de preuves, on veut s'assurer qu'il est correct. Sinon il pourrait accepter une preuve de correction pour une proposition qui serait en fait fausse, le programme n'aurait alors peut-être pas les bonnes propriétés attendues, et provoquer ou laisser arriver un évènement désastreux.

Pour éviter ce scénario catastrophe, il semble primordial de vérifier l'implantation de ce vérificateur de preuves. Après tout, puisque l'on recommande de vérifier les programmes, autant commencer par développer le vérificateur comme on le ferait pour un programme quelconque. De cette manière, on ramène la vérification de n'importe quel programme à la vérification d'un seul : si l'on croit en la preuve de correction du vérificateur, toute preuve

[6] Par exemple l'arrêt des machines de Turing.

[7] En fait, on ne travaillera pas sur un système en particulier, mais sur des classes de systèmes, pour pouvoir corriger le tir si l'on découvre que le système auquel on pensait s'avère incohérent.

faite avec ce vérificateur pourra être considérée comme un théorème de la logique, que l'on a supposée cohérente.

Mais il se pose alors le problème de croire la preuve de correction du vérificateur. En effet, celle-ci a été formalisée dans un certain système logique, dont on doit admettre la cohérence. On remarquera que l'on rentre dans un cycle sans fin, où l'on peut toujours remettre en cause le système dans lequel est exprimée la preuve de correction du système précédent. Pis encore, le deuxième théorème d'incomplétude de Gödel nous dit que le système dans lequel est exprimée une preuve de cohérence a plus de chances d'être incohérent que le système dont on prouve la cohérence. Il nous faut donc abandonner l'idée de tout prouver.

Toutes ces remarques traduisent en fait une chose bien simple, c'est que même en mathématiques, on ne peut être infailliblement sûr de rien. Le mieux que l'on puisse espérer, c'est une *quasi*-certitude en accumulant des vérifications de nature différente. Plus on fera de vérifications, plus on augmentera la confiance que l'on peut porter au système.

Nous partons avec l'hypothèse que le système que nous souhaitons implanter est cohérent. Comme nous l'avons expliqué, c'est une propriété que nous ne pouvons en aucune manière prouver de manière irréfutable. En réalité, nous supposons que le système est fortement normalisable, i.e. que le système n'accepte que des programmes qui terminent. La cohérence logique est souvent une conséquence assez facile de la normalisation forte[8]. Nous pourrons alors vérifier l'implantation.

Pollack [56] s'est attaché au problème de la vérification de l'implantation pour le système Lego, en utilisant le système Lego lui-même. Nous allons tenter de le faire pour Coq, qui a l'avantage de posséder un mécanisme d'extraction qui permet d'engendrer automatiquement le code source en Objective Caml des programmes dont on sait prouver la spécification. Un aspect pratique de faire la preuve de l'implantation de Coq en Coq est que cela permet, sur un exemple grandeur nature, de se placer à la fois du côté de l'utilisateur puisque l'on passe beaucoup de temps à rechercher des preuves à l'aide du système, mais aussi du point de vue du concepteur.

Faire la preuve avec un outil interactif (par opposition à un système automatique et sans preuve explicite) comme Coq fait que l'on est obligé de se pencher sur les détails de l'implantation, et que cela nous fournit des explications informelles, mais convaincantes (ce manuscrit est informel dans le sens où il n'est pas vérifié mécaniquement, mais il est plus accessible que l'ensemble des scripts purs). La preuve a donc au moins autant de valeur qu'une preuve rédigée à la main. Si Coq est erroné, on a une chance de s'en rendre compte lors des interactions avec le système : on a un meilleur contrôle de la nature des arguments logiques employés. Si un des axiomes de la logique s'avère douteux[9], on peut chercher à ne pas vouloir trop reposer dessus. Toutes ces précautions ne peuvent pas être prises avec des systèmes entièrement automatisés, qui ne font que répondre oui ou non à une question. Comme, en l'occurrence, la question est de savoir si le système lui-même est correct, on voudrait avoir une réponse un peu plus circonstanciée.

Ce système de preuve engendré étant relativement proche du système dans lequel est formulée la preuve des propriétés de ce même système, on se trouve dans une situation similaire à celle des compilateurs "bootstrappés", i.e. le compilateur est lui-même écrit en utilisant le langage source du compilateur. Ici, on formalise l'implantation de Coq à l'aide de Coq. L'extraction permet donc d'engendrer un nouveau programme qui aura toutes les fonctionnalités de Coq. Il serait notamment capable de revérifier la formalisation de sa

[8] C'est une hypothèse a priori plus forte puisqu'il peut exister des systèmes cohérents non normalisants.

[9] Reprenons la Théorie des Ensembles de Cantor, système qui semblait simple et naturel, que les mathématiciens employaient sans s'en rendre compte : bien que ce système ait montré une incohérence, les preuves faites jusqu'ici ont pu se traduire dans les Théories des Ensembles Z ou ZF, car la plupart des preuves ne faisaient pas appel à l'ensemble de tous les ensembles, responsable du paradoxe.

propre implantation. Le système sera bootstrappé lorsque le code engendré par extraction sera identique au code du système.

Même si l'objectif général est la vérification de l'implantation, nous ne nous obligerons pas à suivre le code actuel (version 6.2) à la lettre pour plusieurs raisons :

- certains aspects de la logique ne sont pas encore tout à fait satisfaisants, comme par exemple la vérification que les fonctions définies par point fixe terminent.
- L'implantation souffre d'un manque de spécification claire, ce qui peut augmenter le risque d'erreur de programmation, et qui en tout cas est source d'inefficacité car cela oblige à programmer de manière défensive (i.e. en vérifiant de manière parfois redondante que les préconditions sont effectivement satisfaites).
- Le code utilise largement le style de programmation impératif. Or, l'extraction ne permet actuellement d'engendrer que des programmes purement fonctionnels.
- L'algorithme de typage employé n'est pas complet, ce qui signifie que certains programmes bien typés seront rejetés par le système. C'est la formalisation du système qui a permis de mettre à jour certains de ces problèmes.

Ces éléments font que le bootstrap ne pourra pas être réalisé du premier coup, mais au deuxième, lorsque le code extrait extraira un programme de la formalisation. Ce dernier sera identique au code issu de la première extraction (dans la mesure où la fonction d'extraction formalisée est la même que celle du système actuel, sinon on atteint le bootstrap à la troisième génération).

Il faudra faire attention de ne pas confondre les différents niveaux. Il y a celui où l'on décrit la théorie. Pour ce niveau, on emploiera les notations habituelles en mathématiques, ou alors les notations Coq. Il y a aussi le niveau objet, la théorie que l'on développe, qui dans notre cas permet d'exprimer les mêmes choses que le niveau méta. On essaiera d'utiliser des notations différentes pour les mêmes notions à des niveaux distincts.

1.3 Conventions mathématiques

La plupart des résultats présentés dans ce travail ont été formalisés en Coq, et sont disponibles sur le Web au format HTML. Tous les définitions, lemmes et théorèmes figurant dans des environnements numérotés et accompagnés d'une référence en `verbatim` ont été prouvés formellement en Coq, le nom étant celui de l'objet formel. La présentation qui va suivre évitera, dans sa forme, de paraphraser le développement en Coq (seul document vérifié mécaniquement) en employant des notations courantes en mathématiques. Bien que ce décalage puisse être nuisible, il nous semble avoir des avantages certains : la présentation est moins lourde, plus accessible aux non initiés à Coq. Enfin, il atténue le problème de confusion entre les niveaux dans la théorie (théorie méta et théorie objet).

Bien sûr, nous utiliserons les conventions habituelles en ce qui concerne les connecteurs logiques comme la conjonction ($\wedge$), la disjonction ($\vee$), la négation ($\neg$), l'implication ($\Rightarrow$), l'existentielle ($\exists$), la définition d'ensembles par compréhension ($\{x \mid P(x)\}$), etc. L'ensemble des parties d'un ensemble ou d'un type E sera noté $\mathcal{P}(E)$.

Le langage méta étant intuitionniste, il possède des connecteurs permettant d'exprimer l'existence d'algorithmes, i.e. de fonctions calculables ayant certaines propriétés. Un mécanisme d'extraction permet alors d'engendrer automatiquement le source d'un programme Objective Caml implantant cet algorithme, avec la garantie que le programme engendré suit sa spécification. Ces connecteurs, tout comme la plupart de ceux du paragraphe précédent, ne sont pas primitifs dans la logique de Coq ; ce sont de simples notations qui sont définies dans le prélude du système, car elles sont couramment utilisées.

Définition 1.1 *Pour tout prédicat P, l'existentielle constructive $\exists^\star x.\, P(x)$ permet de spécifier un programme qui calcule un objet t tel que $P(t)$ soit vrai. Ce connecteur est plus fort que l'existentielle courante : $(\exists^\star x.\, P(x)) \;\Rightarrow\; \exists x.\, P(x)$.*

Définition 1.2 *Si A et B sont des spécifications de programmes, nous introduisons la disjonction calculatoire $A+B$, qui est plus forte que la disjonction classique : $A+B \;\Rightarrow\; A \vee B$.*

Les développements de cette thèse ont requis la définition de connecteurs supplémentaires, revenant assez souvent dans les spécifications.

Définition 1.3 (connecteur `decide`) *La décidabilité d'une proposition P notée $\mathsf{Dec}\,(P)$ implique qu'il existe un algorithme construisant un booléen ayant la valeur de vérité de P.*

$$\mathsf{Dec}\,(P) \stackrel{\text{def}}{\equiv} P + \neg P$$

Définition 1.4 *Le connecteur suivant est utilisé pour spécifier les programmes faisant des recherches pouvant échouer :*

$$\mathsf{DecFind}\,(x \mid P(x)) \stackrel{\text{def}}{\equiv} \exists^\star x.\, P(x) + \forall x.\, \neg P(x)$$

Il spécifie un programme qui soit retourne un objet vérifiant le prédicat P, soit une constante particulière (qui s'appelle `inright`*) lorsque qu'il n'existe pas d'objet satisfiant P.*

Typiquement, on peut spécifier la fonction qui recherche l'image d'une clé x dans une liste d'association l par :

$$\forall x, l.\, \mathsf{DecFind}\,(y \mid (x, y) \in l)\,.$$

Définition 1.5 *L'existence d'un élément minimal pour une relation R, satisfiant un prédicat P :*

$$\exists^{\mathsf{Ppal}}(x \mid P(x),\, R) \stackrel{\text{def}}{\equiv} \exists^\star x.\, (P(x) \wedge (\forall y.\, P(y) \;\Rightarrow\; x\; R\; y))$$

On peut vérifier que la formule $\exists^{\mathsf{Ppal}}(x \mid P(x),\, =)$ signifie que l'on sait calculer un x tel que $P(x)$, et de plus cet x est *unique*.

Définition 1.6 *Étant donné un prédicat P et un ordre R, la recherche d'un élément minimal pour R satisfiant P se spécifie à l'aide de :*

$$\mathsf{InfPpal}(x \mid P(x),\, R) \stackrel{\text{def}}{\equiv} \exists^{\mathsf{Ppal}}(x \mid P(x),\, R) + \forall x.\, \neg P(x)$$

Cela signifie que soit on sait calculer le plus petit élément (par rapport à R) vérifiant le prédicat P, ou bien on peut prouver qu'il n'existe pas d'élément vérifiant P.

Nous aurons souvent besoin de décrire des structures (ou enregistrements) regroupant plusieurs paramètres. La notation

$$\langle champ_1 : type_1; \ldots; champ_n : type_n \rangle$$

désigne le type des enregistrements à n champs dont les champs $champ_1$ à $champ_n$ ont respectivement les types $type_1$ à $type_n$. La notation totalement explicite pour construire un enregistrement est

$$\langle champ_1 = valeur_1; \ldots; champ_n = valeur_n \rangle\,.$$

Lorsqu'il n'y a pas d'ambiguïté, on utilisera plutôt la notation

$$\langle valeur_1; \ldots; valeur_n \rangle$$

en respectant l'ordre des champs de la déclaration. Enfin, il est toujours possible d'accéder aux différentes composantes d'un enregistrement. La projection sur le champ $champ_i$ d'un enregistement R se note

$$R.champ_i$$

Les enregistrements servent essentiellement à construire un produit cartésien de n types, mais ils seront aussi employés pour définir des sous-ensembles par compréhension. Ainsi, $\{x \in T \mid P(x)\}$ sera en réalité l'ensemble des enregistrements dont le premier champ est un élément t du type T, et dont le deuxième est une preuve de $P(t)$. Cela correspond aussi à la notion de sous-type en PVS (voir [46, 45]).

En dernière remarque, notre langage mathématique, comme tous les langages de programmation, possède une syntaxe pour insérer des commentaires au milieu de formules. Par exemple, lorsqu'une structure est introduite, on utilisera les commentaires pour décrire brièvement les différents champs :

$$\begin{array}{ll} \langle\ \text{num} : & \mathbb{N}; \quad (* \text{ numérateur } *) \\ \quad \text{den} : & \mathbb{N}^* \quad (* \text{ dénominateur } *) \\ \rangle & \end{array}$$

On trouvera dans l'annexe A quelques résultats standards sur les ordres bien fondés utilisés au cours des développements.

1.4 Plan de la thèse

Le chapitre 2 décrit assez informellement un certain nombre de formalismes, qui sont des variations autour du système de Coq, ainsi que leurs principales caractéristiques. Il abordera aussi les problèmes d'architecture que cela pose pour implanter ces formalismes sur ordinateur. Enfin, on ébauche une formalisation décrivant et prouvant la correction de l'architecture d'un système par rapport au système logique, montrant ainsi que l'étape vraiment complexe dans la réalisation d'un système de preuve certifié est la construction de son noyau.

Dans le chapitre 3, nous abordons un aspect pratique d'implantation. Puisqu'un petit langage de programmation est intégré au formalisme, il est souhaitable de savoir implanter efficacement un interpréteur pour ce langage. Nous verrons dans les chapitres ultérieurs à quel point cet aspect est sensible. Nous introduirons un calcul de substitutions explicites conçu pour permettre d'implanter de manière assez directe une stratégie paresseuse, ce qui donnera lieu ensuite à une machine abstraite, qui est une version légèrement simplifiée de celle que nous avons intégrée au système Coq dans la version 6.2.

Dans la deuxième partie de la thèse, nous nous lancerons dans le cœur du sujet : nous proposerons et formaliserons plusieurs systèmes de types, de plus en plus généraux. Ils sont généraux dans le sens où ils comportent de nombreux paramètres. Le fait que le système de types soit encore en évolution nous incite à faire des développements modulaires et paramétrés, afin que les modifications apportées au système ne soient pas trop coûteuses, ce qui nous permettra de réutiliser au maximum les preuves déjà faites. Ainsi, le chapitre 4 décrira une formalisation des Systèmes de Types Purs (PTS), qui serviront en quelque sorte de test pour s'assurer que notre idée de système de preuve entièrement formalisé est réalisable.

Le chapitre 5 poursuit cet effort en offrant un cadre encore plus général, offrant la possibilité d'ajouter des constructions au langage par l'intermédiaire d'une signature. Cette

extensibilité permettra de traiter des systèmes de types aussi complexes que le Calcul des Constructions Inductives (CCI) de manière générique.

La troisième et dernière partie consistera en la description et la formalisation de CCI dans le cadre défini dans la deuxième partie. CCI représente un "grand sous-ensemble" de Coq. Les fonctionnalités comprennent le calcul des constructions muni d'une hiérarchie d'univers et des types inductifs, mais par rapport à Coq, il n'y a pas les types co-inductifs. En contrepartie, ce système va plus loin que l'implantation actuelle de Coq sur certains points : il est possible de considérer du sous-typage, que nous utiliserons pour implanter un système de marquage des types inductifs afin d'assurer la terminaison des définitions récursives. Ce système a été vérifié formellement, à l'exception de quelques axiomes que nous comptons prouver ultérieurement, ce qui permet d'une part d'avoir une bonne confiance en ce système, et d'autre part d'engendrer automatiquement le code d'un système de preuves.

Première partie

Architecture de systèmes de preuves

Chapitre 2

Formalisme et architecture

Une première partie de ce chapitre est une introduction à un certain nombre de systèmes logiques ayant joué un rôle important dans l'élaboration du formalisme de Coq, le Calcul des Constructions Inductives. Il sera beaucoup question de formalismes dans lesquels on manipule des objets-preuves.

Ensuite, nous nous penchons sur le problème d'implanter de tels systèmes sur ordinateur de manière sûre, ce qui nous fera esquisser l'architecture de ces programmes, avec d'une part un *noyau* de faible dimension, et d'autre part un ensemble de *tactiques*, des commandes qui permettent de construire des preuves par des moyens complexes, mais qui n'ont pas besoin d'être certifiées car elles reposent sur le noyau, entité qui est chargée de "vérifier" la preuve produite par les tactiques.

Enfin, nous verrons qu'il est possible, et même assez aisé (bien que parfois fastidieux) de spécifier le comportement d'une interface relativement rudimentaire avec l'utilisateur. Une fois cette tâche accomplie, nous pourrons consacrer le reste de cette thèse à l'étude des formalismes qui seront implantés dans le noyau de notre système de preuve.

2.1 Preuves formelles

2.1.1 Calcul des séquents et déduction naturelle

Nous allons évoquer les différents styles de présentation des systèmes logiques. Le but d'un tel système est de définir quelles sont les propositions (ou formules) que l'on considère comme vraies. On ne précise pas ce que sont exactement les formules, car cela dépend des systèmes, et l'on veut rester général. Pour l'instant, on peut imaginer que ce sont des expressions construites à l'aide d'opérateurs logiques comme $\wedge$, $\vee$, $\Rightarrow$, $\bot$, $\neg$ pour la logique propositionnelle, ou des quantificateurs $\forall$ et $\exists$ pour la logique de premier ordre.

En revanche, on retrouve toujours la notion de *séquent*. Un séquent est un couple de listes de formules. Soit Γ et Δ deux listes de formules, on note le séquent

$$\Gamma \vdash \Delta$$

exprimant le fait que si l'on suppose que toutes les formules de Γ sont vraies, alors l'une des formules de Δ est vraie. On appellera Δ la conclusion du séquent et Γ son contexte.

L'ensemble des séquents que l'on considère comme valides se définit à l'aide de règles d'inférence de la forme

$$\frac{P_1 \quad \dots \quad P_n}{Q}$$

où les P_i et Q sont des séquents. Cette règle exprime le fait que si tous les séquents P_i sont dérivables, alors Q l'est aussi. Ces règles peuvent se combiner pour former des *dérivations*. Cela signifie que si l'on est capable de construire des dérivations dont les conclusions sont P_1, P_2 ... P_n, alors Q sera considéré comme valide. La dérivation se termine avec des règles sans prémisses.

Toutes les variables libres d'une règle sont universellement quantifiées. Cela revient à dire que lorsque l'on considère une règle, on considère en fait toutes les instances consistant à remplacer les variables par des formules quelconques. De plus, les règles peuvent être conditionnelles, c'est-à-dire qu'on ne considère que les instances qui satisfont la condition.

Ce qui peut différer d'un système à l'autre, c'est la forme et le nombre de règles admises : il peut y en avoir un petit nombre, fixé, ou bien on peut avoir un système ouvert et y rajouter ses propres règles. On considèrera ici plutôt des systèmes ayant un ensemble restreint et fixé de règles de déductions élémentaires, ce qui rend la preuve de cohérence du système logique plus simple et plus crédible.

À titre d'exemple, on peut présenter un fragment de la logique intuitionniste, exprimé dans le style de la déduction naturelle, ce qui signifie que l'on ne considère que des séquents où la liste de droite (Δ) est réduite à une seule formule.

$$\frac{T \in \Gamma}{\Gamma \vdash T}\ (\text{Hyp}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow\text{-I}) \qquad \frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow\text{-E})$$

La première de ces règles (qui est conditionnelle) permet de conclure lorsque la formule à prouver est présente dans l'ensemble des hypothèses. La deuxième signifie que pour prouver $A \Rightarrow B$, il suffit de rajouter A à l'ensemble des hypothèses et de prouver B dans ce nouveau contexte. C'est la règle d'*introduction* de l'implication car sa conclusion est une implication. La dernière règle s'appelle *Modus ponens*, ou règle d'*élimination* de l'implication, puisqu'elle permet d'"utiliser" une implication. Selon cette règle, si l'on a prouvé $A \Rightarrow B$ et A, alors on peut en déduire B.

En composant différentes instances de ces règles, on peut prouver que l'implication est transitive ($\Rightarrow$ étant associative à droite, i.e. $A \Rightarrow B \Rightarrow C$ signifie $A \Rightarrow (B \Rightarrow C)$, qui est logiquement équivalent à $A \wedge B \Rightarrow C$) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{B \Rightarrow C \in \Gamma}{\Gamma \vdash B \Rightarrow C}\ (\text{Hyp}) \qquad \dfrac{\dfrac{A \Rightarrow B \in \Gamma}{\Gamma \vdash A \Rightarrow B}\ (\text{Hyp}) \qquad \dfrac{A \in \Gamma}{\Gamma \vdash A}\ (\text{Hyp})}{\Gamma \vdash B}\ (\Rightarrow\text{-E})}{\Gamma = (A \Rightarrow B, B \Rightarrow C, A) \vdash C}\ (\Rightarrow\text{-E})}{A \Rightarrow B, B \Rightarrow C \vdash A \Rightarrow C}\ (\Rightarrow\text{-I})}{A \Rightarrow B \vdash (B \Rightarrow C) \Rightarrow A \Rightarrow C}\ (\Rightarrow\text{-I})}{\vdash (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C}\ (\Rightarrow\text{-I})$$

Noter que l'on a économisé beaucoup d'encre en définissant le nom Γ pour dénoter le contexte $(A \Rightarrow B, B \Rightarrow C, A)$ qui apparaît beaucoup dans la dérivation. Cela donne une idée de la taille que peuvent occuper les dérivations de théorèmes complexes. Il n'est pas raisonnable de présenter les dérivations sous cette forme.

La preuve peut être rendue beaucoup plus courte si l'on supprime les redondances qu'elle comporte. Notamment, il n'est pas nécessaire de rappeler à chaque fois le contexte dans lequel on se trouve puisqu'on peut toujours inférer le contexte des prémisses à partir du séquent conclusion. On introduit donc les *termes-preuves*, qui sont une trace de la dérivation, ne gardant que les informations nécessaires, et l'on utilise la notation

$$\Gamma \vdash \pi : P$$

qui signifie que sous les hypothèses Γ, le terme π est une preuve de la proposition P. Le séquent a été annoté avec le terme-preuve qui permet de reconstruire la dérivation.

Dans notre exemple, les informations dont nous avons besoin comprennent le nom de la règle employée, et pour chaque prémisse, un sous-terme pour représenter celles-ci, ainsi qu'éventuellement d'autres termes lorsque l'instance de la règle n'est pas uniquement définie par le séquent conclusion. C'est le cas pour la troisième règle : on ne peut pas deviner comment on doit instancier A. Les règles pourraient être :

$$\frac{T \in \Gamma}{\Gamma \vdash \mathsf{H} : T}\ (\text{Hyp}) \quad \frac{\Gamma, A \vdash \pi : B}{\Gamma \vdash \mathsf{I}(\pi) : A \Rightarrow B}\ (\Rightarrow\text{-I}) \quad \frac{\Gamma \vdash \pi_1 : A \Rightarrow B \qquad \Gamma \vdash \pi_2 : A}{\Gamma \vdash \mathsf{E}(A, \pi_1, \pi_2) : B}\ (\Rightarrow\text{-E})$$

On pourrait donc résumer la dérivation ci-dessus par le séquent annoté suivant :

$$\vdash \mathsf{I}(\mathsf{I}(\mathsf{I}(\mathsf{E}(B, \mathsf{H}, \mathsf{E}(A, \mathsf{H}, \mathsf{H}))))) : (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C$$

Un résultat particulièrement important est la décidabilité de la vérification de preuve, c'est-à-dire qu'il est décidable de savoir si un terme permet de reconstruire la dérivation d'un séquent. Une preuve constructive de ce résultat nous donne un algorithme de vérification de preuves, ce qui est ce que nous recherchons. Dans le cas ci-dessus, c'est évident puisque les règles vérifient la propriété de la sous-formule, à savoir que toutes les variables qui apparaissent dans la conclusion d'une règle apparaissent aussi dans les prémisses.

2.1.2 Isomorphisme de Curry-Howard

Mais nous avons de bonnes raisons pour choisir une autre solution. La sémantique de Heyting-Kolmogorov donne une interprétation calculatoire des preuves. Les connecteurs usuels des théories de premier ordre s'interprètent de la manière suivante :

- La paire sert à représenter les preuves de conjonction : si π_1 est une preuve de A et π_2 une preuve de B, alors (π_1, π_2) est une preuve de $A \wedge B$. Cela se comporte bien comme la conjonction puisque l'on ne peut construire une preuve de $A \wedge B$ qui si l'on a à la fois une preuve de A et une preuve de B. Enfin, d'une preuve de $A \wedge B$, on pourra extraire une preuve de A et une preuve de B.
- Les fonctions interprètent l'implication : une preuve de $A \Rightarrow B$ est une fonction qui à toute preuve de A, associe une preuve de B. La règle Modus-Ponens s'interprète par l'application : étant donné une preuve f de $A \Rightarrow B$, et une preuve x de A, alors on peut en déduire une preuve de B en calculant $f(x)$.
- La preuve de la disjonction $A \vee B$ est une paire formée d'un booléen b, et d'une preuve de A si $b = \text{true}$ ou d'une preuve de B si $b = \text{false}$. Cela capture bien le fait qu'on pourra prouver $A \vee B$ si l'on a une preuve de A ou une preuve de B. De plus, lorsque l'on dispose d'une preuve de $A \vee B$, ne connaissant pas la valeur du booléen, on ne sait pas si la deuxième composante est une preuve de A ou de B, ce qui oblige à envisager les deux cas.
- Une preuve de $\forall x.\, P(x)$ est une fonction qui à chaque individu t associe une preuve de $P(t)$.
- Une preuve de $\exists x.\, P(x)$ est une paire formée d'un terme t et d'une preuve de $P(t)$.

Nous n'avons donc pas à concevoir une structure de donnée particulière pour représenter les preuves, à partir du moment où notre formalisme comportera des fonctions et des structures de données comme la paire, ce qui semble la moindre des choses puisque nous souhaitons pouvoir certifier des programmes.

L'isomorphisme de Curry-Howard permet de remarquer que si l'on suit la sémantique de Heyting-Kolmogorov, le type d'une preuve est identique à la formule qu'elle prouve, en identifiant le type des fonctions avec le connecteur d'implication ($A \to B = A \Rightarrow B$), ainsi que le produit cartésien et la conjonction ($A \times B = A \wedge B$). Nous verrons plus loin que nous aurons besoin d'un nouvel opérateur de type pour représenter la quantification universelle. Une proposition peut être vue comme le type de ses preuves. Les propositions non prouvables sont donc des types vides.

On se rend compte que les règles de typage du λ-calcul simplement typé de Church correspondent exactement au système que nous avons introduit dans ce chapitre :

$$\frac{(x{:}T) \in \Gamma}{\Gamma \vdash x : T}\ (\text{Hyp}) \qquad \frac{\Gamma, (x{:}A) \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}\ (\to\text{-I}) \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}\ (\to\text{-E})$$

La preuve de la transitivité de l'implication est $\lambda f{:}A \to B.\, \lambda g{:}B \to C.\, \lambda x{:}A.\, (g\ (f\ x))$, qui est l'opérateur de composition de fonctions ($(f, g) \mapsto g \circ f$). Cela se comprend bien grâce à la sémantique de Heyting : il faut trouver une fonction qui associe une preuve de C à chaque preuve x de A, étant données une fonction f qui associe une preuve de B à chaque preuve de A et une fonction g qui associe une preuve de C à chaque preuve de B. Il est clair qu'il suffit d'utiliser f pour transformer x en une preuve de B, que l'on applique à g, donnant une preuve de C.

Le système `Coq` repose beaucoup sur l'isomorphisme de Curry-Howard. Nous pouvons donc illustrer notre exemple. Les notations sont un peu différentes : $\lambda x{:}A.\, B$ se note `[x:A]B`. On commence par déclarer trois variables A, B et C (nous verrons plus tard pourquoi il faut déclarer les variables propositionnelles comme des objets de type Prop).

```
Coq < Variables A,B,C :Prop.
A is assumed
B is assumed
C is assumed

Coq < Check [f :A->B][g :B->C][x :A](g (f x)).
[f :(A->B) ; g :(B->C) ; x :A](g (f x))
     : (A->B)->(B->C)->A->C
```

La commande `Check` prend en argument un terme, puis calcule et affiche le type de ce terme. Le résultat peut se lire de deux façons : le type de l'opérateur de composition de fonctions est $(A \to B) \to (B \to C) \to A \to C$, ou bien l'on infère que le terme est une preuve de la transitivité de l'implication.

Nous venons de voir que le λ-calcul simplement typé permettait de représenter la logique propositionnelle[1]. Si nous voulons travailler avec un système logique de premier ordre (muni des quantificateurs universels et existentiels, et des prédicats), il faut un système de types plus évolué. Nous entrons alors dans le domaine de la Théorie des Types, dont nous présenterons quelques systèmes dans la section 2.2.

2.1.3 Élimination des coupures et preuves canoniques

L'isomorphisme de Curry-Howard identifie les preuves avec des objets calculatoires comme les fonctions. Les fonctions de notre formalisme correspondant aux programmes, on peut se poser la question de comment interpréter l'évaluation d'un programme.

[1] En fait, on n'a qu'un sous-ensemble de la logique propositionnelle. Cependant, il suffirait de rajouter au langage un type $\bot$ (proposition absurde) et une preuve de $((P \to \bot) \to \bot) \to P$ pour chaque type P pour obtenir la logique classique. Tous les connecteurs classiques peuvent s'exprimer : $\neg A = A \to \bot$, $A \wedge B = \neg(A \to B \to \bot)$, $A \vee B = \neg(\neg A \wedge \neg B)$

Comme notre langage de programmation intégré est une sorte de ML, il est fortement typé, ce qui signifie que le résultat d'un programme a le même type que le programme non évalué. C'est d'ailleurs le but du typage : déterminer statiquement (i.e. sans l'exécuter effectivement) le type d'objet que calcule un programme. Ce résultat possède d'autres dénominations comme *auto-réduction* ou *subject reduction*.

Cela signifie que si l'on évalue la preuve d'une proposition, on obtient une preuve de cette même proposition. Il s'agit en quelque sorte d'une simplification de la preuve.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, (x{:}T) \vdash M : P_x}}{\Gamma \vdash \lambda x{:}T.\, M : \forall x{:}T.\, P_x}\ (\forall\text{-I}) \qquad \dfrac{\vdots}{\Gamma \vdash N : T}}{\Gamma \vdash (\lambda x{:}T.\, M\ N) : P_x\{x\backslash N\}}\ (\forall\text{-E})$$

La notation P_x signifie que la variable x peut apparaître dans le prédicat, et $P_x\{x\backslash N\}$ représente P_x dans lequel on a remplacé toutes les occurrences non liées de x par N.

On utilise consécutivement la règle d'introduction, puis d'élimination du quantificateur universel. Un tel enchaînement s'appelle une *coupure*. On aurait pu éviter la construction inutile de cette proposition en remplaçant dans la preuve M toutes les occurrences de x par N, c'est-à-dire chaque utilisation de la règle (Hyp) sur x par la preuve N. Autrement dit, la dérivation résumée par le séquent $\Gamma \vdash (\lambda x : T.\, M\ N) : P_x\{x\backslash N\}$ peut se simplifier en $\Gamma \vdash M\{x\backslash N\} : P_x\{x\backslash N\}$. On reconnaît la β-réduction, qui définit comment se calculent les appels de fonction :

$$(\lambda x{:}A.\, M\ N) \equiv M\{x\backslash N\}$$

Dans la logique du premier ordre intuitionniste, il existe des règles de simplification pour chaque paire de règles formée d'une règle d'introduction et d'une règle d'élimination du même connecteur.

Dans un formalisme ayant cette propriété, on peut prouver que dans un contexte vide, toute dérivation sans coupure d'une proposition P se termine par l'application d'une règle d'introduction du connecteur de tête de P. Cela fait apparaître la notion de preuve canonique, qui peut être vue comme le contrepoint des *valeurs* d'un type de données, objets formés à partir des constructeurs. Ce théorème joue un rôle important dans la possibilité d'extraire des programmes à partir d'une preuve intuitionniste de sa spécification.

Ceci n'est pas vrai en logique classique, qui considère la règle du tiers exclu :

$$\dfrac{}{\Gamma \vdash P \vee \neg P}\ (\text{EM})$$

En effet, c'est une règle d'introduction de la disjonction, mais on ne sait pas éliminer la coupure suivante

$$\dfrac{\dfrac{}{\Gamma \vdash A \vee \neg A}\ (\text{EM}) \qquad \dfrac{\vdots}{\Gamma \vdash A \to P} \qquad \dfrac{\vdots}{\Gamma \vdash \neg A \to P}}{\Gamma \vdash P}\ (\vee\text{-E})$$

puisque la preuve de $A \vee \neg A$ ne contient ni une preuve de A, ni une preuve de $\neg A$. Toutes les propositions risquent donc de contenir des preuves canoniques supplémentaires invoquant l'axiome du tiers exclu.

D'autre part, une propriété importante est que l'élimination des coupures termine, ce qui signifie que si une proposition admet une preuve, alors elle admet aussi une preuve sans coupures. Ce résultat est aussi appelé *normalisation forte*, puisqu'il montre aussi qu'il n'est pas possible d'écrire de programme qui ne termine pas.

La combinaison de ces deux faits a comme conséquence la cohérence du formalisme. Étant donné que la proposition absurde n'a pas de règle d'introduction, elle n'a pas de preuves sans coupures d'après le résultat précédent. Donc, la proposition absurde n'a pas de preuve du tout si le formalisme est fortement normalisant.

2.2 Théorie des Types

Cette section présente la structure du Calcul des Constructions Inductives (CCI), le formalisme qu'implante Coq. Cette structure sera décrite en retraçant les principaux formalismes de la Théorie des Types dont CCI emprunte des aspects. Ces traits caractéristiques seront illustrés par des exemples en Coq.

2.2.1 Types dépendants

Le système de types que nous allons présenter ici est une extension du λ-calcul simplement typé appelée λP. Ce formalisme a été utilisé dans l'un des premiers systèmes de preuves : Automath. Il permet de représenter la logique de premier ordre.

Logique du premier ordre

Les prédicats sont représentés par des fonctions qui à chaque objet du domaine associent une proposition suivant que l'objet satisfait le prédicat ou pas. Cela signifie que les types ne sont plus uniquement des variables propositionnelles, ou construits avec le connecteur d'implication, mais un type peut aussi être un symbole de prédicat appliqué à des termes, ou encore une formule quantifiée universellement (nous laissons pour l'instant de côté le connecteur existentiel, qui peut se déduire du quantificateur universel dans les systèmes classiques) :

$$Ty \quad := \quad X \mid X_1 \to X_2 \mid P(t_1, \ldots, t_n) \mid \forall x{:}X_1.\, X_2$$

Comme nous sommes dans un cadre où les objets sont typés, le prédicat P est déclaré en donnant son arité et le type de ses arguments. Certains des types que l'on peut former syntaxiquement ne sont plus corrects, par exemple si l'un des t_i n'a pas le type avec lequel est déclaré le i-ème argument de P. Les types sont donc aussi soumis à une forme de typage. Comme les environnements contiennent des types, il faudra vérifier que ceux-ci sont bien formés. Au lieu d'avoir un jugement de typage pour les termes et un autre pour les types, ces deux jugements étant mutuellement dépendants, il est plus simple de confondre termes et types et d'introduire une constante particulière Prop, le type des propositions. Un jugement $\Gamma \vdash T : \mathrm{Prop}$ signifie que T est un type bien formé, et nous pouvons donner une règle de typage pour la formation des types fonctionnels :

$$\frac{\Gamma \vdash A : \mathrm{Prop} \qquad \Gamma \vdash B : \mathrm{Prop}}{\Gamma \vdash A \to B : \mathrm{Prop}}$$

Si l'on a déclaré un prédicat P d'arité n dont les arguments ont pour type $T_1, \ldots, T_n$, on voudrait typer l'application de prédicat de la manière suivante :

$$\frac{\Gamma \vdash t_i : T_i}{\Gamma \vdash P(t_1, \ldots, t_n) : \mathrm{Prop}}$$

Nous profitons de cette unification des termes et des types pour fusionner les notions qui apparaissaient dans les deux mondes : les variables et l'application. Ainsi, la notation $P(t_1, \ldots, t_n)$ se code simplement par $(P\ t_1 \ldots t_n)$. Le fait d'utiliser le même espace de noms pour les variables de termes et de types pourra nous poser de nouveaux problèmes de captures de variables. Il faudra être vigilant.

Le fait de confondre l'application de prédicat avec l'application usuelle a plusieurs conséquences. D'une part, il faut pouvoir typer le symbole de prédicat, même s'il n'est pas appliqué. Le prédicat P déclaré ci-dessus devrait avoir le type $T_1 \to \ldots \to T_n \to \text{Prop}$. Pour l'instant, aucune règle ne permet de construire un tel type. D'autre part, il est possible d'écrire des expressions de prédicat comme le prédicat toujours faux : $\lambda x{:}T.\ \bot$

Pour obtenir un système de types correspondant à la logique de premier ordre, il reste à exprimer les quantificateurs universels et existentiels. Suivant la sémantique de Heyting, les preuves d'une quantification universelle comme $\forall x{:}A.\ P(x)$ sont des fonctions qui à chaque objet t du domaine A associe une preuve de $P(t)$. Ici, le type du résultat peut dépendre de la valeur de l'argument. C'est ce qu'on appelle le produit *dépendant*, qui généralise le type habituel des fonctions. En Théorie des Types, le produit dépendant se représente avec un Π : la formule mathématique $\forall x{:}A.\ P(x)$ s'écrit $\Pi x{:}A.\ P(x)$. En `Coq`, on écrirait `(x:A)(P x)`. Les règles suivantes montrent comment se forment les abstractions, les types produits et l'application. Elles sont temporaires et seront raffinées par la suite.

$$\frac{\Gamma, (x{:}A) \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B} \qquad \frac{\Gamma \vdash A : \text{Prop} \qquad \Gamma, (x{:}A) \vdash B : \text{Prop}}{\Gamma \vdash \Pi x{:}A.\ B : \text{Prop}}$$

$$\frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B\{0 \backslash N\}}$$

Avec l'isomorphisme de Curry-Howard, on voit la règle de typage de l'application comme la règle d'élimination du quantificateur universel :

$$\frac{\Gamma \vdash \forall x{:}A.\ P \qquad \Gamma \vdash N : A}{\Gamma \vdash P\{0 \backslash N\}}$$

On peut remarquer que le quantificateur universel permet de représenter l'implication. En effet, $A \to B$ peut se comprendre comme "pour toute preuve de A, B est vrai". Si A n'a pas de preuve, alors cette formule est vraie puisque c'est une quantification sur un ensemble vide. Lorsqu'il existe une preuve de A, alors elle nous dit que B est vraie. Cela correspond donc bien à l'implication. Noter que pour que ce raisonnement soit correct, il est primordial que les fonctions soient *totales*, i.e. associent une valeur de type B à chaque valeur de type A. L'opérateur flèche disparaît du formalisme, mais nous continuerons à utiliser la notation $A \to B$ pour $\Pi x{:}A.\ B$ lorsque x n'apparaît pas dans B.

Typage des prédicats

Ainsi, un prédicat unaire sur les entiers serait un terme ayant pour type nat $\to$ Prop, mais il faut être capable de typer ce type. On peut soit introduire une règle particulière, soit faire rentrer ce cas dans celui qui permet de typer nat $\to$ nat. Il y a deux choses à régler pour pouvoir typer ce terme. Il faut que Prop ait un type, et ensuite autoriser le produit entre un type et Prop. Pour cela, on introduit une autre constante : Type, le type de Prop. La règle de formation des produits est alors étendue de manière à autoriser le produit entre

un objet de type Prop et un autre de type Type[2]. La règle de l'abstraction doit aussi être changée car elle permettrait de dériver un jugement comme

$$\Gamma \vdash \lambda x{:}T.\,\mathrm{Prop} : T \to \mathrm{Type}.$$

Nous ne souhaitons pas former des types comme $T \to \mathrm{Type}$ dans ce système. C'est pourquoi il faut vérifier que le produit (type de l'abstraction) est un type que l'on souhaite effectivement former.

L'autre problème de typage que nous avions à résoudre est que les environnements doivent être aussi soumis à une forme de typage. Nous introduisons un nouveau jugment $\Gamma \vdash$ qui signifie que le contexte Γ ne contient que des types bien formés. On n'aura le droit d'ajouter un type dans l'environnement que si celui-ci est typé par Prop ou Type. Cela permet d'introduire à la fois des variables d'individus (grâce à Prop), mais aussi des variables propositionnelles ou de prédicats (grâce à Type).

Ces nouveautés sont résumées par l'introduction des règles suivantes :

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathrm{Prop} : \mathrm{Type}} \qquad \frac{\Gamma \vdash A : \mathrm{Prop} \qquad \Gamma, (x{:}A) \vdash B : s \qquad s \in \{\mathrm{Prop}, \mathrm{Type}\}}{\Gamma \vdash \Pi x{:}A.\,B : s}$$

$$\frac{\Gamma, (x{:}A) \vdash M : B \qquad \Gamma \vdash \Pi x{:}A.\,B : s \qquad s \in \{\mathrm{Prop}, \mathrm{Type}\}}{\Gamma \vdash \lambda x{:}A.\,M : \Pi x{:}A.\,B}$$

$$\frac{}{[\,] \vdash} \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash T : s \qquad s \in \{\mathrm{Prop}, \mathrm{Type}\}}{\Gamma, (x{:}T) \vdash}$$

Nous commençons à voir que les constantes Prop et Type jouent un rôle particulier. C'est pourquoi nous donnerons un nom particulier à ces constantes. Nous les appellerons *sortes* et elles sont chacune le type d'une certaine classe de types.

Nous avons vu que Prop était le type des propositions. Pour sa part, Type apparaît comme la sorte des prédicats : il contient Prop (type des propositions), nat $\to$ Prop (prédicats d'arité 1 sur les entiers), nat $\to$ nat $\to$ Prop (prédicats d'arité 2 sur les entiers), mais aussi des prédicats sur des fonctions : (nat $\to$ nat) $\to$ Prop. Il n'est en revanche pas possible de définir un prédicat sur les prédicats (dont le type serait (nat $\to$ Prop) $\to$ Prop), car cela n'est pas possible en logique du premier ordre.

Règle de conversion

Avec les règles que nous avons introduites jusqu'ici, nous formons un système qui n'a pas toutes les bonnes propriétés souhaitées. En particulier, il n'a pas la propriété d'autoréduction, qui signifie que si un terme à un type T, alors si on l'évalue, on obtient un terme qui a encore le type T. Ce résultat est très important car il garantit que les programmes bien typés s'exécuteront sans erreur. D'un point de vue logique, cette propriété est en général invoquée pour prouver la cohérence du formalisme.

Supposons que nous ayons un prédicat P sur les entiers et une constante π de type $\Pi n{:}\mathbb{N}.\,P(n)$. Si $M : \mathbb{N}$, alors $(\pi\ M) : P(M)$. Or, si M se réduit vers M', $(\pi\ M') : P(M')$, mais il n'a a priori plus le type $P(M)$.

```
Coq < Variables P : nat->Prop ;
Coq <           pi : (n :nat)(P n).
P is assumed
```

[2] N'autoriser que le produit entre un objet de type Prop et Prop lui-même n'aurait pas permis de typer les types des prédicats d'arité deux.

```
pi is assumed
Coq < Check (pi (plus (S O) (S O))).
(pi (plus (S O) (S O)))
     : (P (plus (S O) (S O)))
Coq < Eval Compute in (plus (S O) (S O)).
     = (S (S O))
     : nat
Coq < Check (pi (S (S O))).
(pi (S (S O)))
     : (P (S (S O)))
```

Le problème que cet exemple soulève n'est pas que le système de types est erroné et que l'on pourra par exemple appliquer un terme à quelque chose qui n'est pas une fonction, mais simplement qu'il manque des règles permettant d'attribuer à $(\pi\ M')$ le type $P(M)$.

Il suffit d'introduire une règle, appelée *règle de conversion*, qui établit que si M est de type T et que U est un type bien formé, convertible avec T, alors M a aussi le type U. Nous sommes volontairement peu explicites sur la définition de "types convertibles" car cela peut varier d'un système à l'autre. Intuitivement, c'est une relation qui indique quels sont les types que l'on veut identifier. Cela comprend généralement la β-réduction, mais nous pouvons imaginer d'autres règles à prendre en compte dans cette équivalence, comme par exemple l'expansion des définitions, la réduction du filtrage (comme dans les langages de la famille ML), ou l'expansion des points fixes.

Enfin, on considère la possibilité de sous-typage, au sens où si $A \leq B$, tout objet de type A est considéré comme ayant aussi le type B. Cela permet de passer certaines étapes sous silence dans le terme preuve. Cela permet de rendre compte de certaines inclusions en mathématiques comme par exemple $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$. Mais encore faut-il que la représentation de n en tant qu'entier soit la même que sa représentation en tant que réel. Sinon, il faut utiliser une coercion qui traduit la représentation.

Ainsi, la règle qui suit nous permet d'obtenir un système qui aura la propriété d'auto-réduction :

$$\frac{\Gamma \vdash M : T \qquad \Gamma \vdash U : s \qquad T \leq U}{\Gamma \vdash M : U}$$

Un exemple particulier de sous-typage que nous allons étudier est la cumulativité. Le principe est d'assouplir un peu le typage en incorporant automatiquement les objets d'un certain univers dans tous les univers plus haut dans la hiérarchie. Cela revient à définir la relation $\leq$ de manière à ce que la règle suivante soit admissible :

$$\frac{\Gamma \vdash T : \mathrm{Type}_i}{\Gamma \vdash T : \mathrm{Type}_{i+k}}$$

2.2.2 Imprédicativité, Calcul des Constructions

Indépendamment de λP, le λ-calcul simplement typé a évolué vers le système F (Girard [26]), qui permet d'écrire toutes les fonctions de l'arithmétique d'ordre 2.

Le système F est imprédicatif, ce qui signifie que l'on peut former une proposition qui est une quantification sur toutes les propositions.

```
Coq < Check (A :Prop)A->A.
(A :Prop)A->A
     : Prop
```

Enfin, on arrive à $F\omega$, qui généralise encore le système F. Dans $F\omega$, il est possible d'écrire des fonctions des types vers les types, ou des prédicats vers les types comme le connecteur existentiel. Nous avons donc maintenant une logique d'ordre supérieur, et le connecteur "si et seulement si" (équivalence de deux propositions) peut se définir à partir des autres connecteurs.

```
Coq < Check [A,B :Prop] (A->B) /\ (B->A).
[A,B :Prop](A->B)/\(B->A)
     : (_,_ :Prop)Prop

Coq < Check ex.
ex
     : (A :Set)(A->Prop)->Prop
```

Ces deux exemples sont des termes que l'on ne pouvait typer dans le formalisme de la section précédente car il font appel à des produits dont le domaine est un type de la sorte Type. Il suffit de modifier la règle de formation du produit pour autoriser ces types. On obtient alors un système de type appelé *Calcul des Constructions*, qui a été défini par Coquand et Huet dans [11, 13]. Les règles suivantes forment un résumé de ce que nous avons vu jusqu'ici :

$$\frac{}{[]\vdash} \qquad \frac{\Gamma\vdash \qquad \Gamma\vdash T:\ s \qquad s\in\{\mathrm{Prop},\mathrm{Type}\}}{\Gamma,(x{:}T)\vdash}$$

$$\frac{\Gamma\vdash}{\Gamma\vdash \mathrm{Prop}:\ \mathrm{Type}} \qquad \frac{\Gamma\vdash A:\ s_1 \qquad \Gamma,(x{:}A)\vdash B:\ s_2 \qquad s_1,s_2\in\{\mathrm{Prop},\mathrm{Type}\}}{\Gamma\vdash \Pi x{:}A.\,B:\ s_2}$$

$$\frac{\Gamma\vdash \qquad (x{:}T)\in\Gamma}{\Gamma\vdash x:\ T} \qquad \frac{\Gamma\vdash M:\ \Pi x{:}A.\,B \qquad \Gamma\vdash N:\ A}{\Gamma\vdash (M\ N):\ B\{0\backslash N\}}$$

$$\frac{\Gamma,(x{:}A\vdash M:\ B \qquad \Gamma\vdash \Pi x{:}A.\,B:\ s \qquad s\in\{\mathrm{Prop},\mathrm{Type}\}}{\Gamma\vdash \lambda x{:}A.\,M:\ \Pi x{:}A.\,B}$$

$$\frac{\Gamma\vdash M:\ T \qquad \Gamma\vdash U:\ s \qquad T\leq U}{\Gamma\vdash M:\ U}$$

2.2.3 Sortes, univers, PTS

Stratification des types

Comme dans le Calcul des Constructions on autorise tous les produits, il est naturel de se poser la question s'il n'est pas possible de confondre les différentes sortes de types Prop et Type en une seule que l'on appellerait Type. Ce type serait lui-même un type grâce à la règle

$$\frac{\Gamma\vdash}{\Gamma\vdash \mathrm{Type}:\ \mathrm{Type}.}$$

Hélas, ce système proposé par Martin-Löf en 1971, qui aurait été remarquablement simple, est incohérent, comme l'a montré le paradoxe de Girard. La théorie des ensembles de Cantor permettait d'introduire l'ensemble de tous les ensembles. Le système proposé introduit le type de tous les types. Il n'est dons pas très surprenant d'aboutir aussi à un paradoxe. Il faut donc abandonner l'idée que tous les types soient situés au même niveau. Les sortes servent alors à identifier les différentes "strates" de types.

Il est parfois insuffisant de n'avoir que deux sortes lorsque l'on veut écrire des fonctions dont les images sont des types. Par exemple, dans le Calcul des Constructions, on ne peut

pas définir la fonction qui a un entier n, associe le type des prédicats d'arité n sur les entiers, car cela serait un objet de type nat $\to$ Type qui n'est pas bien typé car Type n'a pas de type. Nous pouvons utiliser la même astuce que pour typer les prédicats, en rajoutant une nouvelle sorte, qui serait le type de Type.

Cet argument pouvant se reproduire relativement souvent, nous allons maintenant considérer des systèmes ayant un nombre arbitraire de sortes et la possibilité de paramétrer les différentes quantifications autorisées : il s'agit des Systèmes de Types Purs, ou PTS (voir [4] pour une présentation de cette classe de systèmes). L'intérêt d'étudier de tels systèmes paramétrés est de montrer certains résultats métathéoriques pour un grand nombre de systèmes à la fois. Tous les systèmes de types que nous avons vus jusqu'ici peuvent s'exprimer comme des PTS particuliers.

Le cadre général des Systèmes de Types Purs

Un PTS est caractérisé par un triplet de paramètres $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, où $\mathcal{S}$ est l'ensemble des sortes. Le paramètre $\mathcal{A}$ est une relation binaire sur $\mathcal{S}$ qui définit comment sont typées les sortes. En général, cela définit une hiérarchie sans cycle entre les sortes, car un tel cycle nous rapprocherait dangereusement du système Type : Type[3]. Le troisième paramètre indique quels sont les produits qui peuvent être formés, ainsi que la sorte dans laquelle vit le type produit. Ainsi, si $(s_1, s_2, s_3) \in \mathcal{R}$, alors pour tous types $A : s_1$ et $B : s_2$, le produit $\Pi x{:}A.\, B$ a le type s_3. Les règles spécifiques aux PTS sont :

$$\frac{\Gamma \vdash \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, (x{:}A) \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash \Pi x{:}A.\, B : s_3}$$

Exemple 2.1 *Le cube de Barendregt, est un ensemble de huit PTS dont l'ensemble des sortes possède deux éléments* Prop *et* Type*, avec* Prop : Type*, ayant au moins la règle* (Prop, Prop, Prop)*, soit*

$$\mathcal{S} = \{\text{Prop}, \text{Type}\} \qquad \mathcal{A} = \{(\text{Prop}, \text{Type})\}$$

Parmi ces huit systèmes que l'on peut former, on retrouve le λ-calcul simplement typé, λP, F, $F\omega$, et le Calcul des Constructions.

Avec l'ensemble de sortes défini ci-dessus, on peut achever la définition du système F en donnant les règles de formations du produit :

$$\left\langle \begin{array}{ll} (\text{Prop}, \text{Prop}, \text{Prop}) & (*\ ex.\ : \text{nat} \to \text{nat}\ *) \\ (\text{Type}, \text{Prop}, \text{Prop}) & (*\ ex.\ : \Pi P{:}\text{Prop}.\, P\ *) \end{array} \right\rangle$$

Cela montre bien que les cadre des PTS englobe un bon nombre de systèmes intéressants. Un autre avantage d'étudier des systèmes généraux est que cela permet de retarder le choix du formalisme exact que l'on va utiliser.

Il existe une autre motivation plus fondamentale pour considérer des systèmes paramétrés : cela permet de mieux comprendre l'influence de chacun de ces paramètres sur les propriétés métathéoriques. Par exemple, tous les PTS ne sont pas fortement normalisants. Ce n'est pas une si mauvaise chose car cela permet de voir avec précision quels sont les facteurs qui font qu'un système est normalisant ou pas. On peut se rendre compte facilement

[3] Il existe des systèmes (par exemple le système F cyclique, voir [41]) pour lesquels une sorte s_1 a pour type s_2 qui elle-même a pour type s_1, sans que cela ne crée de paradoxe.

dans quelles conditions l'imprédicativité peut être acceptée dans un système sans perdre la cohérence.

Les Systèmes de Types Purs ont été longuement étudiés dans la littérature, c'est pourquoi nous ne donnerons que très peu de détails dans cette section. On pourra se reporter aux travaux de Barendregt [4], ainsi que de Geuvers et Nederhof [22]. Grâce à la concision de la formulation de ces formalismes, ceux-ci se prêtent bien à la vérification formelle de ces résultats sur machine. À l'exception des résultats de normalisation forte, cela a été fait en Lego par Pollack [56]. Un démarche similaire pour le Calcul des Constructions, avec cette fois la preuve de normalisation forte est décrite dans [5]. Le chapitre 4 est une extension de ce travail au cas des PTS avec sous-typage.

Les sortes de Coq

Le système Coq possède une hiérarchie d'univers prédicatifs indexée par des entiers (Type_i), ainsi que deux sortes imprédicatives (Prop et Set), dont le type est Type_0. En première approximation, on pourra confondre Prop et Set. La raison de cette distinction est de préciser le statut des objets vis-à-vis de l'extraction : les objets déclarés dans Prop seront considérés comme non informatifs et seront effacés lors de l'extraction, alors que les objets de Set seront gardés.

Cependant, comme il est parfois difficile de prévoir dans quel univers doit être déclaré un type, l'indice de Type est omis. Le système engendre alors une variable d'univers, lorsque l'on forme un produit, des contraintes sur ces variables sont engendrées, et le système s'assure en permanence qu'il existe une solution. L'exemple ci-dessous montre comment les termes de bases (l'entier 0) est typé, et comment les types sont typés par des sortes, et enfin comment se typent les sortes.

```
Coq < Check 0.
0
     : nat

Coq < Check nat.
nat
     : Set

Coq < Check Set.
Set
     : Type
```

La proposition `True` possède une preuve appelée `I` :

```
Coq < Check I.
I
     : True

Coq < Check True.
True
     : Prop

Coq < Check Prop.
Prop
     : Type

Coq < Check Type.
Type
     : Type
```

La dernière réponse pourrait laisser croire que l'on est dans le système incohérent Type : Type, mais on peut vérifier que le même univers (`t`) ne peut être appliqué à une fonction dont le domaine est dans un univers qui ne le contient pas (par exemple `t` lui-même) :

```
Coq < Definition t := Type.
t is defined

Coq < Check (t : :t).
Error during interpretation of command :
Check (t : :t).
Error : Universe Inconsistency
```

2.2.4 Représentation des structures de données

Dans les langages de programmation typés, on écrit des fonctions, mais on a souvent besoin de définir des types de données. Le λ-calcul rend bien compte de cette notion d'algorithme. Il nous reste encore à voir comment incorporer ces types de données à notre formalisme. Il existe plusieurs manières de procéder, mais celle qui paraît la plus commode est d'ajouter des types inductifs.

De par l'isomorphisme de Curry-Howard, il ne faut pas comprendre le terme "type de données" du strict point de vue informatique. Le but ici n'est pas uniquement de définir les entiers, les listes, les arbres, etc. Il sera possible, avec le même mécanisme, de définir des objets logiques comme les connecteurs logiques, des dérivations (la contrepartie logique des arbres), des prédicats ou des structures algébriques. Nous verrons quelques exemples.

Axiomatisation

La solution la plus simple consiste à axiomatiser, en ajoutant un axiome pour chaque objet ou propriété que l'on veut introduire. Cette méthode permet de rajouter n'importe quel type dans le système.

Le principal danger est que l'on risque de rajouter des axiomes qui rendront le système incohérent. Chaque fois que l'on rajoute un axiome, il faudrait refaire la preuve de cohérence du système pour s'assurer que l'on reste dans un système où l'on ne peut pas prouver l'absurde. Il peut être très compliqué de montrer la cohérence des axiomes d'un développement complet.

Pour illustrer les autres inconvénients de l'axiomatisation, prenons un exemple. Nous aurions pû introduire dans Coq le type des entiers naturels de la manière suivante :

```
Coq < Axiom nat : Set.

Coq < Axiom O : nat.

Coq < Axiom S : nat->nat.

Coq < Axiom nat_ind :(P :nat->Prop)(P O)->((n :nat)(P n)->(P (S n)))->(n :nat)(P n).

Coq < Axiom nat_rec : (P :nat->Set)(P O)->((n :nat)(P n)->(P (S n)))->(n :nat)(P n).
```

Nous avons introduit un nouveau type `nat`, et les deux constantes zéro et successeur (`O` et `S`), qui permettent d'engendrer tous les entiers d'une manière unique (pour un entier donné, il n'existe qu'une seule manière de le représenter avec `O` et `S`, ce qui ne serait pas le cas avec 0, 1 et +). Il reste ensuite à entrer les axiomes de Péano, dont le schéma de récurrence (`nat_ind`), qui permet les raisonnements par récurrence, et qui garantit que tout entier est soit zéro, soit le successeur d'un autre entier.

La constante `nat_rec` permet d'écrire des fonctions définies par récurrence structurelle sur les entiers. L'idée est que (`nat_rec` P f_0 f_S) est la fonction F qui associe f_0 à zéro et (f_S n (F n)) à (`S` n). Autrement dit, on voudrait associer des règles de réduction à cette

constante :

$$(\texttt{nat\_rec}\ P\ f_0\ f_S\ \texttt{O}) \to f_0$$
$$(\texttt{nat\_rec}\ P\ f_0\ f_S\ (\texttt{S}\ n)) \to (f_S\ n\ (\texttt{nat\_rec}\ P\ f_0\ f_S\ n))$$

L'addition se définirait à l'aide des équations $0 + n = n$ et $S(k) + n = S(k + n)$, soit :

```
Coq < Definition plus := [m,n :nat](nat_rec [_]nat n [k,pk](S pk) m).
```

Le problème est que l'on ne peut pas calculer avec `nat_rec`. Ce dernier est un axiome et ne se comporte pas comme le voudraient les règles de réduction ci-dessus. Notamment, (`plus O` n) ne se réduit pas vers n. Tout ce qu'on peut faire, c'est continuer l'axiomatisation ainsi (en supposant que l'égalité a déjà été axiomatisée) :

```
Coq < Axiom nat_rec_O : (P :nat->Set)(x :(P O))(f :(n :nat)(P n)->(P (S n)))
Coq <       (nat_rec P x f O)=x.

Coq < Axiom nat_rec_S : (P :nat->Set)(x :(P O))(f :(n :nat)(P n)->(P (S n)))(n :nat)
Coq <       (nat_rec P x f (S n))=(f n (nat_rec P x f n)).
```

Maintenant, on peut prouver que $0 + 0 = 0$, mais ce n'est pas de la conversion, il faut donc produire une preuve à chaque fois que l'on veut simplifier une expression. Cela risque d'inonder (et même noyer) le terme-preuve avec des trivialités.

Un dernier défaut de l'axiomatisation est qu'elle brise la possibilité d'extraire des programmes des preuves : poser `nat` comme un axiome ne dit pas concrètement vers quel type du langage cible il doit être extrait.

Codages imprédicatifs

Un autre moyen est de coder nos données en utilisant ce qui existe déjà dans le formalisme, à savoir les fonctions. Pour reprendre l'exemple des entiers, on peut coder n comme la fonction qui prend en argument une fonction f et un objet x et qui itère n fois f sur x, c'est-à-dire $f^n(x)$. Il s'agit des entiers de Church. Si l'on utilise ce codage dans le λ-calcul simplement typé, on est obligé de fixer le type de x (et donc aussi celui de f). Pour pouvoir définir l'addition, il faut que ce type (notons le τ) soit le type des entiers lui-même ($n + m$ se calcule en itérant n fois la fonction successeur sur m). Et comme le type des entiers fait appel à τ, cela n'est pas possible.

Dans un système imprédicatif (où l'on a des types polymorphes) comme le système F, cela est possible, c'est pourquoi on appelle ce codage des données un *codage imprédicatif*. Nous pouvons instancier les premiers axiomes de la section précédente :

$$\begin{aligned}
\texttt{nat} &\stackrel{\text{def}}{\equiv} \forall X.\, X \to (X \to X) \to X \\
\texttt{O} &\stackrel{\text{def}}{\equiv} \lambda x.\lambda f.x \\
\texttt{S} &\stackrel{\text{def}}{\equiv} \lambda n.\lambda x.\lambda f.(f\ (n\ x\ f))
\end{aligned}$$

Dans ce système, nous pouvons définir toutes les fonctions de l'arithmétique d'ordre deux, notamment l'addition :

$$\texttt{plus} \stackrel{\text{def}}{\equiv} \lambda m.\lambda n.(m\ n\ \texttt{S})$$

Cette fois, nous pouvons constater que notre définition de l'addition permet de calculer : (`plus O` n) se réduit vers n.

Mais ce codage présente de sérieux défauts pour l'utiliser dans un système de preuves :

- Certains axiomes de Péano ne sont toujours pas prouvables : l'axiome de récurrence `nat_ind` ne peut pas se prouver. Sa contrepartie calculatoire `nat_rec` non plus, ce qui peut rendre l'extraction délicate. De même pour $0 \neq 1$.
- Les codages imprédicatifs ne sont pas très pratiques à manier. Par exemple, le calcul de la fonction prédécesseur ne peut pas se faire en temps constant. Si on itère n fois la fonction $(m, n) \mapsto ((\mathtt{S}\ m), m)$ sur le couple $(\mathtt{O}, \mathtt{O})$, alors on obtient un couple dont la deuxième composante est le prédécesseur de n. La complexité est donc de $O(n)$, ce qui n'est pas très brillant.
- Lorsque l'on définit certaines structures complexes ayant des arguments fonctionnels, on définit un type pour lequel il existe des modèles dans lesquels il y a plus d'éléments que ce qu'on attendrait (voir [48]).

De toute façon, les codages imprédicatifs donnent lieu à des programmes engendrés par extraction peu efficaces : coder les données par des fonctions n'est pas viable en informatique, pour plusieurs raisons :

- Dans beaucoup de langages (comme Pascal ou C), les fonctions ne sont pas des objets de première classe : elles sont compilées, et on ne peut en créer au cours de l'exécution.
- Même dans les langages d'ordre supérieur, comme le code est compilé, l'opération de filtrage (comme par exemple le calcul du prédécesseur pour les entiers) est très coûteuse.

Types inductifs

Les types inductifs sont un moyen de spécifier des types de données de la même manière que l'on définit les types concrets en ML ou de manière plus rudimentaire en C. Cela systématise en quelque sorte l'introduction de constantes axiomatisant les entiers comme cela est fait dans le système T de Gödel.

Dans un langage non typé comme LISP, il suffit de rajouter l'opérateur de paire pour obtenir toutes les structures de données souhaitées : de taille bornée comme les enregistrements, ou de taille arbitraire comme les listes ou les arbres.

Lorsque l'on dispose d'un langage typé, la paire permet de construire un type uniquement à partir de types déjà existants. Or, dans le cas de la liste, le deuxième argument de `cons` devrait aussi être de type liste. Mendler [42] introduit un opérateur de type (noté μ) qui permet de construire des types par point fixe.

Si l'on suppose que l'on a déjà défini un type 1 dont l'unique élément est `tt`, ainsi que l'opérateur de type somme $+$, dont les constructeurs sont `inl` et `inr`, qui injectent respectivement les types A et B dans $A + B$, nous définissons le type des entiers comme

$$\mathtt{nat} \stackrel{\text{def}}{\equiv} \mu X.(1 + X)$$

L'opérateur de type μ possède un seul constructeur ι qui permet de "replier la définition récursive". Pour toute expression de type paramétrée F, ι a le type $F(\mu X.F(X)) \to \mu X.F(X)$. Les constructeurs des entiers se définissent à l'aide de ι.

$$O \stackrel{\text{def}}{\equiv} \iota(\mathtt{inl}(\mathtt{tt}))$$

$$S \stackrel{\text{def}}{\equiv} \lambda n{:}\mathtt{nat}.\,\iota(\mathtt{inr}(n))$$

Pour éviter d'avoir à introduire l'opérateur somme, celui-ci peut être intégré à notre opérateur. Ainsi, on ne pourra former $\mu X.\,F(X)$ que dans le cas où $F(X)$ est de la forme

$C_1(X)+\ldots+C_n(X)$, ce qui correspond aux types concrets de ML. Il y a alors n constructeurs $\iota_1 \ldots \iota_n$ tels que

$$\iota_i : C_i(\mu X.\, F(X)) \to \mu X.\, F(X)$$

Cela évite d'avoir les deux "étages" de constructeurs, d'où une économie de mémoire pour la représentation des données.

Il y aussi un opérateur pour destructurer, qui est la réciproque du constructeur. Cette opération est appelée filtrage en ML. Il permet de récupérer les sous-expressions. Nous le notons Case et il a le type $\mu.X.\, F(X) \to F(\mu X.\, F(X))$.

Dans le cas où l'on intègre la somme de types au point fixe, l'opération de filtrage est différente puisque l'on doit destructurer la somme en même temps que le constructeur, et l'objet obtenu peut avoir des types différents suivant i. Cela nous donne un opérateur de filtrage comme en ML :

$$\text{Case } M \text{ of } f_1 \mid \ldots \mid f_n \text{ end}$$

où les f_i ont le type $C_i(\mu X.\, F(X)) \to T$, le type des objets construits par ce filtrage étant T. Dans un système avec types dépendants, on peut donner plus d'information : lorsque l'on est dans la i-ème branche, on sait que l'objet M déstructuré est de la forme $(\iota_i\ N)$. Ainsi toutes les branches ne retournent pas un objet du même type. Si chaque f_i est de type $\Pi x\!:\!C_i(\mu X.\, F(X)).\, (Q\ (\iota_i\ x))$, alors l'expression de filtrage aura le type $(Q\ M)$. C'est ce qu'on appelle l'élimination dépendante.

Ces types récursifs permettent de définir les listes, les arbres à branchement fini. Ils permettent même de définir des arbres à branchement infini (mais de hauteur toujours finie dans le cas des types inductifs) en autorisant des arguments fonctionnels. Mais à cause de l'imprédicativité, il faut prendre des précautions pour ne pas perdre la bonne fondation de ces structures, ce qui pourrait mettre à mal la cohérence.

Dans le cadre d'un système logique, il n'est pas possible d'introduire les types récursifs de manière aussi générale, car il conduit à des incohérences. Le type $\mu X.\, X \to X$, correspondant à la définition de type Objective Caml suivante

```
type term = Lam of (term -> term)
```

qui permet de définir n'importe quel λ-terme, y compris l'opérateur de point fixe Y ou Ω, qui ne sont pas normalisables.

$$\begin{aligned} \mathit{App}(M,N) &\stackrel{\text{def}}{\equiv} ((\text{Case } M)\ N) \\ \Delta &\stackrel{\text{def}}{\equiv} \texttt{Lam}(\lambda x.\, \mathit{App}(x,x)) \\ \Omega &\stackrel{\text{def}}{\equiv} \mathit{App}(\Delta,\Delta) \end{aligned}$$

On peut aussi construire un paradoxe. La raison essentielle est que l'on peut écrire des fonctions "qui ne terminent pas" comme l'exemple trivial en ML

```
let rec f x = f x
```

De manière plus rigoureuse, cela revient à dire que l'on a défini une fonction partielle. Or, nous avons fait remarquer que pour ne pas briser la cohérence, la sémantique de Heyting concernant l'implication requérait que les fonctions soient totales. Dans notre exemple, le paradoxe est flagrant puisque cette fonction a le type $\forall\alpha,\beta.\, \alpha \to \beta$. Ce type vu comme une proposition est équivalent à l'absurde.

Une façon intuitive de voir que cet opérateur de point fixe conduit à des paradoxes est que le constructeur et le filtrage forment une bijection entre les type X et $F(X)$, pour un certain X. Clairement, cela n'est pas possible pour certains F.

Pour éviter ce paradoxe, nous n'acceptons les points fixes que sur les expressions dont le type a une certaine forme : les définitions "positives". La condition de positivité vient encore restreindre la forme de $F(X)$ en exigeant que les occurrences de X dans les C_i soient positives, ce qui signifie que si l'on a un type X_1 inclus dans un autre type X_2, alors $C_i(X_1)$ est inclus dans $C_i(X_2)$ (les C_i sont monotones croissants). Cela nous garantit l'existence d'un plus petit point fixe, qui est le type que l'on veut définir.

Nous ne donnerons pas plus de détails sur cette condition de positivité pour l'instant (ce sera l'objet des chapitres 6 et 7). À ce stade, nous avons ce que l'on appelle généralement les types inductifs comme ils sont implantés en Coq (si ce n'est qu'il n'y a pas de types inductifs anonymes). L'exemple des entiers naturels donne lieu à la déclaration inductive suivante :

```
Coq < Inductive nat : Set :=
Coq <     O : nat
Coq <   | S : nat->nat.
```

Cette déclaration fait plusieurs choses :

- Elle introduit un type appelé `nat` qui habite la sorte Set.
- Elle introduit deux constantes `O` et `S` appelées constructeurs car elles permettent de créer de nouveaux objets du type nat.
- Elle offre la possibilité de faire du filtrage sur les objets de type `nat`
- Elle permet de définir des fonctions définies par récurrence structurelle sur un entier puisque nous savons que tous les entiers sont bien fondés.

Fonctions définies par point fixe

De même que nous avons introduit le point fixe sur les types pour représenter des objets de taille arbitrairement grande, il nous faut introduire un opérateur de point fixe pour définir des fonctions parcourant ces objets. En effet, l'opérateur de filtrage Case ne permet de destructurer que d'un seul niveau de constructeur, et nous avons restreint la formation des types récursifs précisément pour éviter d'introduire un point fixe trop général.

Un point fixe peut se noter $\mathrm{Fix}(f.M)$, où f est un nom de variable et M est une expression appelée *corps* du point fixe. Le corps du point fixe peut faire référence au point fixe grâce au nom f. La portée de cette variable est M : elle n'est pas visible de l'extérieur.

L'idée est que l'expression $\mathrm{Fix}(f.M)$ devra se comporter comme le programme infini $M\{f\backslash M\{f\backslash \ldots\}\}$, i.e. M dans lequel on a remplacé f par M, en répétant ce procédé infiniment. Nous gardons donc la notation $\mathrm{Fix}(f.M)$, et lorsque cela est nécessaire, nous ferons une expansion du point fixe en remplaçant f par l'expression de point fixe, qui pourra elle-même s'expanser à volonté. La règle de réduction du point fixe serait celle que l'on donne dans les langages de programmation comme Haskell ou Objective Caml :

$$\mathrm{Fix}(f.M) \to M\{f\backslash \mathrm{Fix}(f.M)\}$$

Telle quelle, cette règle brise la normalisation forte à partir du moment où f apparaît dans M, même si l'ordre des appels récursifs est bien fondé, comme par exemple une récurrence sur un entier naturel. Ainsi, outre cette condition de bonne fondation, il nous faut trouver une stratégie d'évaluation du point fixe qui termine[4].

Pour s'assurer que les points fixes terminent on va se limiter à des récurrences structurelles. Sauf cas pathologiques dus à l'imprédicativité, les types récursifs restreints par la

[4] Bien que la terminaison de l'expansion des points fixes ne soit pas une condition nécessaire pour montrer la cohérence d'un système logique et la décidabilité du typage (thèse de Philippe Audebaud, arbres de Böhm rationnels), cela reste le moyen le plus simple de montrer ces résultats, en bénéficiant des résultats de la section 2.1.3.

condition de positivité donnent lieu à des structures bien fondées, i.e. lorsque l'on voit les termes canoniques comme des arbres, toutes les branches ont une longueur finie (mais ils peuvent être infiniment branchants). Cela nous résout à la fois le problème d'avoir un ordre bien fondé et celui de trouver une stratégie :

- comme il n'y a pas de branche infinie, la relation de sous-terme (modulo conversion car il nous faut aussi considérer les termes non canoniques) est bien fondée, et il suffit de vérifier que les appels récursifs se font uniquement avec des sous-termes de l'objet inductif.
- on peut "garder" l'expansion des points fixes en ne l'autorisant que lorsque l'on a fait apparaître un constructeur dans la structure qui nous garantit la terminaison.

Plus concrètement, cela signifie que les points fixes ne permettront de construire que des fonctions (comme en ML) dont le domaine est un type inductif. À cause des types dépendants, on ne peut pas obliger que cet argument soit le premier. Ainsi le point fixe est annoté avec un entier n qui signifie que c'est le n-ième argument qui assure la terminaison. On appelle ce paramètre l'*argument récursif* du point fixe. Ensuite, la stratégie d'évaluation est qu'on ne peut expanser un point fixe que lorsque son argument récursif est sous la forme d'un constructeur.

```
Coq < Eval Compute in [n :nat](plus n n).
     = [n :nat]
        (Fix plus
           {plus [n0 :nat] : nat->nat :=
              [m :nat]Cases n0 of
                        O => m
                      | (S p) => (S (plus p m))
                      end} n n)
     : nat->nat

Coq < Eval Compute in [n :nat](plus (S (S n)) n).
     = [n :nat]
        (S
          (S
            (Fix plus
               {plus [n0 :nat] : nat->nat :=
                  [m :nat]
                   Cases n0 of
                     O => m
                   | (S p) => (S (plus p m))
                   end} n n)))
     : nat->nat
```

Dans le premier cas, l'expansion de point fixe n'a pas lieu car `plus` (qui est défini comme un point fixe dont le premier argument est récursif) est appliqué à une variable. En revanche, dans le deuxième example le point fixe s'expanse car il se retrouve appliqué à `(S (S n))` qui est sous forme de constructeur. L'expansion peut même se faire une deuxième fois, puis elle s'arrête car c'est `n` qui se retrouve appliqué au point fixe.

Autres exemples de types inductifs

On peut définir le produit cartésien à l'aide de types inductifs :

```
Coq < Inductive prod [A,B :Set] : Set :=
Coq <   pair : A->B->(prod A B).
```

On a vu que le produit cartésien était isomorphe à la conjonction. Comme en Coq, on fait la distinction entre Prop et Set, il faut dupliquer les définitions pour chaque sorte, et ce que l'on entend par conjonction est :

```
Coq < Inductive and [A,B :Prop] : Prop :=
Coq <   conj : A->B->(and A B).
```

Le constructeur `conj` correspond exactement à la règle d'introduction de la conjonction. Les règles d'élimination s'obtiennent à l'aide de l'opérateur de filtrage :

```
Coq < Check [A,B :Prop ; p :A/\B]Cases p of (conj a b) => a end.
[A,B :Prop ; p :(A/\B)]Cases p of (conj a _) => a end
     : (A,B :Prop)A/\B->A

Coq < Check [A,B :Prop ; p :A/\B]Cases p of (conj a b) => b end.
[A,B :Prop ; p :(A/\B)]Cases p of (conj _ b) => b end
     : (A,B :Prop)A/\B->B
```

Le connecteur existentiel est une paire, sauf que le type de la deuxième composante dépend de la première. Cela ne pose donc pas de problème :

```
Coq < Inductive ex [A :Set ; P :A->Prop] : Prop :=
Coq <   ex_intro : (x :A)(P x)->(ex A P).
```

Afin de permettre l'extraction, il n'est pas possible d'écrire la fonction qui calcule le témoin à partir d'une preuve d'existentielle car celle-ci est définie dans Prop, et disparaîtra lors de l'extraction alors que le type du témoin vit dans Set et est gardé.

```
Coq < Check [A :Set ; P :A->Prop ; e :(ex A P)]<A>Cases e of (ex_intro x p) => x end.
Toplevel input, characters 6-73
> Check [A :Set ; P :A->Prop ; e :(ex A P)]<A>Cases e of (ex_intro x p) => x end.
>       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error : Incorrect elimination of e in the inductive type
  (ex A P)
 The elimination predicate A has type Set
 It should be one of :
  (ex A P)->Prop, Prop

 Elimination of an inductive object of sort : Set
 is not allowed on a predicate in sort : Prop
 because non-informative objects may not construct informative ones.
```

Toutefois, la règle d'élimination de l'existentielle est dérivable :

```
Coq < Check ex_ind.
ex_ind
     : (A :Set ; P :(A->Prop) ; P0 :Prop)((x :A)(P x)->P0)->(ex A P)->P0
```

Pour prouver une proposition P_0 sachant que l'on peut prouver $\exists x : A.\, P(x)$, il suffit de prouver $\forall x : A.\, P(x) \to P_0$, ou encore P_0 dans un contexte où l'on a introduit une variable fraîche x et une preuve de $P(x)$, grâce à la règle d'introduction du produit.

Les types inductifs ne servent pas uniquement à représenter les structures de données et les connecteurs logiques. Il est possible de définir des prédicats de la même manière. Cela correspond à la manière de définir des prédicats en Prolog.

La relation "être plus petit que" est la plus petite relation $\leq$ telle que $n \leq n$ et si $n \leq m$, alors $n \leq S(m)$:

```
Coq < Inductive le : nat->nat->Prop :=
Coq <   le_n : (n :nat)(le n n)
Coq < | le_S : (n,m :nat)(le n m)->(le n (S m)).
```

Ici encore, les constructeurs correspondent aux règles d'introduction des instances du prédicat. Le système engendre automatiquement la règle d'élimination témoignant du fait que `le` est la plus petite relation satisfiant les deux clauses ci-dessus.

```
Coq < Check le_ind.
le_ind
     : (P :(nat->nat->Prop))
        ((n :nat)(P n n))
        ->((n,m :nat)(le n m)->(P n m)->(P n (S m)))
        ->(n,n0 :nat)(le n n0)->(P n n0)
```

On peut aussi définir des types inductifs mutuellement récursifs, comme par exemple le type des arbres d'arité quelconque, que l'on définit simultanément avec les listes d'arbres. Cet exemple est développé plus longuement dans [49].

2.2.5 Extraction

Le formalisme de Coq est intuitionniste et fortement normalisant, ce qui fait que le résultat de la section 2.1.3 sur l'élimination des coupures est vérifié. Rappelons que la seule règle d'introduction de l'existentielle est :

$$\frac{\Gamma \vdash \pi : P(t)}{\Gamma \vdash \langle t, \pi \rangle : \exists^\star x.\, P(x)} \ (\exists\text{-I})$$

On en déduit que toutes les preuves closes d'une formule existentielle peuvent se simplifier en une preuve qui se termine par l'application de cette règle, et donc on peut extraire le témoin t. D'autre part, on peut aussi récupérer une preuve que ce témoin t vérifie le prédicat P. Comme l'élimination des coupures correspond à l'évaluation d'un programme fonctionnel, on peut voir la preuve d'une existentielle comme un programme qui calcule le témoin. En logique classique, on perd ce résultat d'extraction en même temps que celui sur les preuves canoniques. C'est en cela que les preuves classiques sont dites *non constructives*.

La certification de programmes en Coq repose sur cette propriété (pour une description plus approfondie, voir [50]). On peut par exemple spécifier la division euclidienne par

$$\forall a \in \mathbb{N}.\, \forall b \in \mathbb{N}^*.\, \exists^\star q, r \in \mathbb{N}.\, (a = b.q + r \ \wedge \ r < b)$$

Une preuve de cette propriété sera donc une fonction qui a tout couple d'entiers (a_0, b_0) accompagné d'une preuve que b_0 n'est pas nul associe un couple d'entiers (q_0, r_0) et une preuve de $a_0 = b_0.q_0 + r_0 \ \wedge \ r_0 < b_0$. Le fait que b n'est pas nul est une précondition qui indique sous quelles conditions on a le droit d'utiliser la fonction, mais on ne voudrait pas donner la preuve. De même le fait que $a_0 = b_0.q_0 + r_0 \ \wedge \ r_0 < b_0$ est une post-condition, mais nous ne souhaitons pas extraire un programme qui calcule effectivement ces preuves.

Nous voulons pouvoir effacer les parties purement logiques qui se trouvent dans les preuves. Au moment où l'on déclare un type en Coq, nous devons décider si ce type représente des objets informatifs, ou calculatoires. Au moment de l'extraction, les objets logiques sont effacés et il ne reste plus que les objets informatifs. Cette distinction se fait en fonction de la sorte du type : les types de la sorte Set sont calculatoires, alors que les types de Prop sont non informatifs. Dans notre exemple, si l'on supprime les arguments logiques ($b > 0$ et $a = b.q + r \ \wedge \ r < b$), notre proposition devient $\mathbb{N} \to \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, qui est le type de la fonction extraite que nous voulons. Un résultat d'extraction nous garantit alors que cette fonction calcule bien le quotient et le reste.

Le fait de distinguer Prop et Set nous oblige parfois à dupliquer des définitions, comme par exemple l'existentielle. En Coq, il y a `ex` et `sig`, que nous notons dans notre méta-langage $\exists$ et $\exists^*$.

```
Coq < Print sig.
Inductive sig [A :Set; P :A->Prop] : Set :=
      exist : (x :A)(P x)->(sig A P)
```

En comparant cette définition avec celle de `ex` de la section précédente, on voit qu'elles sont définies de manière identique, prennent des argument identiques, mais `ex` vit dans Prop, alors que `sig` est dans Set. Cela signifie que le programme extrait ne calculera pas les temoins des existentielles $\exists$, mais uniquement ceux de $\exists^*$. Avec `sig`, il est possible de récupérer le témoin :

```
Coq < Check [A :Set ; P :A->Prop ; e :(sig A P)]Cases e of (exist x p) => x end.
[A :Set ; P :(A->Prop) ; e :((sig A P))](let (x, _) = e in x)
     : (A :Set ; P :(A->Prop))(sig A P)->A
```

Nous pouvons alors spécifier la fonction de division euclidienne, calculant le quotient et le reste, par la proposition (vivant dans la sorte Set) :

$$\forall a \in \mathbb{N}. \forall b \in \mathbb{N}^*. \exists^\star q, r \in \mathbb{N}. (a = b.q + r \ \wedge \ r < b)$$

Nous pouvons aussi spécifier la fonction qui ne calcule que le quotient en utilisant l'existentielle logique pour r :

$$\forall a \in \mathbb{N}. \forall b \in \mathbb{N}^*. \exists^\star q \in \mathbb{N}. \exists r \in \mathbb{N}. (a = b.q + r \ \wedge \ r < b)$$

Cette spécification est déjà prouvée dans les théories de Coq, et la commande `Write Caml File` permet d'extraire un programme Objective Caml de cette preuve :

```
Coq < Check quotient.
quotient
     : (b :nat)
       (gt b O)
       ->(a :nat){q :nat | (EX r :nat | a=(plus (mult q b) r)/\(gt b r))}

Coq < Write Caml File "div" [eucl_dev].
Warning : The constant nat_rec is expanded.
Warning : The constant sumbool_rec is expanded.
Warning : The constant le_gt_dec is expanded.
```

Le résultat est l'écriture d'un fichier source `div.ml` qui comporte le contenu calculatoire de `quotient` ainsi que toutes les objets dont il dépend. Ce fichier peut être directement compilé et exécuté.

Pour résumer, nous avons que la certification de programme en Coq se faisait :

1. En donnant une spécification de notre programme sous la forme d'une proposition.
2. En prouvant cette spécification.
3. En extrayant le contenu calculatoire de cette preuve, l'algorithme étant implicite dans la manière dont a été prouvée la spécification.

Il faut reconnaître que ce schéma n'est pas toujours le meilleur. D'une part parce que la plupart du temps, nous cherchons à vérifier des programmes qui existent déjà. Il pourrait être alors difficile de faire la preuve qui correspond à l'algorithme employé. Heureusement, pour cela, il existe la tactique `Program` [47] de Coq qui permet de faire automatiquement[5] les étapes de la preuve correspondant à un programme censé correspondre à la spécification. En plus, on peut imaginer donner en entrée un programme qui ne soit pas exprimé dans un langage de la famille ML (voir la thèse de Filliâtre [20]).

D'autre part, raisonner dans l'autre sens est parfois plus efficace, même sans parler de spécification de programmes. Avec un peu d'expérience, les théorèmes purement logiques peuvent eux aussi se comprendre comme des spécifications de programmes. L'utilisateur est souvent plus à l'aise pour concevoir un programme étant donné une spécification que pour

[5] Dans le cas des fonctions définies par récurrence, il faut parfois les annoter avec une sorte d'invariant.

concevoir les étapes d'une démonstration. Cela est particulièrement vrai pour les théorèmes se prouvant par récurrence. La démonstration par récurrence est quelque chose d'assez simple lorsqu'on l'utilise informellement. Mais formellement, lorsqu'il s'agit de donner clairement l'énoncé prouvé par récurrence, on se rend compte que l'on oublie souvent quelles sont les variables quantifiées et celles qui ne le sont pas. Ceci se produit moins souvent lorsque l'on écrit un programme par récurrence : on sait en général bien quels sont les arguments qui évoluent au cours des appels récursifs.

2.2.6 Le système de Coq

Le système de Coq est en quelque sorte la réunion de ces évolutions en Théorie des Types. La description du système est faite dans le manuel de référence. La définition des types inductifs implantés se trouve dans le mémoire d'habilitation de Paulin [49].

La principale contribution de cette thèse est la formalisation d'un système regroupant des éléments présentés dans cette section, mais il diffère un peu du système implanté par Coq. Les points présents dans Coq non formalisés sont :

- Les types co-inductifs [24], qui permettent de définir des objets de hauteur potentiellement infinie, ainsi qu'un mécanisme de point fixe (restreint afin de préserver la cohérence), permettant de construire de tels objets.
- Les types inductifs imbriqués ne sont pas pris en compte. Ils ne sont même pas autorisés.
- La satisfiabilité de contraintes d'univers gérant automatiquement la numérotation des univers n'est pas abordée dans ce travail.
- L'extraction n'a pas été formalisée, ce qui nous empêchera de faire effectivement le bootstrap.

Réciproquement, il y a des aspects formalisés qui n'apparaissent pas encore dans le système actuel :

- La possibilité d'avoir un système muni de sous-typage, même si dans le cas présent nous ne profiterons pas beaucoup de cette option.
- Le système de marques vérifiant la terminaison des points fixes, qui semble réparer un certain nombre de problèmes rencontrés avec la condition de garde implantée, notamment l'existence de termes non normalisants. Ce point assez technique sera détaillé ultérieurement.

Nous avons recensé dans cette section un certain nombres de propriétés souhaitables pour notre formalisme comme l'élimination des coupures, l'auto-réduction, ou l'extraction. La preuve de ces résultats s'appelle l'étude *métathéorique*. Cette étude ne s'inscrit pas à proprement parler dans le simple objectif de vérifier que l'implantation correspond bien au formalisme. On pourrait supposer les résultats métathéoriques et bâtir l'implantation par dessus.

Cependant, comme nous proposons un certain nombre de systèmes nouveaux, soit dans leur présentation, soit ayant réellement des fonctionalités supplémentaires, il est rassurant de partir sur de bonnes bases en procédant à l'étude métathéorique.

2.3 Les systèmes de preuve

2.3.1 Architecture

Les formalismes décrits dans la section précédente possèdent très peu de règles d'inférences, correspondant à des étapes de raisonnement très petites. Ceci rend le formalisme plus simple à assimiler lorsque l'on veut étudier ses propriétés. Le revers de la médaille est qu'il est très fastidieux d'écrire à la main des preuves dans ces formalismes. Leur taille est tellement grande qu'il devient même presque impossible d'écrire manuellement un terme-preuve d'un théorème conséquent sans faire d'erreur. Ces erreurs peuvent être fondamentales, comme par exemple ne pas comprendre la règle de typage du filtrage sur les types inductifs, ou dûes à la négligence de l'auteur qui oublie volontairement ou non des parties qui lui paraissent absolument évidentes, ou même une simple erreur de typographie.

Comme la tâche de vérifier la correction est fastidieuse mais ne nécessite pas d'intelligence, elle convient tout à fait à l'ordinateur. Nous allons donc chercher à construire des programmes capables de vérifier si une preuve écrite par l'utilisateur est correcte ou non. C'est le moins que nous ayons à faire pour nous convaincre qu'un théorème admet bien une preuve. Un tel système doit remplir certains critères :

- Il *doit* être sûr. Cela signifie que lorsque le programme accepte une preuve, celle-ci *doit* être une preuve correcte dans le formalisme que l'on prétend implanter. C'est la correction de l'implantation vis-à-vis du formalisme.
- Un critère souhaitable, bien que pas absolument nécessaire est la *complétude* de l'implantation, à savoir le fait que celle-ci ne rejette aucune preuve correcte dans le formalisme.
- L'efficacité est aussi une caractéristique importante. Il ne sert à rien d'avoir un système totalement générique, capable de résoudre une classe de problèmes faramineuse s'il lui faut plusieurs années pour valider la preuve de correction d'un programme, car ce dernier risque d'être obsolète le jour où l'on aura la réponse.
- L'ergonomie du système doit être prise en compte de manière à ce que l'utilisateur puisse développer sa preuve dans des conditions relativement proches de celles qu'il connait habituellement.

Il pourrait paraître naturel que le critère de sûreté soit toujours considéré comme prioritaire. Cependant, comme la certitude absolue n'existe pas, on fait toujours un compromis entre sûreté, efficacité et ergonomie.

Une mauvaise approche serait d'avoir une panoplie d'outils ayant différents degrés de sûreté et d'efficacité, et de choisir celui adapté à nos besoins. Le risque est de construire une sorte de Tour de Babel où il devient difficile de communiquer des résultats prouvés dans des systèmes différents.

Un système qui serait capable de se placer au niveau souhaité est de loin préférable. Or, un système rapide mais peu sûr est condamné à le rester, et l'on ne peut rien faire pour améliorer sa fiabilité, si ce n'est recommencer l'implantation sur des bases plus saines. Notre objectif sera plutôt d'essayer de déterminer comment repousser au maximum les limites de la sûreté. Tout ce qui peut augmenter la sûreté doit être pris en compte. Si cela remet en cause l'efficacité de manière dramatique, on peut concéder un peu de sûreté, on en saura au moins un peu plus à propos des hypothèses sur lesquelles repose le système.

Notre principe est qu'une bonne architecture doit pouvoir permettre d'augmenter significativement l'efficacité si l'on accepte de perdre en sûreté[6]. Le fait de pouvoir ajouter des axiomes dans un développement permet de se placer dans un formalisme où les preuves

[6] Si cela ne rend pas le système moins sûr, cela s'appelle une optimisation.

sont plus courtes (et eventuellement prouver plus de propositions). Ainsi, il est possible de raisonner en logique classique en Coq simplement en posant l'axiome

```
Coq < Axiom NNPP : (P :Prop)~~P->P.
```

Le tiers exclus est une conséquence de cet axiome. Pour raisonner encore plus librement, on peut vouloir identifier les propositions équivalentes :

```
Coq < Axiom IMP_ANTISYM_AX : (P,Q :Prop)(P->Q) -> (Q->P) -> P==Q.
```

L'une des conséquences de cet axiome est la non pertinence des preuves, i.e. deux preuves d'une même proposition sont égales. Comme dernier exemple, l'extensionalité est aussi un axiome qui permet de couper court dans certaines preuves.

```
Coq < Axiom EXT_AX : (A,B :Set)(f,g :A->B)((x :A)(f x)=(g x)) -> f=g.
```

Mais il faut reconnaître que peu de travail a été fait en Coq pour profiter de ces axiomes lorsqu'ils sont posés.

Spécification de l'architecture

Le meilleur moyen pour garantir une grande sécurité est de formaliser et prouver les algorithmes employés. La formalisation complète d'un système est quelque chose de complexe, et c'est ce que nous essayons de faire dans cette thèse. Dans un premier temps, on pourrait se satisfaire de la spécification de l'implantation. Les avantages de la spécification sont nombreux :

- c'est un premier pas vers la formalisation. Cela oblige à formuler clairement le rôle de chacune des fonctions. On prend un peu de hauteur et on voit les choses de manière un peu plus abstraite et générale.
- Cela permet de dégager des *composants* ou modules indépendants, ce qui facilite leur réutilisation d'une version à l'autre. L'on définit alors vraiment ce qu'est l'architecture du système, i.e. les dépendances entre modules.
- L'efficacité peut être améliorée en évitant les vérifications redondantes qui peuvent avoir des conséquences dramatiques sur la complexité des algorithmes. C'est ce qu'on appelle généralement un style de programmation moins *défensif*.

On voit que l'effort de spécification nous fait gagner sur tous les plans : une plus grande confiance, une implantation plus efficace et en général aussi une meilleure ergonomie, du fait d'un comportement plus uniforme et plus simple à expliquer.

Architectures avec noyau

Le principe de de Bruijn conseille l'emploi de formalismes pour lesquels chaque personne puisse se convaincre par elle-même de la validité de la preuve d'un théorème (voir aussi la discussion en conclusion de la thèse de Pollack [56]). Cette conviction devrait pouvoir s'acquérir sans l'aide d'un outil fourni par un tiers auquel on devrait faire confiance. Si l'on n'est pas capable de décider tout seul de la validité, on doit pouvoir être capable d'écrire un programme qui le fera à notre place.

Pour satisfaire cette exigence, le plus simple semble d'avoir un moyen de représenter les preuves. Ces termes-preuves peuvent être facilement échangés. Il est alors en général assez facile d'écrire un vérificateur de preuve de petite taille. Notamment, tout ce qui a rapport aux moyens de construire la preuve peut être éludé. On aboutit à une architecture avec *noyau*, qui identifie deux composantes :

- L'interface avec l'utilisateur, qui peut être rudimentaire ou au contraire offrir des fonctionalités puissantes permettant de construire automatiquement des preuves

– Le noyau, dont le seul (mais capital) travail est de vérifier que les preuves engendrées par l'interface sont correctes.

Le point-clé est que la sûreté du système repose exclusivement sur le noyau. Si l'interface est erronée et engendre une preuve comportant des erreurs, alors le noyau le signalera.

Cette séparation illustre bien la manière dont un humain développe une preuve : il y a tout d'abord un niveau informel où l'on recherche la preuve en utilisant des heuristiques ou des intuitions. Cela permet de passer au niveau suivant, plus formel, ou l'on écrit avec attention la preuve imaginée, cette fois en vérifiant chaque étape soigneusement. Ici, l'interface peut faire appel à des algorithmes très complexes. Il semble difficile de garantir que la preuve engendrée soit correcte. C'est pourquoi il est plus sûr de séparer l'étape de construction et celle de validation d'une preuve.

Le problème est qu'en général, on ne construit pas une preuve de la même manière qu'on la vérifie : la recherche d'une preuve se fait en décomposant le problème initial en sous-problèmes que l'on saura résoudre plus facilement. Lors de la vérification, on construit des théorèmes de plus en plus complexes, chacun se déduisant par étape élémentaire de ceux déjà établis. Cela nous conduit en général à une architecture à deux passes.

Or, un principe important d'ergonomie est qu'il est préférable d'informer le plus tôt possible l'utilisateur des erreurs qu'il commet : il est très frustrant d'écrire une longue preuve et se rendre compte à la fin qu'une erreur située au début rend le reste de la preuve inutile. Cela signifie que l'on fera des vérifications aussi au moment où l'on recherche la preuve. Or, ce travail sera refait par le noyau.

Cela peut sembler inefficace à première vue. Mais il faut bien se rendre compte que le fait d'avoir un noyau apporte beaucoup à la sûreté du système. Sachant que le noyau revérifiera la preuve produite, cela permet de développer les algorithmes de recherche de preuve avec une plus grande liberté, ce qui peut conduire à des implantations plus efficaces que si elles avaient dû être garanties sans défauts.

Une possibilité pour éviter ces vérifications un peu redondantes serait d'avoir un système de types plus complexe avec métavariables et substitutions explicites, comme proposé par Muñoz dans sa thèse [43]. L'avantage est qu'alors le mécanisme de construction de la preuve est certifié correct, et l'on a évité une phase distincte de validation de la preuve. Mais il ne faut pas perdre de vue que pour respecter le principe de de Bruijn, ce système ne doit pas être trop complexe. Dans cette thèse, nous en resterons à un système sans métavariables.

Une autre architecture : LCF

Une idée originale du système de preuves LCF [28] est de définir un type abstrait des théorèmes. Les seules fonctions permettant de construire de nouveaux théorèmes sont les règles logiques du formalisme, prennant en argument les théorèmes correspondant aux prémisses d'une règle, et retournant le théorème correpondant à la conclusion de cette même règle. L'utilisation d'un langage fortement typé comme ML nous garantit qu'il n'existe pas de moyen détourné pour construire un théorème, que d'utiliser les règles élémentaires du formalisme.

Avec cette architecture, on distingue bien la recherche de la preuve de sa construction. Comme pour les systèmes avec métavariables, la preuve est correcte par construction. En revanche, il y a quand même une duplication des vérifications, puisque les théorèmes ne peuvent se construire que dans le sens direct, c'est-à-dire en n'utilisant que des théorèmes déjà prouvés, ce qui ne convient pas lorsque l'on développe les preuves à l'aide de tactiques. C'est ce que nous verrons dans une section prochaine.

Termes-preuves

Le fait de manipuler explicitement les termes de preuves est un choix d'implantation qui peut augmenter la sûreté du système. Certes, il s'agit d'un handicap, puisqu'il faut non seulement (dans le style de LCF) décomposer toute preuve en l'application des règles élémentaires du formalisme, mais en plus il faut construire explicitement l'objet, ce qui peut poser des problèmes de taille mémoire.

Mais d'un autre côté, le fait d'avoir un langage de preuve rend possible la communication de ces preuves, et l'on peut tout à fait imaginer la réalisation de programmes accompagnés d'une preuve des propriétés qu'il vérifie. Le destinataire peut alors vérifier par lui-même (*proof carrying code*) la validité de la preuve puisque le format de celle-ci peut être rendu public.

L'absence de termes-preuves dans un système comme HOL, le descendant le plus proche de LCF, fait qu'il ne satisfait pas tout à fait le principe de de Bruijn, puisqu'il n'est pas possible de dissocier la preuve de la manière dont elle a été obtenue, ce qui requiert le système dans son intégralité. Ce système n'en est quand même pas loin puisque l'on peut choisir de garder l'enchainement de règles élémentaires simplement en changeant l'implantation du type des théorèmes. Le fait que ce type soit abstrait rend la chose d'autant plus facile.

Mais l'absence de termes-preuve pose encore un problème d'ordre pratique. Dans un système admettant un nombre très restreint de principes primitifs, les notations courantes en mathématiques sont dérivées. Il serait évidemment lourd de devoir débuter chaque session en refaisant toutes les preuves de ce prélude. C'est pourquoi l'on a souvent un moyen de charger un ensemble de théorèmes depuis un disque, sans faire de vérifications. Le problème est que cela introduit un nouveau moyen de construire un théorème, sans passer par les règles élémentaires. Notamment, le disque aurait pu être altéré et ainsi contenir un théorème absurde. En l'absence de termes-preuve, le seul remède pour garantir la cohérence est de sauvegarder en même temps le script de preuve qui a engendré le théorème. On refait donc l'étape de recherche de la preuve. Le terme de preuve peut être vu comme une forme compilée du script de preuve, où la phase de recherche est déjà résolue.

2.3.2 Description du noyau

Nous avons défini le noyau d'un système de preuves comme la partie chargée de s'assurer que la preuve esquissée informellement est correcte. Dans le cas des systèmes ayant des termes-preuves, cela est particulièrement aisé, le langage des preuves étant en général conçu de manière à lever toute ambiguïté.

Le but de cette section n'est pas de décrire précisément ce qui se passe à l'intérieur du noyau, car cela dépend fortement du formalisme employé. Les parties 2 et 3 de cette thèse tenteront de répondre à cette question en proposant des systèmes de types dont on prouvera la métathéorie, ainsi que la décidabilité du typage. En revanche, nous définirons de manière relativement générale ce que doit être l'interface de ce noyau. Il est important que cette interface soit aussi indépendante du formalisme que possible, car cela fournit une base solide pour développer l'interface, sans que tout soit remis en cause par une modification mineure du formalisme.

De manière abstraite (et rudimentaire), un vérificateur de preuve est un programme qui prend en entrée une proposition ainsi qu'un candidat preuve, et qui répond si c'est effectivement une preuve de la proposition. Avec l'isomorphisme de Curry-Howard, il n'y a donc besoin que d'une seule opération : la vérification de type. Le résultat métathéorique derrière cette opération est la décidabilité du typage. Ce qui revient à dire que notre but

est de prouver la spécification suivante :

$$\forall \Gamma, M, T.\quad \mathsf{Dec}\,(\Gamma \vdash M : T)$$

Le séquent $\Gamma \vdash M : T$ se comprenant comme la proposition "il existe une dérivation complète dont la conclusion est le séquent $\Gamma \vdash M : T$". Cette formule résume bien notre objectif, mais l'utilisabilité de notre système nous pousse à la raffiner un peu.

Lorsque le typage échoue, l'utilisateur aurait grand besoin d'un message d'erreur, c'est-à-dire une information qui lui dise à quel endroit la vérification de la preuve a échoué, afin de pouvoir chercher plus efficacement le moyen de réparer cette erreur.

Aussi, on souhaiterait pouvoir vérifier la construction de la preuve d'un théorème de manière un peu plus incrémentale. Nous avons déjà fait remarquer à quel point il peut être frustrant de ne faire aucune vérification au cours de la construction de la preuve. Pour éviter ce genre de mésaventure, on aura tendance à faire appel au noyau pour vérifier de temps en temps la correction de la preuve que l'on construit. Pour des raisons d'efficacité évidentes, on voudrait éviter de re-vérifier sempiternellement le début du développement.

On considèrera un système de preuves comme une machine abstraite, dont l'état est le développement qui a été vérifié jusqu'à présent, et l'on dispose de commandes qui permettent de faire évoluer cet état vers la construction du théorème souhaité.

L'étude formelle cette machine abstraite, ainsi que la formalisation des messages d'erreurs sera l'objet de la section 2.4. Pour l'instant, nous dirons simplement que l'on peut modéliser l'état courant à l'aide d'un contexte bien formé, dans la mesure où celui-ci peut contenir soit des hypothèses faites, soit des définitions permettant d'associer un nom à des termes, pour modéliser les lemmes intermédiaires. Ainsi, la notation $\Gamma\,[x{:}T]$ est le contexte Γ étendu avec une hypothèse nommée x dont l'énoncé est T, et $\Gamma\,[x = M{:}T]$ est le contexte Γ étendu avec la définition de x dont la valeur est M et dont le type est T.

Suivant ces considérations, nous pouvons alors voir que notre objectif initial peut se décomposer, donnant lieu à une nouvelle spécification de l'interface du vérificateur de type. Celle-ci comporte plusieurs fonctions, et nous emploierons la notation des enregistrements pour la définir :

$$\begin{array}{lrlll}
\langle\ \texttt{infer}: & \forall \Gamma, M, T. & \Gamma \vdash \Rightarrow\ \mathsf{Dec}\,(\Gamma \vdash M : T)\,; & \text{(* vérification de type *)} \\
\quad \texttt{add_var}: & \forall \Gamma, x, T. & \Gamma \vdash \Rightarrow\ \mathsf{Dec}\,(\Gamma\,[x{:}T] \vdash)\,; & \text{(* ajout d'une hypothèse *)} \\
\quad \texttt{add_def}: & \forall \Gamma, x, M, T. & \Gamma \vdash \Rightarrow\ \mathsf{Dec}\,(\Gamma\,[x{=}M{:}T] \vdash) & \text{(* ajout d'un lemme *)} \\
\rangle &&&
\end{array}$$

La spécification initiale est très facilement dérivable : il suffit d'appliquer les algorithmes `add_var` ou `add_def` sur chacune des variables du contexte de la preuve à construire, puis un appel à `infer` permet de vérifier la preuve du théorème principal.

Le noyau devrait aussi offrir des fonctions utiles pour écrire des tactiques, comme par exemple une fonction qui fait de la vérification de type sachant que le type proposé est bien formé, ou encore une fonction qui calcule le type d'un terme que l'on sait bien typé (i.e. reconstruire le type sans faire de vérifications). Ces fonctions permettent une implantation plus efficace, car elles font moins de vérifications. Cependant, il ne faut pas abuser de ce genre de fonctions, car cela risque d'obliger à exporter du noyau des fonctions auxiliaires, ce qui peut rendre l'interface du noyau plus sensible aux modifications du formalisme.

2.3.3 Les tactiques

Comme nous l'avons déjà remarqué plusieurs fois, la construction des termes-preuves est difficilement faisable à la main sans aide. D'une part à cause de sa taille, dû au degré de détail requis, mais aussi parce que ça ne correspond pas toujours à la façon de raisonner en concevant en premier les étapes qui devraient avoir lieu en dernier, si l'on veut respecter l'ordre dans lequel on déduit les faits.

Dans le système LCF est apparue pour la première fois la notion de *tactique*. Le principe est de raisonner "à l'envers". On se fixe un *but*, i.e. une proposition que l'on souhaite prouver. En général, la forme de ce but nous donne une idée des dernières étapes de la preuve. On emploie alors une tactique, qui permet d'engendrer des sous-buts. Si l'on arrive à prouver ces sous-buts, alors la tactique permet de construire la preuve du but initial à partir des preuves des sous-buts.

```
type tactic = goal -> (goal list * (proof list -> proof))
```

Cette définition en Objective Caml fait bien apparaître qu'une tactique est une fonction prenant en argument un but, et retourne une liste de sous-buts, ainsi qu'une *validation*. Cette validation est la fonction qui construira le morceau de preuve correspondant à la tactique.

Dans notre cas, le type `proof` des preuves peut être tout simplement le type des termes, et un but peut être tout simplement la paire formée d'un contexte et de la proposition à prouver.

Exemple 2.2 *La règle d'introduction du quantificateur universel donne lieu à une tactique* `intro` *que l'on pourrait, en simplifiant, implanter de la manière suivante :*

```
let intro goal =
  let (x,ty,cl) = dest_prod (concl goal) in
    ([mk_goal (add_var (hyps goal) (x,ty)) cl],
     (fun [prf] -> mk_lam (x,ty,prf)))
```

Dans un premier temps, on décompose la conclusion du but qui doit être un produit (représentant l'implication et le quantificateur universel), sinon la tactique ne devrait pas être employée. Il ne reste plus qu'à former la liste des sous-buts, correspondant aux prémisses de la règle d'introduction du produit qui ne sont pas déjà complètement connues, ainsi que la validation permettant de construire la preuve à partir de la preuve de l'unique sous-but. Le constructeur de terme correspondant à l'introduction du produit est l'abstraction. Il suffit donc de rajouter un constructeur d'abstraction au sommet du terme pour obtenir une preuve du but initial.

Cette commande peut échouer au moment de son application, si le but n'est pas un produit. Mais aussi, si la fonction construisant la preuve est appliquée à une liste ne contenant pas un unique élément, ce qui dénote une erreur dans le système qui gère la production des preuves. Un autre problème serait que l'abstraction ne puisse pas être formée (par exemple si l'on se trouve dans système comme λP où tous les produits ne peuvent pas être formés. Comme le remarque Pollack dans [57], il serait bon de faire ce genre de vérifications au moment où l'on applique la tactique.

On peut associer une tactique à chacune des règles élémentaires. Si l'on n'utilise que ces tactiques, on peut construire n'importe quelle dérivation, exactement de la manière dont on la ferait sur un tableau. Pour augmenter la puissance du système, on va chercher à écrire des tactiques de plus en plus puissantes, notamment en utilisant les tactiquelles (ou *tacticals*) que nous allons énumérer plus bas. Il nous semble important de toujours donner à l'utilisateur la possibilité d'utiliser ces tactiques élémentaires, qui ont l'avantage d'être rapides et d'avoir un comportement simple.

Les *tacticals* sont des fonctions qui permettent de composer les tactiques. La liste suivante forme une base de tacticals qui permet déjà de construire des tactiques intéressantes.

T_1;T_2 : Applique T_1 au but courant, puis T_2 à chacun des buts engendrés par T_1. Une variante est $T; [T_1 \mid \ldots \mid T_n]$ qui applique T_i au i-ème but engendré par T.

`Repeat` **T** Applique T tant que cela est possible.

T_1 `Orelse` **T_2 :** Essaie d'appliquer T_1, mais si elle échoue (si elle ne s'applique pas, par exemple), alors applique T_2 au but courant.

`Idtac` engendre un sous-but identique à celui passé en argument. Il s'agit en fait d'une tactique, mais elle n'a d'intérêt que combinée avec les *tacticals*.

2.4 Formalisation d'une interface

Dans cette section, on s'intéresse à ce qui vient autour du noyau : l'interface avec l'utilisateur, i.e. la syntaxe concrète et la sémantique des commandes et des messages d'erreur. On suppose ici que l'on dispose d'un système avec un certain nombre de bonnes propriétés, comme l'existence d'un module de vérification de type et l'on décrira une formalisation[7] spécifiant comment le noyau peut être habillé de manière à former un système avec lequel on peut interagir à l'aide de commandes.

Messages d'erreur

Au lieu de spécifier la fonction de vérification de type par la spécification $\mathsf{Dec}\ (\Gamma \vdash M : T)$, qui s'extrait en une fonction retournant un booléen suivant que le jugement est dérivable ou pas, on définira un ensemble de messages d'erreur E, et la fonction de typage sera spécifiée de la façon suivante :

$$(\Gamma \vdash M : T) + (\exists^{\star} e \in E.\, \neg(\Gamma \vdash M : T))$$

qui s'extrait vers un programme qui repond `left` si le jugment est dérivable, ou bien retourne une erreur e sous la forme `right`(e) lorsque le jugement n'est pas dérivable. L'inconvénient de cette spécification est qu'elle n'établit aucun lien entre le message d'erreur émis et la preuve que M n'a pas le type T. Quand le typage échoue la fonction peut à priori retourner n'importe quel message d'erreur. Nous résoudrons ce problème en formalisant le sens des messages d'erreurs de façon à garantir que celui-ci soit pertinent.

Ainsi, on peut simplement considérer que l'on remplace une information non calculatoire (la preuve que $\Gamma \vdash M : T$ est absurde), par une information calculatoire (le message d'erreur e). On s'arrangera de manière à ce que le message d'erreur permette de retrouver à quel endroit la construction du jugement à échoué.

Le fait de modifier la spécification des fonctions de typage signifie que l'on rajoute les messages d'erreur dans le noyau. Cela n'est pas obligatoire, et l'on peut imaginer que le noyau échoue de manière non informative. Dans ce cas, une fonction logée dans l'interface pourrait essayer de faire un diagnostic pertinent sur la raison de l'échec, ce qui permettrait à l'utilisateur humain de corriger ses erreurs.

Ce style est évidemment maladroit car en général, l'information dont on a besoin est simplement calquée sur l'algorithme de typage, et l'état de la pile au moment où l'erreur est détectée est souvent suffisant. Mais il se peut parfois que l'algorithme de typage soit

[7] Les sources complets de la formalisation en Coq sont accessibles sur le Web à l'adresse `http://pauillac.inria.fr/~barras/typechecker/`.

tellement peu intuitif que l'utilisateur a besoin d'une information plus complexe à déterminer. Les compilateurs ML donnent parfois des messages d'erreurs assez compliqués qui ne permettent pas toujours de localiser l'erreur. Dans ces cas, l'écriture d'une fonction de typage indépendante peut se justifier.

Lorsque nous sommes dans les cas simples où les messages d'erreurs suivent l'algorithme, la modification du noyau n'est pas trop importante, et de toute façon, cela n'affecte que la partie "négative" de la spécification. Tout ce qui concerne la correction de l'algorithme est inchangé.

Langage de commandes

Puis on formalise ce qui vient autour du noyau : l'interface. Cela consiste d'abord à voir le noyau non pas comme une simple fonction de typage qui prendrait d'un bloc un développement, mais plutôt comme une machine dont un certain nombre de commandes permet de modifier l'état. Comme nous l'avons esquissé dans la description du noyau, l'état de cette machine est un contexte, avec comme invariant que ce contexte est toujours bien formé.

Enfin, on formalisera une partie du vérificateur de preuve que l'on ne range généralement pas dans le noyau mais dont la sûreté du système dépend pourtant : la traduction entre la syntaxe concrète et la syntaxe abstraite. Cela comprend la formalisation des AST (arbre de syntaxe abstraite), et notamment les traductions des termes entre leurs versions nommées et en indices de de Bruijn [19].

2.4.1 Termes

On se donne un ensemble de sortes SORT et un ensemble de noms NAME. Nous définissons la syntaxe concrète des termes de la manière suivante :

Définition 2.3 (type `expr`) *Les expressions sont soit une sorte, une variable nommée, un produit, une abstraction ou une application.*

$$T_1, T_2 : \textsc{Expr} \quad := \quad s \mid x \mid \Pi x{:}T_1.\, T_2 \mid \lambda x{:}T_1.\, T_2 \mid (T_1\ T_2)$$

avec $s \in \textsc{Sort}$ *et* $x \in \textsc{Name}$.

Définition 2.4 (prédicat `expr_vars`) *L'ensemble des variables libres d'une expression, noté* $\mathsf{FV}(M)$ *est défini de manière classique :*

$$\begin{aligned}
\mathsf{FV}(s) &= \varnothing \\
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(\Pi x{:}A.\, B) &= \mathsf{FV}(A) \cup (\mathsf{FV}(B) \backslash \{x\}) \\
\mathsf{FV}(\lambda x{:}A.\, M) &= \mathsf{FV}(A) \cup (\mathsf{FV}(M) \backslash \{x\}) \\
\mathsf{FV}((M\ N)) &= \mathsf{FV}(M) \cup \mathsf{FV}(N)
\end{aligned}$$

Définition 2.5 (prédicat `alpha`) *Le prédicat d'*α*-conversion utilise deux listes de noms*

$$\frac{s \in \textsc{Sort}}{\Sigma \vdash s \overset{\text{dB}}{\longleftrightarrow} s} \qquad \frac{\Sigma(n) = x \qquad \forall k{<}n.\, \Sigma(k) \neq x}{\Sigma \vdash x \overset{\text{dB}}{\longleftrightarrow} n}$$

$$\frac{\Sigma \vdash E_1 \overset{\text{dB}}{\longleftrightarrow} A \qquad \Sigma; x \vdash E_2 \overset{\text{dB}}{\longleftrightarrow} M}{\Sigma \vdash \lambda x{:}E_1.E_2 \overset{\text{dB}}{\longleftrightarrow} \lambda A.\, M} \qquad \frac{\Sigma \vdash E_1 \overset{\text{dB}}{\longleftrightarrow} u \qquad \Sigma \vdash E_2 \overset{\text{dB}}{\longleftrightarrow} v}{\Sigma \vdash (E_1\ E_2) \overset{\text{dB}}{\longleftrightarrow} (u\ v)}$$

$$\frac{\Sigma \vdash E_1 \overset{\text{dB}}{\longleftrightarrow} A \qquad \Sigma; x \vdash E_2 \overset{\text{dB}}{\longleftrightarrow} M}{\Sigma \vdash \Pi x{:}E_1.E_2 \overset{\text{dB}}{\longleftrightarrow} \Pi A.\, M}$$

FIG. 2.1: Traduction entre notations de de Bruijn et variables nommées

(ou de manière équivalente une liste de couples de noms) pour représenter les renommages.

$$\frac{}{([\,] \mid x) \overset{\alpha}{\equiv} ([\,] \mid x)} \qquad \frac{}{(\Sigma_1; x \mid x) \overset{\alpha}{\equiv} (\Sigma_2; y \mid y)} \qquad \frac{(\Sigma_1 \mid x) \overset{\alpha}{\equiv} (\Sigma_2 \mid y) \quad x \neq x' \quad y \neq y'}{(\Sigma_1; x' \mid x) \overset{\alpha}{\equiv} (\Sigma_2; y' \mid y)}$$

$$\frac{s \in \textsc{Sort}}{(\Sigma_1 \mid s) \overset{\alpha}{\equiv} (\Sigma_2 \mid s)} \qquad \frac{(\Sigma_1 \mid M) \overset{\alpha}{\equiv} (\Sigma_2 \mid M') \quad (\Sigma_1 \mid N) \overset{\alpha}{\equiv} (\Sigma_2 \mid N')}{(\Sigma_1 \mid (M\ N)) \overset{\alpha}{\equiv} (\Sigma_2 \mid (M'\ N'))}$$

$$\frac{(\Sigma_1 \mid T) \overset{\alpha}{\equiv} (\Sigma_2 \mid T') \quad (\Sigma_1; x \mid M) \overset{\alpha}{\equiv} (\Sigma_2; y \mid M')}{(\Sigma_1 \mid \lambda x{:}T.\, M) \overset{\alpha}{\equiv} (\Sigma_2 \mid \lambda y{:}T'.\, M')}$$

$$\frac{(\Sigma_1 \mid A) \overset{\alpha}{\equiv} (\Sigma_2 \mid A') \quad (\Sigma_1; x \mid B) \overset{\alpha}{\equiv} (\Sigma_2; y \mid B')}{(\Sigma_1 \mid \Pi x{:}A.\, B) \overset{\alpha}{\equiv} (\Sigma_2 \mid \Pi y{:}A'.\, B')}$$

Deux termes M et N seront dits α-convertibles si $([\,] \mid M) \overset{\alpha}{\equiv} ([\,] \mid N)$ est dérivable.

De manière interne, on n'utilisera que des termes en notation de de Bruijn [19], ce qui signifie que les variables sont représentées par des entiers, et que les lieurs ne portent pas de nom :

Définition 2.6 (type `term`) *La syntaxe abstraite (interne) des termes utilise les notations de de Bruijn :*

$$T_1, T_2 : \textsc{Term} \quad := \quad s \mid n \mid \Pi T_1.\, T_2 \mid \lambda T_1.\, T_2 \mid (T_1\ T_2)$$

avec $s \in \textsc{Sort}$ et $n \in \mathbb{N}$.

Définition 2.7 (prédicat `term_expr_equiv`) *La traduction entre la syntaxe concrète et la syntaxe abstraite est décrite par une relation ayant un paramètre supplémentaire qui donne un nom aux variables de de Bruijn libres (voir figure 2.1).*

Nous pouvons prouver que cette relation définit une bijection entre le type des expressions quotienté par l'α-conversion et le type des termes (en de Bruijn).

Lemme 2.8 (`equiv_unique`) *Deux termes équivalents à une même expressions sont égaux :*

$$\Sigma \vdash E \overset{\text{dB}}{\longleftrightarrow} t \ \wedge\ \Sigma \vdash E \overset{\text{dB}}{\longleftrightarrow} u \ \Rightarrow\ t = u$$

Lemme 2.9 (`unique_alpha`) *Deux expressions équivalentes au même terme sont α-convertibles :*

$$\Sigma_1 \vdash E_1 \overset{\mathrm{dB}}{\longleftrightarrow} t \ \wedge \ \Sigma_2 \vdash E_2 \overset{\mathrm{dB}}{\longleftrightarrow} t \ \Rightarrow \ (\Sigma_1 \mid E_1) \overset{\alpha}{\equiv} (\Sigma_2 \mid E_2)$$

Ces deux propriétés ne sont pas utilisées dans la suite. Elles servent simplement à s'assurer que la traduction vers la syntaxe concrète est possible :

Définition 2.10 (prédicat `name_unique`**)** *Pour éviter les ambiguïtés lors des traductions entre les deux styles de notations, nous devrons parfois utiliser des listes de noms distincts :*

$$\mathrm{AllDiff}(\Sigma) \ \overset{\mathrm{def}}{\equiv} \ \forall i, j.\, \Sigma(i) = \Sigma(j) \Rightarrow i = j$$

Algorithme 2.11 (`term_of_expr`) *Pour toute expression E, soit il existe un terme qui lui est équivalent dans Σ, soit il existe une liste non vide de variables apparaissant libre dans E, mais qui n'apparaissent pas dans Σ, ce qui empêche la traduction de E :*

$$\forall \Sigma, E.\, (\exists^\star t \in \textsc{Term}.\, \Sigma \vdash E \overset{\mathrm{dB}}{\longleftrightarrow} t) \ + \ (\exists^\star l \neq [].\, l \subset \mathsf{FV}(E) \backslash \Sigma)$$

La traduction en sens inverse n'est pas censée échouer car elle n'est appelée qu'avec des termes engendrés par le système, et donc bien formés, soit avec des traductions de termes donnés par l'utilisateur.

Algorithme 2.12 (`expr_of_term`) *Pour toute liste Σ de noms distincts, et pour tout terme ayant toutes ses variables libres liées par Σ, il existe une expression qui lui soit équivalente.*

$$\forall \Sigma, t.\, \mathrm{AllDiff}(\Sigma) \ \wedge \ |t| < |\Sigma| \ \Rightarrow \ \exists^\star E \in \textsc{Expr}.\, \Sigma \vdash E \overset{\mathrm{dB}}{\longleftrightarrow} t$$

La condition $\mathrm{AllDiff}(\Sigma)$ sert à éviter les conflits de noms : il n'est pas possible de traduire l'indice de de Bruijn 1 avec le contexte de noms $[x; x]$ car x désigne le de Bruijn 0. L'autre précondition assure que chaque indice de de Bruijn possède un nom.

2.4.2 Signification des messages d'erreurs

Définition 2.13 (type `type_error`**)** *L'ensemble des messages d'erreur de typage suit la syntaxe suivante :*

$$\begin{aligned} e : \textsc{TypErr} \ := \quad & \mathrm{Under}(T, e) \mid \mathrm{DbError}(n) \mid \mathrm{Topsort}(\mathrm{s}) \\ & \mid \mathrm{LambdaTopsort}(T, s) \mid \mathrm{ExpectType}(T_1, T_2, T_3) \\ & \mid \mathrm{NotAType}(T_1, T_2) \mid \mathrm{NotAFun}(T_1, T_2) \\ & \mid \mathrm{ApplyErr}(T_1, T_2, T_3, T_4) \end{aligned}$$

avec $T, T_1, T_2, T_3, T_4 \in \textsc{Term}$, $s \in \textsc{Sort}$ et $n \in \mathbb{N}$.

La figure 2.2 présente la signification informelle des messages d'erreurs de typage. Il s'agit du texte qui sera imprimé lorsque la machine retournera une erreur.

Définition 2.14 (prédicat `expln`**)** *La définition formelle du message d'erreur de la figure 2.3 permet de représenter le message d'erreur par la proposition logique correspondant à ce qui est affirmé informellement. On dira que Expln_Γ forme l'ensemble des messages d'erreurs cohérents dans le contexte Γ.*

Il est important de bien se convaincre que les figures 2.2 et 2.3 sont bien cohérentes.

Pour être plus précis, seul un sous-ensemble des erreurs cohérentes peut être retourné, en fonction de l'opération de typage :

$\mathrm{Under}(t_1, \ldots \mathrm{Under}(t_n, e))$	"Dans le contexte $t_1, \ldots, t_n$, e"
$\mathrm{DbError}(n)$	"L'indice de de Bruijn n n'est pas défini"
$\mathrm{Topsort(s)}$	"La sorte s n'a pas de type"
$\mathrm{LambdaTopsort}(t, s)$	"Le terme t est une fonction sur un type de sorte s"
$\mathrm{ExpectType}(m, t, t_e)$	"Le terme m a le type t, mais pas t_e"
$\mathrm{NotAType}(m, t)$	"Le terme $m : t$ n'est pas typable par une sorte"
$\mathrm{NotAFun}(m, t)$	"Le terme $m : t$ n'est pas fonctionnel"
$\mathrm{ApplyErr}(u, a, v, b)$	"Le terme $u : a$ ne peut être appliqué à $v : b$"

FIG. 2.2: Syntaxe concrète des messages d'erreur

$$\frac{\Gamma\,[t] \vdash \qquad e \in \mathrm{Expln}_{\Gamma\,[t]}}{\mathrm{Under}(t, e) \in \mathrm{Expln}_{\Gamma}} \qquad \frac{n \geq |\Gamma|}{\mathrm{DbError}(n) \in \mathrm{Expln}_{\Gamma}}$$

$$\frac{\forall t.\, \neg(\Gamma \vdash s : t)}{\mathrm{Topsort(s)} \in \mathrm{Expln}_{\Gamma}} \qquad \frac{\Gamma\,[t] \vdash m : s \qquad \forall t.\, \neg(\Gamma \vdash s : t)}{\mathrm{LambdaTopsort}(\lambda t.\, m, s) \in \mathrm{Expln}_{\Gamma}}$$

$$\frac{\Gamma \vdash m : t_{\mathrm{act}} \qquad \neg(\Gamma \vdash m : t_{\mathrm{exp}}) \qquad |t_{\mathrm{exp}}| < |\Gamma|}{\mathrm{ExpectType}(m, t_{\mathrm{act}}, t_{\mathrm{exp}}) \in \mathrm{Expln}_{\Gamma}}$$

$$\frac{\Gamma \vdash m : t \qquad \forall s \in \textsc{Sort}.\, \neg(\Gamma \vdash m : s)}{\mathrm{NotAType}(m, t) \in \mathrm{Expln}_{\Gamma}}$$

$$\frac{\Gamma \vdash m : t \qquad \forall a, b.\, \neg(\Gamma \vdash m : \Pi\, a.\, b)}{\mathrm{NotAFun}(m, t) \in \mathrm{Expln}_{\Gamma}}$$

$$\frac{\Gamma \vdash u : \Pi\, a.\, b \qquad \Gamma \vdash v : c \qquad \neg(\Gamma \vdash v : a)}{\mathrm{ApplyErr}(u, \Pi\, a.\, b, v, c) \in \mathrm{Expln}_{\Gamma}}$$

FIG. 2.3: Définition formelle des messages d'erreur

Définition 2.15 (prédicat `inf_error`) *L'ensemble* InfErr_t *des erreurs pouvant survenir pendant l'opération d'inférence est défini récursivement :*

$$\begin{aligned}
\mathrm{InfErr}_s &= \{\mathrm{Topsort(s)}\} \\
\mathrm{InfErr}_n &= \{\mathrm{DbError}(n)\} \\
\mathrm{InfErr}_{\lambda t.\, m} &= \mathrm{InfErr}_t \cup \mathrm{NotAType}(t, \textsc{Term}) \cup \mathrm{Under}(t, \mathrm{InfErr}_m) \\
&\quad \cup \{\mathrm{LambdaTopsort}(\lambda t.\, m, s)\} \\
\mathrm{InfErr}_{(u\ v)} &= \mathrm{InfErr}_u \cup \mathrm{NotAFun}(u, \textsc{Term}) \cup \mathrm{InfErr}_v \\
&\quad \cup \mathrm{ApplyErr}(u, \textsc{Term}, v, \textsc{Term}) \\
\mathrm{InfErr}_{\Pi\, t.\, u} &= \mathrm{InfErr}_t \cup \mathrm{NotAType}(t, \textsc{Term}) \cup \mathrm{Under}(t, \mathrm{InfErr}_u) \\
&\quad \cup \mathrm{Under}(t, \mathrm{NotAType}(u, \textsc{Term}))
\end{aligned}$$

On peut prouver que les erreurs cohérentes de InfErr_t permettent de prouver que t est un terme mal typé.

Lemme 2.16 (`inf_error_no_type`)

$$\mathrm{Expln}_\Gamma \cap \mathrm{InfErr}_m \neq \varnothing \ \Rightarrow\ \forall t.\, \neg(\Gamma \vdash m : t)$$

Note : il est nécessaire que Expln_Γ non vide implique que Γ soit bien formé, sinon on n'aurait pas $\Gamma \vdash m : t$, sans qu'il existe d'erreur pertinente pour l'opération d'inférence de type.

Définition 2.17 (prédicat `chk_error`) *La vérification que m a le type t échoue lorsque m est mal typé, lorsque t est mal typé, sauf si t est une sorte*[8]*, ou lorsque le type de m n'est pas convertible avec t :*

$$\begin{aligned}
\mathrm{ChkErr}_{(m:t)} &= \mathrm{InfErr}_m \cup \{e\mathrm{InfErr}_t \mid t \notin \textsc{Sort}\} \\
&\quad \cup \mathrm{ExpectType}(m, \textsc{Term}, t)
\end{aligned}$$

Ici encore, si l'ensemble des erreurs $\mathrm{ChkErr}_{(m:t)}$ a une intersection non vide avec l'ensemble des erreurs, alors m n'a pas le type t :

Lemme 2.18 (`chk_error_no_type`)

$$\mathrm{Expln}_\Gamma \cap \mathrm{ChkErr}_{(m:t)} \neq \varnothing \ \Rightarrow\ \neg(\Gamma \vdash m : t)$$

Définition 2.19 (prédicat `decl_error`) *L'ajout d'une variable de type t dans l'environnement échoue si t est mal typé ou si son type n'est pas convertible avec une sorte :*

$$\mathrm{DeclErr}_t = \mathrm{InfErr}_t \cup \mathrm{NotAType}(t, \textsc{Term})$$

Les erreurs correctes de $\mathrm{DeclErr}_t$ impliquent que $\Gamma\,[t]$ est un environnement mal formé :

Lemme 2.20 (`decl_err_not_wf`)

$$\mathrm{Expln}_\Gamma \cap \mathrm{DeclErr}_t \neq \varnothing \ \Rightarrow\ \neg(\Gamma\,[t] \vdash)$$

On peut alors reprendre les algorithmes de typage de façon à ce qu'ils retournent un message d'erreur cohérent et pertinent par rapport à l'opération exécutée :

$$\begin{aligned}
\Gamma \vdash \Rightarrow\ & \exists^\star t.\, (\Gamma \vdash m : t) + \exists^\star e.\, e \in \mathrm{Expln}_\Gamma \cap \mathrm{InfErr}_m \\
\Gamma \vdash \Rightarrow\ & (\Gamma \vdash m : t) + \exists^\star e.\, e \in \mathrm{Expln}_\Gamma \cap \mathrm{ChkErr}_{(m:t)} \\
\Gamma \vdash \Rightarrow\ & (\Gamma\,[t] \vdash) + \exists^\star e.\, e \in \mathrm{Expln}_\Gamma \cap \mathrm{DeclErr}_t
\end{aligned}$$

[8] Les sortes sont les "types des types", ce qui fait que ce sont les seuls types qui n'ont pas nécessairement de type.

Ces spécifications viennent se substituer à celles de la section précédente. Les résultats 2.16, 2.18 et 2.20 prouvent que ces nouvelles spécifications sont un raffinement des anciennes.

Elles se prouvent de la même manière que les spécifications de la section précédente, sauf qu'en cas d'echec (partie droite de la disjonction dans la specification), il faut donner un peu plus d'information (il faut prouver que le message d'erreur est correct et cohérent).

2.4.3 Fonctionnement de la machine

Le fonctionnement du système de preuves doit rappeler la façon de procéder en mathématiques. En général, un développement mathématique est une suite de déclarations qui peuvent être soit des hypothèses supplémentaires que l'on fait (ou l'introduction de nouvelles variables), soit l'énoncé d'un théorème que l'on souhaite prouver. Il existe bien d'autres opérations dans le langage mathématique usuel (comme par exemple la possibilité de poser des hypothèses locales, que l'on décharge ensuite dans tous les résultats qui l'utilisent), mais afin de rester simple, on se contentera de ces deux opérations.

Ainsi, un développement mathématique peut se formaliser comme un contexte (i.e. une liste de variables et de définitions), et savoir si un développement est correct revient à vérifier si le contexte est bien formé. S'il n'y avait pas d'interaction avec l'utilisateur, il suffirait d'écrire une fonction de typage des contextes.

Mais il faut tenir compte de la manière dont les preuves sont construites. L'utilisateur ne construit pas tout un développement d'un seul bloc : il commence par faire quelques définitions, mais il a besoin de savoir tout de suite si ce qu'il a déjà conçu ne contient pas d'erreurs, car si c'était le cas, tout ce qu'il pourrait faire ensuite pourrait s'avérer inutile.

L'utilisateur a donc besoin de savoir pas à pas si ce qu'il fait est correct, et pour des raisons d'efficacité évidentes, il ne veut pas re-vérifier tout ce qu'il a fait précédemment à chacun de ces pas. On décompose donc la vérification d'un contexte entier en une succession de commandes qui construisent progressivement le contexte, avec comme invariant que ce contexte est toujours bien formé. Toute commande qui tenterait de briser cet invariant serait rejetée.

On conçoit donc une machine ayant comme état un contexte valide, et muni de deux opérations : l'une ajoutant une variable, et l'autre ajoutant une définition. Pour le confort, il est utile de pouvoir aussi supprimer des définitions ou des variables acceptées, dans le cas où l'utilisateur se rend compte qu'il n'a pas formalisé ce qu'il croyait. Sans une telle commande, il devrait recommencer son développement depuis le début. On ajoute une troisième commande.

Définition 2.21 (type `state`) *L'état de la machine est un couple formé d'un environnement Γ et d'une liste de noms Σ que l'on associe aux variables globales. L'invariant est que l'environnement est bien formé, et qu'il existe un nom unique pour chacune des variables globales (i.e. définie par Γ)*

$$\textsc{State} \stackrel{\text{def}}{\equiv} \{(\Gamma, \Sigma) \in \textsc{Ctx} \times \textsc{Name}^* \mid \Gamma \vdash \ \wedge\ |\Gamma| = |\Sigma| \ \wedge\ \mathrm{AllDiff}(\Sigma)\}$$

Définition 2.22 (type `command`) *Les commandes que comprend notre système sont : l'inférence et la vérification de type, l'ajout d'un axiome, le retrait d'un axiome, l'affichage des axiomes actuellement définis et la fin de session.*

$$\begin{aligned}\textsc{Command} \quad := \quad & \mathrm{INFER}(T) \mid \mathrm{CHECK}(T_1, T_2) \mid \mathrm{AXIOM}(x, T) \\ & \mid \mathrm{DELETE} \mid \mathrm{LIST} \mid \mathrm{QUIT}\end{aligned}$$

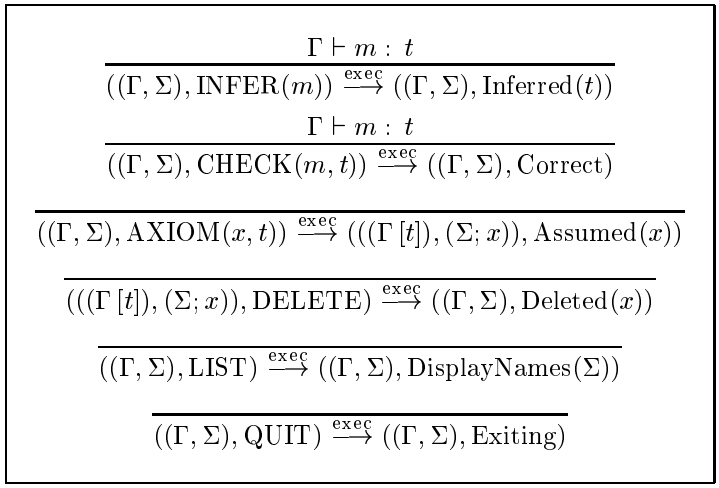


FIG. 2.4: Transitions

Définition 2.23 (type `message`) *Les messages émis en cas de succès de commandes ci-dessus sont :*

$$\begin{aligned} \text{MESSAGE} \quad := \quad & \text{Inferred}(T) \mid \text{Correct} \mid \text{Assumed}(x) \mid \text{Deleted}(x) \\ & \mid \text{DisplayNames}(\Sigma) \mid \text{Exiting} \end{aligned}$$

On lit des commandes sous forme d'AST dans un flot d'entrée. Étant donné un état et une commande, on définit les transitions : soit la commande réussit et l'on obtient un nouvel état accompagné d'un message indiquant ce qui s'est passé (voir figure 2.4) ; soit il s'est produit une erreur qui explique ce qui ne va pas.

Définition 2.24 (type `error`) *L'ensemble des erreurs pouvant survenir comprend les messages d'erreur, la tentative de redéfinir un axiome, et la tentative d'effacer un axiome alors que l'environnement est vide.*

$$\text{ERROR} \quad := \quad \text{NameClash}(x) \mid \text{TypeError}(e) \mid \text{CantDelete}$$

avec $x \in \text{NAME}$ *et* $e \in \text{TYPERR}$.

À chaque opération, on associe l'ensemble des erreurs qui peuvent surgir (figure 2.5). Ce qu'on prouve, c'est que si une opération peut retourner un message d'erreur, alors la commande n'aurait pas pu réussir. Cela nous garantit qu'il n'y aura pas de message émis à tort.

Le résultat principal est que pour toute commande, on peut soit atteindre un nouvel état tout en affichant un message indiquant que tout s'est bien passé, soit il existe un message d'erreur cohérent et qui soit pertinent avec la commande dans l'etat initial :

Lemme 2.25 (`interp_command`) *Pour tout état initial* S_i*, et toute commande* c*, soit il existe une transition vers un nouvel état, soit il existe une transition d'erreur :*

$$\forall S_\text{i}, c.\, \exists^\star (S_\text{f}, m).\, (S_\text{i}, c) \xrightarrow{\text{exec}} (S_\text{f}, m) \;+\; \exists^\star e.\, (S_\text{i}, c) \xrightarrow{\text{error}} e$$

Ceci garantit que le programme extrait de `interp_command` (l'interpréteur de commandes) est correct, c'est-à-dire qu'il ne fait que des transitions valides. Le lemme qui suit

$$\frac{e \in \mathrm{Expln}_\Gamma \cap \mathrm{InfErr}_m}{((\Gamma,\Sigma),\mathrm{INFER}(m)) \xrightarrow{\mathrm{error}} \mathrm{TypeError}(e)}$$

$$\frac{e \in \mathrm{Expln}_\Gamma \cap \mathrm{ChkErr}_{(m:t)}}{((\Gamma,\Sigma),\mathrm{CHECK}(m,t)) \xrightarrow{\mathrm{error}} \mathrm{TypeError}(e)}$$

$$\frac{e \in \mathrm{Expln}_\Gamma \cap \mathrm{DeclErr}_t}{((\Gamma,\Sigma),\mathrm{AXIOM}(x,t)) \xrightarrow{\mathrm{error}} \mathrm{TypeError}(e)}$$

$$\frac{x \in \Sigma}{((\Gamma,\Sigma),\mathrm{AXIOM}(x,t)) \xrightarrow{\mathrm{error}} \mathrm{NameClash}(x)}$$

$$\frac{}{(([],\Sigma),\mathrm{DELETE}) \xrightarrow{\mathrm{error}} \mathrm{CantDelete}}$$

FIG. 2.5: Transitions avec echec

est en quelque sorte un résultat de complétude. Il sert juste à vérifier que les spécifications sont saines.

Lemme 2.26 (`trans_error_no_confusion`) *Pour une commande donnée, il n'existe pas à la fois une transition correcte et une transition d'erreur partant du même état.*

$$\neg(\ (S_\mathrm{i}, c) \xrightarrow{\mathrm{exec}} (S_\mathrm{f}, m) \quad \wedge \quad (S_\mathrm{i}, c) \xrightarrow{\mathrm{error}} e\ )$$

Dans un état donné, un message d'erreur ne peut être retourné que lorsqu'il n'y a aucune transition possible vers un nouvel état. Ainsi, le système ne peut pas rejeter une commande qui admet une transition vers un nouvel état.

2.4.4 Construction du système

Enfin, on spécifie donc un interpréteur d'AST, qui commence par essayer de traduire l'AST en une commande, exécute la commande, et traduit en sens inverse le message résultant. L'extraction de l'interpréteur est un programme qui prend en argument l'état courant ainsi qu'un AST, et retourne soit le nouvel état avec un message, soit un message d'erreur. Cette fonction exécute une seule commande.

L'exécution d'un source (un flot d'AST) consiste à itérer l'interpréteur sur chacun des AST, chaque commande ayant comme état initial l'état final de la commande précédente. En sortie, on a un flot de messages d'acquiescement. Lorsque survient une erreur, on peut réagir de 2 façons : soit repartir avec le même état (comme un *toplevel*), soit tout arrêter (comportement habituel d'un compilateur). On ne pourrait pas faire cette distinction si les messages d'acquiescement et d'erreur étaient confondus.

Nous avons écrit la boucle de *toplevel* à la main. Cette boucle pouvant potentiellement ne jamais se terminer, il aurait fallu la spécifier à l'aide des types co-inductifs, mais cela aurait empêché l'extraction vers Objective Caml, qui est un langage à évaluation stricte.

En réalité, le flot d'entrée n'est pas un flot d'AST, mais un flot de caractères. De même, le flot de sortie doit être un flot de caractères. Nous avons donc dû écrire un analyseur syntaxique, ainsi qu'un afficheur pour les AST et les messages. Comme la représentation des AST et des messages a été choisie très proche de la syntaxe concrète, ces fonctions n'ont pas posé de difficulté, et nous supposerons qu'elles sont correctes.

En accolant ces programmes avec l'interpréteur d'AST, nous obtenons un système de preuve totalement indépendant, dont les sections précédentes forment le manuel utilisateur. À titre d'exemple, on peut commencer à axiomatiser l'ensemble des entiers naturels, et vérifier le type de quelques termes :

```
Coc < Axiom nat:Prop.
nat admis.

Coc < Axiom O:nat.
O admis.

Coc < Axiom S:nat->nat.
S admis.

Coc < Infer (S (S O)).
Type inféré: nat

Coc < Check [x:nat](S (S x)) : nat->nat.
Correct.
```

Les exemples suivants montrent comment le système réagit lorsque l'on entre une commande qui violerait l'invariant :

```
Coc < Axiom S:Prop.
Erreur: Nom S déjà utilisé.

Coc < Infer (S S).
Erreur: Le terme S : nat->nat ne peut être appliqué à S : nat->nat.
```

Ces messages sont, comme on s'y attendait, à la fois corrects et pertinents. Ils suggèrent avec précision la nature de l'erreur. Cependant, l'affichage de contextes locaux est problématique :

```
Coc < Infer [A:Prop][x:A](x x).
Erreur: Dans le contexte:
x0 : Prop
x1 : x0
Le type de x1, qui est x0 ne se réduit pas vers un produit.
```

Le message d'erreur est particulièrement difficile à lire car les noms entrés par l'utilisateur ont étés changés, et il devient difficile de faire le lien entre le terme entré et ce que le message affiche. Cela est dû au fait que le représentation interne des termes ne tient pas compte des noms, qui sont purement et simplement oubliés pendant la phase de synthèse. Seuls les objets introduits de manière globale (comme `O` et `S`) ont un nom.

Nous pouvons envisager deux manières de corriger ce problème en changeant la structure des termes. La première est d'opter pour une représentation des variables avec des noms, comme l'a fait Pollack dans [56]. Ce choix rend la métathéorie plus ardue, car au lieu de faire des récurrences structurelles, on utilise un schéma de récurrence modulo α-conversion.

Nous préférerons rester avec des indices de de Bruijn, en annotant les lieurs (abstractions et produits) avec un nom. Ce nom est considéré comme un préfixe pour les noms effectivement affichés (le suffixe étant calculé à l'affichage, de manière à distinguer les lieurs avec le même préfixe). Les opérations internes (typage, réduction, substitution, etc.) n'ont pas besoin d'en tenir compte.

Formellement, ce n'est pas aussi simple qu'il n'y paraît, car à certains endroits, nous devrons raisonner modulo une règle de pseudo-α-conversion, au lieu de l'égalité de Leibniz, très commode car elle est substitutive. Mais cela reste quand même plus simple que l'option variables nommées. Ce problème peut devenir assez sérieux pour montrer la confluence. Par exemple, si l'on considère l'η-réduction, le Calcul des Constructions en de Bruijn est confluent, mais les versions avec noms ne le sont pas (par rapport à l'égalité de Leibniz) : $\lambda x.((\lambda y.y)\ x)$ se β-réduit vers $\lambda x.x$, mais s'η-réduit vers $\lambda y.y$.

2.4.5 Conclusion de cette section

Tous les résultats de cette section sont faciles à prouver. Le point le plus important de ce travail est de fournir une structuration des preuves qui soit facilement compréhensible, sans être trop verbeuse.

Le langage de commandes certifié est encore de très bas niveau. L'extension naturelle de ce travail serait d'inclure des concepts mathématiques plus élaborés, comme par exemple le mécanisme de section.

Une autre direction à explorer serait de prouver l'analyseur syntaxique et l'afficheur. On ne s'y intéressera pas pour l'instant : on peut toujours aller plus loin dans la formalisation. Après avoir vérifié l'analyseur syntaxique, on pourra chercher à vérifier le système d'exploitation pour s'assurer que les caractères reçus sont bien ceux que l'utilisateur a effectivement tapé, et ainsi de suite. Nous préférons nous arrêter avant l'analyseur syntaxique. C'est un domaine qui a été abondamment étudié en informatique, et il n'est pas sûr que Coq soit le meilleur outil pour atteindre cet objectif. Des outils comme Yacc permettent d'écrire des analyseurs syntaxiques avec beaucoup plus de facilités que Coq.

Pour conclure, nous rappelons que l'on s'était fixé comme mission de suivre la structure d'un manuel de référence. En fait, nous pourrions considérer que le développement en Coq *est* la documentation. Bien sûr, comme notre objectif est le bootstrap de Coq, cela pose un problème de dépendance : pour lire le manuel du Coq bootstrappé, il faudrait être capable de comprendre un développement en Coq, ce qui est paradoxal. En réfléchissant, on se rend compte que cette situation est en fait assez courante : il existe bien des documents en français qui expliquent la grammaire française... La solution que nous apportons est de traduire le développement Coq vers des langages plus accessibles. Cela peut être le langage mathématique usuel comme cela est fait dans cette thèse, ou bien, en extrapolant sur le travail fait par Coscoy [15] sur l'explication de termes-preuves, on pourrait imaginer "extraire" du développement un manuel de référence rédigé en langue naturelle.

Chapitre 3

Réduction

Ce chapitre décrit un calcul de substitutions explicites, ou plus exactement, un calcul de fermetures, ainsi qu'une machine de réduction issue de ce calcul. Nous commencerons par motiver l'étude de l'implantation de fonctions de réduction en décrivant comment ces fonctions s'inscrivent dans le noyau d'un vérificateur de preuve.

La présence de la règle de conversion fait que l'algorithme de typage doit parfois tester si deux types sont convertibles, notamment dans le cas de l'application : lorsque l'on applique un terme de type $T \to U$ à un terme de type A, il faut vérifier que A et T sont convertibles.

Exemple 3.1 *Pour typer le terme*

$$\lambda x{:}P(2+2).\,(\lambda y{:}P(4).\,M\ x)$$

il faut tester la convertibilité entre $P(2+2)$ (le type de x), et $P(4)$ (le type du domaine de la fonction à laquelle x est appliqué).

En soi, cet exemple nous donne déjà une raison pour chercher à faire le test de conversion efficacement. Il faut reconnaître que dans le cas général, on ne fait pas une utilisation intensive de la règle de conversion. Par expérience, le test de conversion, même implanté naïvement, n'a jamais été une source d'inefficacité dans le typage de Coq.

3.1 Réflexion calculatoire

La situation change lorsque l'on met en œuvre la technique de *réflexion calculatoire* pour faire des preuves. Cette méthode a été adaptée à Coq par Boutin dans sa thèse [9]. Nous allons en expliquer de manière un peu informelle les principes. Nous avons évoqué le fait que notre système logique comportait un langage de programmation. Le principe de la réflexion est d'utiliser ce langage pour *calculer* au lieu de *raisonner*.

Cette idée de calculer plutôt que de prouver est très répandue en mathématiques. Elle est même probablement prédominante chez les ingénieurs. Les illustrations ne manquent pas. Pour factoriser un polynômes du deuxième degré $P(X) = aX^2 + bX + c$, on nous donne une "recette" : on calcule le discriminant $\Delta = b^2 - 4ac$, et suivant son signe, une formule permet de calculer le polynôme factorisé. Lorsque l'on apprend cette recette, on fait le calcul pour vérifier qu'elle est correcte, c'est-à-dire que le polynôme obtenu par calcul est égal au polynôme de départ. Mais on ne refait pas la factorisation à la main dans les cas particuliers : on se contente d'appliquer la recette apprise.

Il y a donc une fonction f des polynômes vers les polynômes qui fait la factorisation (ces manipulations sont syntaxiques), et une preuve de correction indiquant que le polynôme calculé est égal (extensionnellement) au polynôme initial :

$$\forall P.\, f(P) = P.$$

Lorsque l'on veut factoriser un polynôme P, on applique ce théorème de correction ce qui permet de remplacer P par $f(P)$. La deuxième étape est d'appliquer la règle de conversion pour évaluer $f(P)$; cette dernière étape correspond à l'application de la recette.

Le point-clé pour comprendre le bénéfice de la réflexion est que seule la première étape laisse une trace dans le terme-preuve. Ainsi, on ne garde dans la preuve que le fait qu'on a factorisé le polynôme (l'application du lemme de correction), mais on ne garde pas le détail de la factorisation : les différentes étapes de la recette ne sont pas gardées.

L'approche tactique consisterait à écrire un programme[1] qui ferait automatiquement les étapes de factorisation. La réflexion est une approche concurrente aux tactiques.

Dans le cas de la factorisation de polynômes du second degré, le nombre d'étapes est toujours le même, donc le rapport de taille des termes-preuve par la réflexion ou par l'approche tactique est constant. Ce n'est pas vrai dans l'exemple ci-dessous.

Suivant cette technique, une tactique de simplification de polynômes (développement, regroupement des monômes de dégrés identiques, etc.) a été écrite par Boutin [9], puis simplifiée par Loiseleur : il s'agit de la tactique `Ring`, qui essaie de reconnaître un polynôme dans une des structures d'anneau déclarées par l'utilisateur, et met ce polynôme sous une certaine forme canonique. La comparaison de deux polynômes devient facile.

```
Coq < Show.
1 subgoal

  a : Z
  b : Z
  ============================
   '(a+b)*(a-b) = a*a-b*b'

Coq < Ring '(a+b)*(a-b)' 'a*a-b*b'.
1 subgoal

  a : Z
  b : Z
  ============================
   '(-1)*(b*b)+a*a = (-1)*(b*b)+a*a'

Coq < Reflexivity.
Subtree proved!
```

Sur ce dernier exemple, le gain de la réflexion par rapport à l'approche tactique n'est plus un simple facteur constant. En effet, l'application du lemme de correction de notre recette est de taille proportionnelle à celle du polynôme, mais indépendant de la complexité du problème, le nombre d'étapes pour arriver à la forme simplifiée. L'approche par tactiques ferait autant de réécritures que d'étapes de simplifications, chaque étape laissant une trace dans le terme-preuve d'une taille proportionnelle au but.

La contrepartie est que l'on a repoussé toute la difficulté dans le test de conversion. L'implantation de Coq V5.10 ne permettait pas de trier une liste de plus de 150 entiers sur

[1] qui ne se trouve pas au même niveau que la fonction f : une tactique est un programme que l'on rajoute à l'implantation du système, alors que f est une fonction dans le système.

une machine de puissance raisonnable. Nous avons dû réimplanter les fonctions de réduction de Coq pour les versions ultérieures, ce qui permet désormais d'exploiter normalement la réflexion. À titre d'exemple, l'algorithme de tri fusion s'exécute effectivement avec une complexité de $N.\log N$, ce qui nous permet de trier des listes de plusieurs dizaines de milliers d'éléments en quelques minutes, soit un facteur approximatif de 400 avec la même implantation en Objective Caml.

Nous n'allons pas décrire en détail l'implantation actuelle, qui risque d'évoluer. Pour simplifier, nous allons nous intéresser au cas de la β-réduction dans le λ-calcul pur. Le système proposé a été partiellement vérifié en Coq.

3.2 Cas du λ-calcul pur

Nous allons seulement étudier le cas général du λ-calcul pur. Bien que la représentation des termes de Coq soit nettement plus complexe (les abstractions sont annotées avec des types, et il y a divers opérateurs, notamment à cause des types inductifs, des métavariables, etc.), nous pouvons nous ramener au cas du λ-calcul pur sans difficulté. Il suffit de considérer des nœuds d'application étiquetés et de modifier un peu les règles de réduction, mais cela n'est pas un changement fondamental. Le problème de complexité vient de la manière dont on fait les substitutions, non pas de la reconnaissance des radicaux.

3.2.1 Définition du problème

Définition 3.2 (type `term`) *Les termes du λ-calcul pur en notation de de Bruijn sont définis ainsi :*

$$t \quad := \quad n \mid \lambda t \mid (t_1\ t_2)$$

avec n un entier, et t_1, t_2 des termes.

Par convention, nous utiliserons uniquement des lettres en minuscule pour désigner des termes, car nous définirons d'autres classes de termes, et nous tenons à marquer une différence. Les règles de priorité pour noter les termes sont que l'abstraction est prioritaire sur l'application, et celle-ci est toujours représentée avec des parenthèses. Ainsi, $(\lambda 0\ 0)$ représente le terme $((\lambda 0)\ 0)$. En général, nous rendrons les termes plus lisibles en surchargeant les lieurs et les variables avec des noms. Le terme ci-dessus pourra être écrit $(\lambda_x.\, 0_x\ 0)$.

Le code Objective Caml des déclarations de types et des algorithmes se trouve en annexe B.

Définition 3.3 (prédicats `beta, ctxtt`) *La β-réduction des termes purs se définit à l'aide des règles suivantes :*

$$\frac{}{(\lambda m\ t) \triangleright_\beta m\{0 \backslash t\}} \qquad \frac{m \triangleright_\beta t}{\lambda m \triangleright_\beta \lambda t}$$

$$\frac{m \triangleright_\beta m'}{(m\ t) \triangleright_\beta (m'\ t)} \qquad \frac{m \triangleright_\beta m'}{(t\ m) \triangleright_\beta (t\ m')}$$

Le but ultime que nous nous fixons est de calculer la forme normale d'un terme pur. Tout d'abord, nous faisons quelques définitions qui permettront de formuler clairement notre but.

Définition 3.4 (prédicat `normal`) *L'ensemble $\mathcal{NF}^R$ des formes normales d'une relation R est l'ensemble des termes n'ayant aucun réduit :*

$$\mathcal{NF}^R \stackrel{\text{def}}{\equiv} \{m \mid \forall m'.\, m\ R\ m' \ \Rightarrow\ \bot\}$$

Il est bien connu que tous les λ-termes n'ont pas de forme normale. L'exemple le plus connu est $(\lambda_x(0_x\ 0_x)\ \lambda_x(0_x\ 0_x))$ qui se réduit vers lui-même. Mais nous nous placerons dans des systèmes pour lesquels tous les termes bien typés sont fortement normalisables.

La définition informelle de l'ensemble des termes fortement normalisables est habituellement qu'il n'existe pas de suite infinie de réduits. Cette définition n'est pas aussi pratique que celle employée par Altenkirch dans sa preuve formelle de normalisation forte du système F [2] :

Définition 3.5 (prédicat `sn`) *L'ensemble des termes fortement normalisables pour une règle de réduction R, est le plus petit ensemble X vérifiant la condition :*

$$\forall x.\,(\forall y.\,x\ R\ y\ \Rightarrow\ y \in X)\ \Rightarrow\ x \in X$$

Cet ensemble sera noté $\mathcal{SN}^R$.

En fait, les termes fortement normalisables sont les termes accessibles (définition A.1 de l'annexe) pour le symétrique de la réduction : $\mathcal{SN}^R \stackrel{\text{def}}{\equiv} \mathrm{Acc}_{R^{-1}}$

Cette définition fait apparaître clairement que les termes normaux sont fortement normalisables.

Nous pouvons ramener notre objectif à trouver une implantation efficace de la spécification suivante :

$$\forall m \in \mathcal{SN}^\beta.\,\exists^\star m'.\,m \triangleright_\beta m' \wedge m' \in \mathcal{NF}^\beta$$

Soit : pour tout terme fortement β-normalisable m, il existe un réduit m' (et un algorithme qui permette de le calculer) qui soit en forme β-normale.

3.2.2 Stratégies

Il reste un choix essentiel à faire : celui de la stratégie, c'est-à-dire de l'ordre dans lequel seront réduits les radicaux. Les performances (i.e. le nombre de pas à faire pour atteindre la forme normale) peuvent dépendre fortement de la stratégie employée. La terminaison peut même en dépendre. Cependant, nous ne nous intéressons qu'à réduire des termes fortement normalisables, donc nous pourrons librement choisir la stratégie qui nous convient le mieux.

Nous ne considérerons que des stratégies simples, en excluant par exemples les stratégies de réduction optimales [37], qui ne se prêtent pas à des implantations aussi efficaces que prévu. Principalement, il y a deux dimensions dans le choix du radical à réduire en priorité. La dimension "horizontale" indique si l'on réduit en priorité les radicaux se trouvant dans le sous-terme gauche ou droit d'une application. L'autre dimension permet soit de réduire les radicaux situés au sommet du terme, soit ceux qui sont proches des feuilles.

La stratégie d'appel par nom (abrégée CBN pour *call by name*) consiste à réduire en premier le radical le plus proche de la racine, en commençant par chercher le radical dans le sous-terme gauche de l'application. Une autre stratégie, l'appel par valeur (CBV pour *call by value*) réduit, comme l'appel par nom, le radical le plus proche de la racine, mais en cherchant d'abord dans le sous-terme droit de l'application.

Une variation de ces stratégies est d'avoir une réduction forte ou faible : la réduction faible consiste à ne pas considérer les radicaux situés sous une abstraction. Ces derniers ne seront réduits que lorsqu'il n'y aura plus que de radicaux sous des abstractions.

Ces stratégies ont des domaines d'applications différents : le principal avantage de la stratégie CBN est qu'elle ne réduit pas de radicaux inutilement. En revanche, la stratégie CBV réduit l'argument des appels de fonctions avant la fonction elle-même. S'il apparaît que la fonction n'utilise pas son argument, alors la stratégie CBV aura fait des réductions inutiles. D'un autre côté réduire un appel de fonction avant de réduire l'argument peut être

très inefficace, car l'on risque de dupliquer les radicaux qui se trouvent dans l'argument de la fonction que l'on réduit.

Les langages fonctionnels stricts comme Objective Caml ont opté pour une stratégie CBV faible. Ce choix se justifie par le fait que l'on écrit peu de programmes qui n'utlisent pas tous leurs arguments, et le fait d'avoir une stratégie faible permet d'une part de compiler les fonctions afin d'améliorer considérablement l'exécution, et d'autre part de simuler un appel par nom lorsqu'on le souhaite en cachant les valeurs dans des fonctions ayant un argument muet.

Lorsque l'on cherche à réduire des termes représentant des termes-preuves, il n'est pas du tout évident que cette stratégie soit toujours aussi efficace : on écrit plus souvent des fonctions qui n'utilisent pas leurs arguments (en termes logiques, on n'utilise pas toujours toutes les hypothèses dont on dispose), et l'on fait des codages fonctionnels sans toujours s'en rendre compte. Par exemple, en Objective Caml, il est fortement déconseillé d'écrire la fonction suivante :

```
let cond c t e = if c then t else e
```

qui donne un autre nom à la conditionnelle. En effet, à cause de l'appel par valeur les deux branches de l'alternative sont calculées, alors qu'une seule servira. Lorsque l'on engendre automatiquement des termes-preuves, ce genre de situation peut de produire. Une autre remarque est que l'on n'utilise pas souvent deux fois la même hypothèse. Tout ceci fait qu'en général, on préfère employer une stratégie CBN pour les termes-preuves.

À cause de l'isomorphisme de Curry-Howard, on peut avoir ces deux types de termes à réduire. Pire, il se peut qu'au sein d'un même terme, il y ait des parties logiques, et d'autres plus calculatoires. Nous allons chercher à implanter une stratégie qui cumule les avantages de ces deux stratégies. Il s'agit de la stratégie *paresseuse*. Il s'agit à la base d'un stratégie CBN, mais où l'on ne calcule qu'une seule fois l'argument d'une fonction, que l'on met à jour par effet de bord. Nous verrons plus loin les détails pratiques de cette implantation.

3.3 Premières implantations

Dans cette section, on propose un certain nombre d'implantations de fonctions de réduction d'efficacité variable. L'annexe B regroupe ces différentes implantations.

3.3.1 Implantations avec substitution

La première idée que l'on peut avoir pour implanter les fonctions de réduction est de suivre tout simplement la définition formelle. On commence par implanter une fonction de substitution, et l'on écrit très facilement une fonction qui cherche tous les radicaux dans un terme. La fonction de normalisation la plus simple que l'on puisse imaginer serait :

```
let rec nf = function
  | Rel i    -> Rel i
  | App(a,b) -> (match nf a with
                   (Lam t) -> nf (subst b t)
                 | na      -> App(na, nf b))
  | Lam t    -> Lam (nf t)
```

Pour donner une idée de l'efficacité (ou plutôt de l'inefficacité) de cette implantation, nous donnerons la complexité en temps du calcul du prédécesseur de n en utilisant la représentation de Church. Il est bien connu que le nombre de pas de réduction est au mieux en

$O(n)$. Nous souhaiterions avoir un interpréteur le moins pénalisant possible, qui permette en tout cas d'avoir cette complexité.

La fonction proposée ci-dessus est particulièrement inefficace : sa complexité est de $O(2^n)$. Cela est dû au fait que l'on normalise complètement a. Lorsque cela fait apparaître un radical, on substitue dans ce terme normal. Il faut donc recommencer à normaliser, et l'on fait deux appels récursifs sur des termes de taille au moins égale à a, d'où le facteur exponentiel. Une amélioration considérable est de ne pas renormaliser perpétuellement le corps de la fonction, en calculant simplement une forme normale de tête :

```
let rec hnf = function
  | App(a,b) -> (match (hnf a) with
                 | Lam f -> hnf (subst b f)
                 | a'    -> App(a',b))
  | t        -> t
```

Cette fonction est appliquée à notre terme à normaliser, puis récursivement à tous les sous-termes de la forme normale de tête, grâce à la fonction `strong` :

```
let rec strong f t =
  match (f t) with
  | Rel i    -> Rel i
  | Lam u    -> Lam (strong f u)
  | App(a,b) -> App(strong f a, strong f b)

let norm = strong hnf
```

La complexité est nettement meilleure ($O(n^2)$), mais il reste une marge de progrès. C'est une fonction bâtie sur ce modèle qui était implantée en `Coq` et qui ne permettait pas de trier de longues listes.

Le facteur quadratique vient du fait que l'on substitue plusieurs fois dans le même terme. Non seulement, on réalloue à chaque fois le terme dans lequel on substitue, mais en plus on substitue dans des termes de plus en plus gros, et parfois on pourrait deviner que la variable n'apparaît pas.

Prenons par exemple une fonction totalement appliquée à deux arguments dont le corps est un terme M ayant un certain nombre d'occurrences des variables x et y. Observons la réduction de ces deux radicaux :

$$\begin{array}{c}(\lambda_x\lambda_y(\overbrace{\ldots 1_x \ldots \lambda\lambda \ldots 2_y \ldots 3_x \ldots}^{M})\ t_1\ t_2)\\ (\lambda_y(\ldots \uparrow^1 t_1 \ldots \lambda\lambda \ldots 2_y \ldots \uparrow^3 t_1 \ldots)\ t_2)\\ (\ldots t_1 \ldots \lambda\lambda \ldots \uparrow^2 t_2 \ldots \uparrow^2 t_1 \ldots)\end{array}$$

Lors de cette opération, le terme M a été parcouru deux fois, t_1 quatre fois (deux fois à la première étape car il a fallu reloger les indices de de Bruijn de t_1, et deux fois ensuite lorsque l'on substitue t_2 dans un terme où t_1 apparaît deux fois), et t_2 une fois. Lors de la deuxième étape, on substitue deux fois dans t_1 alors que l'on pourrait deviner que ces sous-termes ne peuvent pas contenir d'occurrence de y puisqu'ils sont issus de la substitution de x qui n'est pas dans le scope de y.

3.3.2 La machine abstraite de Krivine (KAM)

Nous venons de voir que le problème venait du fait que l'on avait n étapes, chacune prenant un temps en moyenne proportionnel à n. On ne peut évidemment pas gagner sur le

nombre d'étapes, puisqu'il est lié à la complexité de notre problème (calculer le prédecessur de n). En revanche, nous pouvons nous arranger pour ne pas substituer n fois dans le même terme. La solution est d'accumuler les substitutions élémentaires en une substitution parallèle (appelée *environnement*) qui grossit au fur et à mesure que l'on réduit des radicaux.

Reprenant notre exemple, nous voyons que nous évitons les parcours inutiles de M et de t_1 :

$$\begin{gathered}(\lambda_x\lambda_y(\overbrace{\ldots 1_x \ldots \lambda\lambda \ldots 2_y \ldots 3_x \ldots}^{M})\ t_1\ t_2)\\ ((\{t_1/0_x\}\,\lambda_y M)\ t_2)\\ \{t_1/1_x;\ t_2/0_y\}M\\ (\ldots t_1 \ldots \{t_1/1_x;\ t_2/0_y\}\lambda\lambda \ldots 2_y \ldots 3_x \ldots)\\ (\ldots t_1 \ldots \lambda\lambda \ldots \uparrow^2 t_2 \ldots \uparrow^2 t_1 \ldots)\end{gathered}$$

Les deux premières étapes montrent comment s'accumulent les substitutions lors d'une β-réduction. Ensuite, la substitution se propage dans M en remplaçant dans le même passage les variables x et y. Cela permet en outre d'éviter la relocation de t_1 à la première occurrence de x car il se retrouve substitué dans son contexte d'origine. Au total, les termes M, t_1 et t_2 ne sont parcourus qu'une seule fois.

La machine abstraite de Krivine, qui est décrite dans [1] manipule des fermetures, qui sont des couples formés d'un environnement et d'un terme. Une fermeture dénote le terme dans lequel on effectue la substitution parallèle représentée par l'environnement. Une fermeture peut être vue comme un calcul non encore effectué.

L'état de la machine de Krivine est formé d'un triplet (e, t, s) : le couple (e, t) forme une fermeture dénotant la tête du terme à réduire, et s est la liste des arguments en attente d'être consommés, sous forme de fermeture. Cette liste s'appelle la *pile*. Les règles de transitions de la machine de Krivine sont les suivantes :

$$\begin{aligned}(e,\ (u\ v),\ s) &\rightarrow (e,\ u,\ \langle e, v\rangle :: s)\\ (e,\ \lambda t,\ c :: s) &\rightarrow (c :: e,\ t,\ s)\\ (\langle e', v\rangle :: e,\ 0,\ s) &\rightarrow (e',\ v,\ s)\\ (c :: e,\ n{+}1,\ s) &\rightarrow (e,\ n,\ s)\end{aligned}$$

La première règle propage la substitution à travers les applications. Les radicaux sont réduits simplement en déplaçant la fermeture de la pile vers l'environnement. Les deux dernières règles permettent de rechercher dans l'environnement la valeur associée à une variable.

Cette machine s'arrête lorsqu'elle trouve en tête de terme une variable libre (i.e. plus grande que la taille du contexte dans lequel elle se trouve), ou une abstraction non appliquée. Cela signifie que l'on a atteint la forme normale de tête faible du terme.

3.3.3 Machine de Krivine avec réduction forte (KN)

Si nous voulons calculer la forme normale d'un terme, il faut pouvoir être capable de réduire sous les abstractions qui ne se retrouvent pas appliquées. Il suffirait pour cela d'introduire deux règles que l'on pourrait représenter ainsi (en assimilant l'état (e, t, s) avec le terme que produit l'exécution complète de la machine) :

$$\begin{aligned}(e,\ \lambda t,\ []) &\rightarrow \lambda(\langle [], 0\rangle ::\uparrow^1 e,\ t,\ [])\\ ([],\ n,\ [\langle e_1, t_1\rangle \ldots \langle e_n, t_n\rangle]) &\rightarrow (n\ (e_1,\ t_1,\ []) \ldots (e_n,\ t_n,\ []))\end{aligned}$$

L'opération consistant à reloger entièrement l'environnement ($\uparrow^1 e$) peut être très coûteuse. Dans sa thèse, Crégut invente la machine KN [16, 17], qui évite ce problème en

ajoutant à l'état un entier qui mesure le nombre d'abstractions non substituées que l'on a franchi. Cet entier est utilisé pour reloger les variables libres. Au lieu de décaler tout l'environnement de un, on se contente d'incrémenter k. Au lieu d'insérer la variable 0 en tête, on introduit le de Bruijn négatif $-k$ (après incrémentation) qui se relogera bien en 0. Cela donne lieu à l'implantation suivante :

```
type clos = Clos of env * lambda
 and env  = clos list

let rec kn k e t s =
  match (e,t,s) with
    (e, App (a,b), s)             -> kn k e a (Clos(e,b)::s)
  | (e, Lam f, (arg::s))          -> kn k (arg::e) f s
  | (((Clos(e',u))::_), Rel 0, s) -> kn k e' u s
  | ((_::e), Rel n, s)            -> kn k e (Rel(n-1)) s
  | (e, Lam f, [])                ->
      Lam (kn (k+1) ((Clos([], Rel(-k-1)))::e) f [])
  | ([], Rel n, s)                ->
      List.fold_left (fun h (Clos(e,t)) -> App(h, kn k e t []))
        (Rel(k+n)) s

let norm_kn t = kn 0 [] t []
```

L'annexe B.3 propose une version améliorée : d'une part elle est programmée de manière récursive terminale, ce qui permettra d'évaluer des termes de taille quelconque sans que la pile de l'évaluateur soit affectée. Ensuite, on ne crée jamais de fermeture dont le terme est une simple variable : si c'est le cas, on va directement chercher dans l'environnement la fermeture correspondante.

Cette dernière optimisation, qui n'a pas l'air cruciale, supprime en fait de nombreuses indirections, et permet d'obtenir une complexité $O(n)$ sur l'exemple du prédécesseur.

Le code donné en annexe est autonome (il ne fait appel à aucune fonction auxiliaire). Cela forme un interpréteur en appel par nom qui est à la fois remarquablement concis et étonnament efficace. Il paraît assez difficle de l'améliorer localement de manière sensible. Évidemment, pour avoir un degré d'efficacité nettement supérieur, on peut toujours essayer de compiler les termes. Solution que nous éviterons car elle nécessite beaucoup d'efforts pour être mise en œuvre et cela est assez compliqué à maintenir, surtout dans un système de preuve qui n'a pas vocation d'être l'environnement ultime d'évaluation des programmes, bien que la réflexion calculatoire tendrait à nous y amener.

Toutefois, cette machine ne nous satisfait pas car elle emploie une stratégie d'appel par nom, alors que nous souhaitons une stratégie paresseuse. Nous allons maintenant aborder de considérations qui vont nous permettre d'introduire du partage.

3.4 La stratégie paresseuse

La stratégie paresseuse ou *lazy* est une stratégie similaire à l'appel par nom (réduction dans le sous-terme gauche de l'application en premier) sauf qu'elle répare le problème de duplication des radicaux, en réduisant simultanément tous les radicaux issus d'un même sous-terme. Tout comme l'appel par nom, elle ne réduit pas de radicaux inutiles (ou alors en même temps qu'un radical utile). Tout comme l'appel par valeur, elle ne réduit jamais plusieurs fois le même radical, puisque toutes les copies d'un même radical sont réduites

simultanément. Le nombre d'étapes minore à la fois celui de l'appel par nom et de l'appel par valeur. Malgré le coût supplémentaire pour gérer le partage, cela semble un bon compromis pour une stratégie que l'on va utiliser à la fois pour les preuves et les programmes fonctionnels.

Le partage s'implante généralement à l'aide de références qui pointent vers des calculs gelés. Lorsque l'on veut faire du filtrage sur un terme qui est un calcul gelé, on effectue un pas de calcul et on remet à jour la référence, de manière à ce que les autres occurrences en profitent.

Cependant, l'utilisation d'indices de de Bruijn nous réserve un problème inattendu qui nous empêche de faire du partage. Voici comment l'on ferait du partage avec des termes avec variables nommées :

$$\begin{array}{rl} & \lambda u.(\lambda x.(x\ \lambda y.x)\ (\underline{\lambda z.z\ u})) \\ \rightarrow & \lambda u.(\underline{\lambda z.z\ u}\ \lambda y.(\underline{\lambda z.z\ u})) \\ \rightarrow & \lambda u.(\underline{u}\ \lambda y.\underline{u}) \end{array}$$

La réduction du premier radical duplique un terme qui contient un radical. On voudrait pouvoir partager physiquement les résidus soulignés, afin de réduire en une seule étape tous les radicaux des résidus issus d'un même sous-terme, comme illustré ci-dessus.

Si l'on regarde ce qui se passe lorsque l'on utilise les indices de de Bruijn pour représenter les termes, on s'aperçoit que le partage n'est pas possible si les résidus apparaissent à des niveaux différents :

$$\begin{array}{rl} & \lambda_u(\lambda_x(0_x\ \lambda_y 1_x)\ (\underline{\lambda_z 0_z\ 0_u})) \\ \rightarrow & \lambda_u(\underline{\lambda_z 0_z\ 0_u}\ \lambda_y(\underline{\lambda_z 0_z\ 1_u})) \end{array}$$

Les deux résidus ont étés relogés différemment, et ils ne sont plus égaux, ce qui empêche de les partager en mémoire.

Pour préserver le partage le plus longtemps possible, on retarde le calcul des relocations, en ajoutant un constructeur de relocation (un *shift*) dans les termes. Ainsi, les variables 0_x et 1_x sont respectivement substitutées par $\lambda_z 0_z\ 0_u$ et $\uparrow^1\lambda_z 0_z\ 0_u$:

$$\begin{array}{rl} & \lambda_u(\lambda_x(0_x\ \lambda_y 1_x)\ (\underline{\lambda_z 0_z\ 0_u})) \\ \rightarrow & \lambda_u(\underline{\lambda_z 0_z\ 0_u}\ \lambda_y \uparrow^1(\underline{\lambda_z 0_z\ 0_u})) \\ \rightarrow & \lambda_u(\underline{0_u}\ \lambda_y \uparrow^1\underline{0_u}) \end{array}$$

La relocation est *explicite* : il ne s'agit pas du résultat de la relocation, mais d'un nouveau constructeur de terme. Contrairement aux substitutions, il n'est pas nécessaire de propager les relocations. En effet, une substitution peut remplacer une variable par une abstraction, et faire aparaître de nouveaux radicaux. Il faut donc alternativement réduire les radicaux et propager les substitutions. La situation est différente avec les relocations puisqu'une variable est toujours relogée en une variable : le calcul des relocations ne crée pas de radical. Il suffit de prendre garde que ces relocations peuvent masquer des radicaux si elles se trouvent insérées entre une application et une abstraction.

Dans un premier temps, nous présenterons le système sous une forme entièrement applicative, ce qui permet de faire une preuve de correction plus simple. Nous espérons qu'ensuite, la preuve de correction s'adaptera à l'implantation avec partage. L'idée étant que les effets de bords sont juste une manière de réduire simultanément des radicaux identiques qui ont plusieurs occurrences dans le terme à réduire.

3.5 Système de substitutions explicites

Les systèmes de substitutions explicites sont des ensembles de règles de réduction dont le but est d'expliquer comment s'effectue l'opération de substitution. Ils sont souvent utilisés

(Beta)	$(\lambda M\ N)$	$\to$	$[\mathrm{id}.N]M$
(App)	$[S](M\ N)$	$\to$	$([S]M\ \ [S]N)$
(Lambda)	$[S]\lambda M$	$\to$	$\lambda\,[\Uparrow S]M$
(VarId)	$[\mathrm{id}]n$	$\to$	n
(FVarCons)	$[S.T]0$	$\to$	T
(RVarCons)	$[S.T](n+1)$	$\to$	$[S]n$
(FVarLift)	$[\Uparrow S]0$	$\to$	0
(RVarLift)	$[\Uparrow S](n+1)$	$\to$	$\big[\uparrow^1 S\big]\,n$
(VarPhiPhi)	$\big[\uparrow^k\uparrow^p S\big]\,n$	$\to$	$\big[\uparrow^{k+p} S\big]\,n$
(VarPhiId)	$\big[\uparrow^k \mathrm{id}\big]\,n$	$\to$	$n+k$
(FVarPhiCons)	$\big[\uparrow^k (S.T)\big]\,0$	$\to$	$\big[\uparrow^k \mathrm{id}\big]\,T$
(RVarPhiCons)	$\big[\uparrow^k (S.T)\big]\,(n+1)$	$\to$	$\big[\uparrow^k S\big]\,n$
(FVarPhiLift)	$\big[\uparrow^k \Uparrow S\big]\,0$	$\to$	k
(RVarPhiLift)	$\big[\uparrow^k \Uparrow S\big]\,(n+1)$	$\to$	$\big[\uparrow^{k+1} S\big]\,n$

FIG. 3.1: Le système $\lambda\phi$

pour prouver la correction des machines abstraites de réduction. Par exemple, $\lambda\sigma$ sert à prouver la correction de la KAM. Nous allons définir et étudier un système de substitutions explicites, qui permettra de tenir compte des idées de la section précédente. Tout d'abord nous présenterons le système $\lambda\phi$, qui possède de nombreux points commun avec celui que nous proposerons.

3.5.1 Le système $\lambda\phi$

Cette section rappelle le système $\lambda\phi$, présenté par Lescanne dans [36]. Ce système est intéressant de par la manière dont il a été conçu : avec la volonté de faire un système permettant d'effectuer des substitutions dans des termes sans métavariables, en introduisant les opérateurs de substitutions par nécessité.

Définition 3.6 *Les termes de $\lambda\phi$ comportent les variables, abstraction, application et les fermetures ; les substitutions sont l'identité, le* cons*, et les relocations (*shift *de n variables et* lift*) :*

$$\begin{aligned} T_\phi &:= n \mid \lambda T \mid (T\ T') \mid [S]T \\ S_\phi &:= \mathrm{id} \mid S.T \mid \uparrow^n S \mid \Uparrow S \end{aligned}$$

avec $n \in \mathbb{N}$.

Dans les substitutions, les opérateurs *lift* ($\Uparrow$) et *shift* ($\uparrow$) sont prioritaires par rapport au *cons* : $\uparrow^k S.T$ est équivalent à $(\uparrow^k S).T$.

Définition 3.7 *Les règles de réduction de $\lambda\phi$ sont présentées figure 3.1. L'opérateur $\Phi(S,k)$, que nous notons $\uparrow^k S$ signifie que l'on effectue la substitution S, puis on reloge le terme obtenu de k variables à partir de 0.*

Le système privé de la règle Beta *s'appelle ϕ.*

On remarquera que les règles VarPhiId à RVarPhiLift sont des doubles des règles VarId à RVarLift. Elles servent à prendre en compte le cas où une relocation se retrouve en tête de substitution. Notre système n'aura pas cette duplication grâce à l'opérateur de relocation explicite.

Le système ϕ est fortement normalisant et confluent, ce qui peut se prouver semi-automatiquement par ORME. Lescanne prouve aussi la confluence de $\lambda\phi$ sur l'ensemble des termes clos.

3.5.2 Le système de fermetures inspiré de $\lambda\phi$

Dans cette section, nous introduisons un calcul de substitutions explicites, système qui servira de base à l'implantation de nos fonctions de réduction. Puis, nous décrirons des simplifications qui donneront lieu à un système très concis $\lambda\phi^c$, que nous étudierons formellement.

Définition 3.8 *La syntaxe des termes avec relocations et substitutions explicites est définie par récurrence mutuelle, comme l'étaient fermetures et environnements :*

$$\begin{array}{rcl} T & := & \overline{n} \mid \lambda_{\langle S,t\rangle} T \mid (T \cdot T') \mid \uparrow^n T \mid [S]t \\ S & := & \mathrm{id} \mid S.T \mid \uparrow^n S \mid \Uparrow^n S \end{array}$$

avec $n \in \mathbb{N}$.

Les opérateurs de substitution sont les mêmes que ceux de $\lambda\phi$, sauf que les *lifts* sont n-aires afin d'avoir une représentation plus compacte.

Un point important à noter est que l'opérateur de fermeture $[S]t$ porte sur des termes *purs* comme définis section 3.2.1. Cela nous empêche d'avoir des termes avec plusieurs niveaux de substitution. C'est la raison pour laquelle nous qualifierons plutôt ce système de *système de fermetures*. Cette volonté d'interdire les termes avec plusieurs niveaux de substitutions est le résultat de la discussion sur la source d'inefficacité des fonctions de réduction implantées naïvement.

Afin de ne pas trop introduire de confusion entre les deux classes de termes, nous emploierons uniquement des minuscules pour représenter des termes purs, et uniquement des majuscules pour les termes avec fermeture. D'autre part les variables, l'application et l'abstraction auront des notations sensiblement différentes.

Notre calcul occupe une position intermédiaire entre les structures de données utilisées par la KAM et celles des substitution explicites :

- Par rapport à la KAM, les fermetures sont plus complexes. $[S]t$ représente bien la fermeture au sens des machines abstraites, mais il y a les trois autres constructeurs correspondant aux fermetures partiellement évaluées. Grâce au partage, les fermetures que l'on stocke dans la pile peuvent bénéficier des calculs fait en tête de terme ; il faut donc une notation pour ces formes partiellement calculées. Cela est inutile dans la KAM puisqu'il n'y a pas de partage. Du côté des susbtitutions, le *cons* correspond au *cons* des environnements. Nous avons en plus des opérateurs de relocation pour tenir compte des variables libres (réduction forte).
- Par rapport aux calculs de substitutions explicites : l'opérateur de substitution ne porte pas sur les termes mais sur les termes purs, comme nous l'avons déjà fait remarquer.

(App)	$[S](u\ v)$	$\to$	$([S]u \cdot [S]v)$	
(Lambda)	$[S]\lambda t$	$\to$	$\lambda_{\langle S,t\rangle}\ [\Uparrow^1 S]t$	
(VarId)	$[\mathrm{id}]n$	$\to$	$\overline{n}$	
(VarCons0)	$[S.T]0$	$\to$	T	
(VarConsS)	$[S.T](n{+}1)$	$\to$	$[S]n$	
(VarShift)	$[\uparrow^k S]n$	$\to$	$\uparrow^k[S]n$	
(VarLiftLt)	$[\Uparrow^k S]n$	$\to$	$\overline{n}$	si $n < k$
(VarLiftGe)	$[\Uparrow^k S]n$	$\to$	$\uparrow^k[S](n{-}k)$	si $n \geq k$

Fig. 3.2: Règles de réductions des substitutions

Définition 3.9 *Les règles de réduction se construisent de la même manière que pour ϕ, ce qui donne un système plus simple puisque nous ne nous occupons pas des relocations explicites (voir figure 3.2).*

Les deux premières règles servent à propager la substitution jusqu'aux feuilles des termes (i.e. jusqu'aux variables). On comprend au passage que le *lift* compte le nombre de λ franchis. Il reste alors à définir comment s'expansent les variables. L'identité n'affecte pas les variables, et le *cons* permet de faire des substitutions en parallèle.

La règle de l'abstraction crée un terme qui contient des informations redondantes. Cela est dû au fait que les abstractions ont essentiellement deux utilisations :

- soit comme fonction dont on veut évaluer le résultat par application. Pour respecter notre principe de ne pas superposer les niveaux de substitution, il faut attendre que l'argument effectif soit connu pour propager la substitution. C'est le but de la fermeture qui annote l'abstraction. Comme le terme de la fermeture n'est jamais modifié, nous pourrions en fait mettre une version compilée de ce terme pur.
- soit comme un terme quelconque dont on veut connaître la forme normale. Dans ce cas, il nous faut propager ls substitution dans le corps sans avoir d'argument. Plus précisement, on évalue la fonction avec une variable fraîche. Le sous-terme T de l'abstraction $\lambda_{\langle S,t\rangle}T$ permet de partager ce calcul entre toutes les occurrences qui utilisent cette abstraction en tant que valeur.

Cette représentation redondante de l'abstraction compliquera la preuve de correction de ce calcul avec le λ-calcul pur car il faudra maintenir une cohérence entre ces deux sous-termes.

Nous voyons aussi comment la duplication des règles de $\lambda\phi$ est évitée, en poussant tout simplement les *shifts* des substitutions dans les termes.

On peut remarquer pour l'instant, la représentation n-aire des opérateurs de relocation n'est pas utilisée puisque les règles de la figure 3.2 n'engendrent et ne progagent que des relocations d'amplitude un. On rend la représentation plus compacte en adoptant les règles de simplifications de la figure 3.3. Cependant, le système resterait cohérent si l'on ne considérait pas ces règles. On peut même n'en considérer qu'une partie, avec pour seule contrainte que si l'on prend la quatrième règle, alors il faut aussi prendre la cinquième pour refermer la paire critique de $[\Uparrow^k \mathrm{id}]n$ lorsque $n < k$.

On peut montrer semi-automatiquement que ce système est confluent et fortement normalisant (avec ou sans les règles optionnelles), grâce à ORME. Il suffit d'indiquer à ORME

$$
\begin{array}{rcl}
\uparrow^k\uparrow^{k'}S & \to & \uparrow^{k+k'}S \\
\Uparrow^k\Uparrow^{k'}S & \to & \Uparrow^{k+k'}S \\
\uparrow^k\uparrow^{k'}T & \to & \uparrow^{k+k'}T \\
\Uparrow^k\mathrm{id} & \to & \mathrm{id} \\
\uparrow^k\overline{n} & \to & \overline{n+k}
\end{array}
$$

FIG. 3.3: Règles de simplifications

le sens des égalités comme nous les avons indiquées.

Pour définir la β-réduction, il faut tenir compte du fait que des relocations explicites peuvent se glisser entre l'abstraction et l'application. Si nous prenons les règles de simplifications, il suffit de considérer les cas où il y a zéro ou une relocations intercalées. Le problème ne se pose pas avec l'opérateur de fermeture $[S]t$ car celui-ci se réduit toujours grâce aux règles de la figure 3.2

Définition 3.10 *La relation $\bar{\beta}$ caractérise la β-réduction au niveau des termes avec fermetures :*

$$
\begin{array}{rcl}
(\lambda_{\langle S,m\rangle}T\cdot N) & \to & [S.\,N]m \\
(\uparrow^k\lambda_{\langle S,m\rangle}T\cdot N) & \to & \left[\uparrow^k S.\,N\right]m
\end{array}
$$

Ce calcul de fermeture ne pose pas les problèmes rencontrés habituellement avec les calculs de substitutions explicites généraux, car il évite la paire critique qui consiste à propager une substitution à travers un β-radical. Le problème vient généralement du fait que la propagation d'une substitution S à travers un radical, permet d'avoir soit un terme dans lequel on fait d'abord S, puis la substitution élémentaire engendrée par β, soit d'abord β, puis S. Pour refermer cette paire critique, on introduit des règles de composition des substitutions dont la plupart du temps la règle MAP, ce qui produit un système possèdant des termes non fortement normalisables, comme le montre Melliès [40].

La confluence du système comprenant toutes les règles de la section précédente et $\bar{\beta}$ serait assez facile, puisque $\bar{\beta}$ n'engendre qu'une seule paire critique $(\uparrow^0\lambda_{\langle S,t\rangle}T\cdot N)$, qui se referme immédiatement. De toute façon, nous montrerons la correction de notre système sans avoir recours au résultat de confluence.

Le système que nous allons étudier est une version plus simple que celle présentée dans cette section. Les simplifications prises en compte sont les suivantes :

- Nous pouvons vérifier facilement que $\Uparrow^1 S$ se comporte comme $\uparrow^1 S.\,\overline{0}$. Nous oublions donc l'opérateur *lift*.
- L'opérateur *shift* apparaît toujours associé à *cons* (seule la règle $\bar{\beta}$ introduit le *shift* des substitutions). Nous pouvons alors fusionner ces deux opérateurs.
- L'indice de de Bruijn $\overline{n}$ sera représenté par $\uparrow^n\overline{0}$.

$$
\begin{array}{lrcl}
(\textsc{Beta}_0) & (\lambda_{\langle S,t\rangle} T \cdot N) & \to & [\uparrow^0 S.\, N]t \\
(\textsc{Beta}_1) & (\uparrow^n \lambda_{\langle S,t\rangle} T \cdot N) & \to & [\uparrow^n S.\, N]t \\
 & & & \\
(\textsc{App}) & [S](u\ v) & \to & ([S]u \cdot [S]v) \\
(\textsc{Lambda}) & [S]\lambda t & \to & \lambda_{\langle S,t\rangle}\, [\uparrow^1 S.\, \overline{0}]\, t \\
(\textsc{VarId}) & [\mathrm{id}]n & \to & \uparrow^n \overline{0} \\
(\textsc{VarCons0}) & [\uparrow^k S.\, T]\, 0 & \to & T \\
(\textsc{VarConsS}) & [\uparrow^k S.\, T]\, (n+1) & \to & \uparrow^k [S]n \\
(\textsc{Shift0}) & \uparrow^0 T & \to & T \\
(\textsc{ShiftComp}) & \uparrow^n \uparrow^k T & \to & \uparrow^{n+k} T
\end{array}
$$

FIG. 3.4: Règles de réductions du système $\lambda\phi^c$

3.5.3 Le système $\lambda\phi^c$

Définition 3.11 (types `fterm, subs`) *Compte tenu des simplifications de la section précédente, les termes avec relocations et les substitutions explicites deviennent :*

$$
\begin{array}{lcl}
T_{\lambda\phi^c} & := & \overline{0} \mid \lambda_{\langle S,t\rangle} T \mid (T \cdot T') \mid \uparrow^n T \mid [S]t \\
S_{\lambda\phi^c} & := & \mathrm{id} \mid \uparrow^n S.\, T
\end{array}
$$

avec $n \in \mathbb{N}$.

On utilisera quand même la notation $\Uparrow S$ pour représenter la substitution $\uparrow^1 S.\, \overline{0}$. On remarquera que les substitutions sont redevenues quasiment des listes de fermetures, au détail près que l'on ajoute comme information le nombre de variables libres créées à chaque étape.

Cette définition est similaire à la syntaxe restreinte dans [18], page 7, où l'on utilise les λ-termes comme syntaxe auxiliaire pour les fermetures.

Définition 3.12 (prédicats `fbeta, lamphi`) *Les règles de réduction du système $\lambda\phi^c$ (applicables uniquement au sommet d'un terme) sont présentées figure 3.4. Le sous-système formé des deux premières règles sera appelé $\bar{\beta}$. Le reste des règles forme le système ϕ^c.*

On aurait pû ne pas prendre la règle ShiftComp, avec le schéma de règle $\bar{\beta}$ suivant, mais cette définition derait plus complexe à manipuler :

$$(\textsc{Beta}_n) \quad (\uparrow^{k_1} \ldots \uparrow^{k_n} \lambda_{\langle S,t\rangle} T \cdot N) \quad \to \quad [\uparrow^{\sum k_i} S.\, N]\, t$$

Lemme 3.13 (`wf_lamphi`) *Le système ϕ^c est fortement normalisant.*

Preuve Automatique avec ORME. ■

La définition ci-dessous décrit les contextes dans lesquels on autorise les réductions. Nous nous restreignons à ne pas faire de réduction dans les substitutions, car il risque d'être assez compliqué de prouver la normalisation forte du calcul (que nous ne ferons que conjecturer).

Réduire sous les substitutions n'est pas nécessaire pour atteindre la forme normale. La réduction sous les abstractions est déjà assez problématique et nous avons dû "stratifier" les réductions en fonction du nombre d'abstractions sous lesquelles elles ont lieu.

Définition 3.14 (prédicat `ctxt_sub`) *Pour une relation R, la réduction sous i abstractions est définie par les clauses suivantes :*

$$\frac{M \to_R N}{M \to_{[R]_0} N} \qquad \frac{M \to_{[R]_i} N}{(M \cdot T) \to_{[R]_i} (N \cdot T)} \qquad \frac{M \to_{[R]_i} N}{(T \cdot M) \to_{[R]_i} (T \cdot N)}$$

$$\frac{M \to_{[R]_i} N}{\lambda_{\langle S,t\rangle} M \to_{[R]_{i+1}} \lambda_{\langle S,t\rangle} N} \qquad \frac{M \to_{[R]_i} N}{\uparrow^k M \to_{[R]_i} \uparrow^k N}$$

Cet opérateur de fermeture sera utilisé avec les relations $\bar{\beta}$, ϕ^c et $\bar{\beta}\phi^c$. La notation $\lambda\phi^c$ désigne la réduction $[\lambda\phi^c]$.

Pour simplifier, nous dirons que les termes appartenant à $\mathcal{SN}^{\lambda\phi^c}$ sont les termes fortement normalisables. En fait, ils désignent des termes faiblement normalisables pour le $\lambda\phi^c$ où l'on autorise les réductions dans tous les contextes, mais dont aucun chemin de réduction suivant notre stratégie (sans réductions dans les substitutions) n'est infini.

Lemme 3.15 (`ft_ind2`) *Soit P un prédicat sur les termes. P est vrai sur l'ensemble des termes s'il vérifie les conditions suivantes :*

- $P(\overline{0})$
- $\forall n \in \mathbb{N}.\, P([\mathrm{id}]n)$
- $\forall k, S, M.\, P(M) \;\Rightarrow\; P(\big[\uparrow^k S.\, M\big]0)$
- $\forall n, k, S, M.\, P(\uparrow^k [s]n) \;\Rightarrow\; P(\big[\uparrow^k S.\, M\big](n{+}1))$
- $\forall M, N.\, P(M) \;\wedge\; P(N) \;\Rightarrow\; P((M \cdot N))$
- $\forall S, u, v.\, P([S]u) \;\wedge\; P([S]v) \;\Rightarrow\; P([S](u\ v))$
- $\forall M, S, m.\, P(M) \;\wedge\; P([S]\lambda m) \;\Rightarrow\; P(\lambda_{\langle S,m\rangle} M)$
- $\forall S, t.\, P([\Uparrow S]t) \;\Rightarrow\; P([S]\lambda t)$
- $\forall n, M.\, P(M) \;\Rightarrow\; P(\uparrow^n M)$

Ce schéma de récurrence dit que l'ensemble des termes est décrit par application des constructeurs du λ-calcul et par ϕ^c-expansion. Il est commode pour prouver des propriétés qui se transmettent par propagation des substitutions.

3.6 Correction de $\lambda\phi^c$ par rapport au λ-calcul

Ce que nous souhaitons montrer, c'est que l'on peut déduire un algorithme de normalisation pour le λ-calcul pur, étant donné un algorithme de normalisation pour $\lambda\phi^c$. Soit t un terme pur à normaliser. On commence par former la fermeture $[\mathrm{id}]t$, que l'on normalise avec l'algorithme opérant sur les fermetures. La forme normale obtenue ne contient donc plus de substitution. Il suffit alors d'effectuer les relocations explicites pour revenir dans le monde des termes purs.

L'idée de base serait de définir une fonction d'interprétation qui relie chaque terme avec fermeture à un terme pur (intuitivement, en effectuant toutes les substitutions et relocations), et de montrer que cette interprétation préserve la β-réduction et les formes normales.

$$
\begin{array}{rcl}
\{s\}n & = & s(n) \\
\{s\}(a\ b) & = & (\{s\}a\ \ \{s\}b) \\
\{s\}\lambda m & = & \lambda\,\{\Uparrow s\}m \\
 & & \\
[\![\overline{0}]\!]_i & = & 0 \\
[\![(M\cdot N)]\!]_i & = & ([\![M]\!]_i\ [\![N]\!]_i) \\
[\![\lambda_{\langle S,t\rangle}T]\!]_0 & = & \{[\![S]\!]\}\lambda t \\
[\![\lambda_{\langle S,t\rangle}T]\!]_{i+1} & = & \lambda[\![T]\!]_i \\
[\![\lambda_{\langle S,t\rangle}T]\!]_\infty & = & \lambda[\![T]\!]_\infty \\
[\![\uparrow^k T]\!]_i & = & \uparrow^k [\![T]\!]_i \\
[\![[S]t]\!]_i & = & \{[\![S]\!]\}t \\
 & & \\
[\![\mathrm{id}]\!](n) & = & n \\
[\![\uparrow^k S.\,T]\!](0) & = & [\![T]\!]_0 \\
[\![\uparrow^k S.\,T]\!](n{+}1) & = & \uparrow^k [\![S]\!](n)
\end{array}
$$

FIG. 3.5: Interprétation

Mais la redondance de l'abstraction va nous causer quelques problèmes : les deux sous-termes d'une abstraction $\lambda_{\langle S,t\rangle}T$ représentent le même terme pur au moment où ils sont créés par la règle de propagation à travers l'abstraction. Mais ils divergent lorsque l'on fait des réductions dans le terme avec fermetures T . Cependant, comme on ne réduit pas dans S, on a comme invariant que T représente un réduit de $\big[\Uparrow^1 S\big]t$.

Le problème est alors de choisir quel sous-terme compte pour interpréter notre terme avec fermetures. En d'autres termes, comment interpréter un terme $(\lambda_{\langle S,t\rangle}T\cdot N)$, pour que l'interprétation de $[S.\,N]t$ en soit un réduit ? On ne peut choisir la version partiellement réduite T (i.e. $[\![\lambda_{\langle S,t\rangle}T]\!] = \lambda[\![T]\!]$), car alors la β-réduction, qui emploie la fermeture, pourrait nous faire "reculer" dans la réduction (au vu de l'invariant précédent, $\{[\![S.\,N]\!]\}t = (\{\Uparrow^1 S\}t)\{0\backslash[\![N]\!]\}$ a peu de chances d'être un réduit de $(\lambda[\![T]\!]\ [\![N]\!])$). Choisir la fermeture pour interpréter l'abstraction pose le problème qu'alors les réductions faites sous l'abstraction ne donnent pas lieu à un pas de réduction dans l'interprétation vers un terme pur. Cela ne garantit pas la normalisation forte du terme avec fermetures en présence de réduction sous les abstractions. La solution adoptée est de paramétrer l'interprétation par un entier i qui permettra d'observer les réductions faites sous i abstractions.

3.6.1 Interprétation

Définition 3.16 (`tsub, intf, ints, intft`) *L'interprétation des termes avec fermeture vers les termes purs (figure 3.5) consiste essentiellement à effectuer les substitutions. Cette interprétation est définie simultanément avec l'interprétation des substitutions vers les fonctions des entiers vers les termes purs. On définit aussi $[\![M]\!]_\infty$, qui consiste à toujours interpréter $\lambda_{\langle S,t\rangle}T$ à l'aide de T.*

$$\frac{}{\overline{0} \in \mathcal{I}} \qquad \frac{T \in \mathcal{I}}{\uparrow^k T \in \mathcal{I}}$$

$$\frac{M \in \mathcal{I} \qquad N \in \mathcal{I} \qquad [\![(M \cdot N)]\!]_0 \in \mathcal{SN}^\beta}{(M \cdot N) \in \mathcal{I}}$$

$$\frac{T \in \mathcal{I} \qquad [\Uparrow^1 S]t \in \mathcal{I} \qquad \{[\![\Uparrow^1 S]\!]\}t \triangleright^*_\beta [\![T]\!]_0}{\lambda_{\langle S,t\rangle} T \in \mathcal{I}}$$

$$\frac{U \in \mathcal{I} \qquad [S]t \to_{\phi^c} U}{[S]t \in \mathcal{I}}$$

FIG. 3.6: Invariant de cohérence des abstractions

Lemme 3.17 (`eq_subst_cons_rec`) *On montre un résultat d'équivalence entre interprétation et substitution :*

$$\{[\![\uparrow^n S.\, N]\!]\}t = (\uparrow_1^n \{[\![\Uparrow^1 S]\!]\}t)\{0 \backslash [\![N]\!]_0\}$$

Preuve Par récurrence sur t. Pour traiter le cas de l'abstraction, il faut renforcer l'énoncé de la manière suivante :

$$\{[\![\Uparrow^k (\uparrow^n S.\, N)]\!]\}t = (\uparrow_{k+1}^n \{[\![\Uparrow^{k+1} S]\!]\}t)\{k \backslash [\![N]\!]_0\}$$

■

3.6.2 Invariant

Nous allons définir la contrepartie des termes fortement normalisables dans le monde des termes avec fermetures en définissant un *invariant*. On montrera que cet ensemble de termes est clos par $\lambda\phi^c$ et que l'interprétation d'un terme vérifiant l'invariant est fortement normalisable. Une condition assurée par l'invariant est la cohérence entre les deux branches de l'abstraction déjà citée.

Définition 3.18 (prédicat `invar`) *L'ensemble des termes invariants est le plus petit ensemble de termes clos par les règle de la figure 3.6.*

Attention : la réduction avec partage pourrait briser l'invariant, car par effet de bord, elle fait des réductions dans les substitutions. Mais on a toujours que les deux branches de l'abstraction ont des interprétations convertibles.

Lemme 3.19 (`invar_intf_le`, `invar_intft`) *La suite des interprétations d'un terme invariant forme un suite de réduits :*

$$\begin{aligned} M \in \mathcal{I} \ \wedge \ i \leq j \quad &\Rightarrow \quad [\![M]\!]_i \triangleright^*_\beta [\![M]\!]_j \\ M \in \mathcal{I} \quad &\Rightarrow \quad [\![M]\!]_i \triangleright^*_\beta [\![M]\!]_\infty \end{aligned}$$

Preuve Par récurrence sur $j - i$, on se ramène à prouver $[\![M]\!]_i \triangleright^*_\beta [\![M]\!]_{i+1}$ pour $M \in \mathcal{I}$. Ceci se prouve par récurrence sur M. Le cas de l'abstraction utilise l'invariant. ■

Lemme 3.20 (`invar_sn_n`) *Tout terme vérifiant l'invariant a une interprétation de niveau 0 fortement normalisable (et elles le sont toutes grâce au lemme précédent) :*

$$M \in \mathcal{I} \ \Rightarrow \ [\![M]\!]_i \in \mathcal{SN}^\beta$$

Preuve Par récurrence sur la dérivation de $M \in \mathcal{I}$. Le cas de l'application est trivial du fait de l'invariant. L'abstraction utilise le résultat que si t est fortement normalisable, alors λm l'est aussi. ■

Pour le résultat de correction et de normalisation, on montre que notre système simule bien la β-réduction :

Lemme 3.21 (`intf_sigma_ctxt`, `intf_beta_ctxt_le`, `intf_beta_ctxt`)

$$\begin{array}{rcl} M \in \mathcal{I} \ \wedge \ M \to_{[\phi^c]} N & \Rightarrow & [\![M]\!]_i \rhd_\beta^* [\![N]\!]_i \\ M \to_{[\bar{\beta}]_j} N \ \wedge \ i \leq j & \Rightarrow & [\![M]\!]_i \rhd_\beta^* [\![N]\!]_i \\ M \to_{[\bar{\beta}]_i} N & \Rightarrow & [\![M]\!]_i \rhd_\beta [\![N]\!]_i \end{array}$$

Preuve Ces résultats se prouvent séparément, par récurrence sur l'hypothèse de réduction entre M et N. ■

Les deux premiers résultats indiquent que les réductions faites au niveau des termes avec abstractions vont donner lieu à des réductions au niveau des termes purs. Le troisième dit en plus que si l'on fait fait un pas de $\bar{\beta}$-réduction sous i abstractions, alors l'interprétation de niveau i est β-réduite d'un pas aussi. Cela permettra de montrer que l'on n'a pas de réductions infines au niveau des termes avec fermetures, puisque l'on pourrait alors former une suite infinie de réduits.

Comme autre remarque, on pourrait être étonné que ϕ^c ne s'interprète pas par l'identité ($M \to_{[\phi^c]} N \Rightarrow [\![M]\!]_i = [\![N]\!]_i$), puisque l'interprétation était supposée effectuer les substitutions.

Cela n'est pas vrai à cause indirectement de notre abstraction redondante, mais plus directement de la règle VarCons0, puisque :

$$[\![[\uparrow^k S.\, T]0]\!]_i = [\![T]\!]_0 \rhd_\beta^* [\![T]\!]_i$$

Ceci n'étant valable que si T vérifie l'invariant.

Ainsi, un pas de $\bar{\beta}$-réduction fait décroître strictement l'interprétation de niveau i, et au sens large celles de niveau inférieur. Celles des niveaux plus grands peuvent ne pas décroître. Par exemple, une $\bar{\beta}$-réduction au sommet du terme a peu de chance de donner lieu à des interprétations de niveau un que se réduisent l'une vers l'autre (ce qui serait le cas si l'on pouvait montrer l'étape $\to_?$:

$$\frac{(\lambda_{\langle S,t\rangle} T \cdot N) \to_{[\bar{\beta}]_0} [\uparrow^0 S.\, N]\, t}{[\![(\lambda_{\langle S,t\rangle} T \cdot N)]\!]_1 = (\lambda [\![T]\!]_0\ [\![N]\!]_1) \to [\![T]\!]_0\{0 \backslash [\![N]\!]_1\} \to_?\ \{[\![\Uparrow^1 S]\!]\} t \{0 \backslash [\![N]\!]_0\}}$$

L'invariant nous permettrait plutôt de prouver la réduction contraire, puisque $\{[\![\Uparrow^1 S]\!]\} t \to [\![T]\!]_0$ et $[\![N]\!]_0 \to [\![N]\!]_1$.

Lemme 3.22 (`invar_lamphic`) *L'invariant est préservé par réduction (pas de réduction dans les substitutions) :*

$$M \to_{\lambda\phi^c} N \ \wedge \ M \in \mathcal{I} \ \Rightarrow \ N \in \mathcal{I}$$

Preuve Par récurrence sur $M \to \lambda\phi^c N$. La seule règle qui qui crée des abstractions avec fermetures est celle qui propage les substitutions à travers l'abstraction pure, et celle-ci vérifie clairement l'invariant. ■

Ce qui permet de conclure à la correction de notre système de réduction par rapport au λ-calcul pur :

Lemme 3.23 (`lamphic_interp`)

$$M \to_{\lambda\phi^c} N \ \wedge \ M \in \mathcal{I} \ \Rightarrow \ [\![M]\!]_0 \triangleright_\beta^* [\![N]\!]_0$$

3.6.3 Correction des formes normales

Nous avons prouvé que notre système permettait bien de calculer des réduits de termes purs. Il reste maintenant à prouver que l'on calcule bien la forme normale, et enfin que notre stratégie termine (section 3.6.4).

Le schéma habituel de la preuve serait de montrer que notre calcul permet de rendre compte de n'importe quelle β-réduction faite sur l'interprétation d'un terme M. Mais ceci n'est pas vrai dans notre calcul. Tout ce que nous pouvons prouver, c'est que s'il y a un radical dans l'interprétation, alors il y a un $\lambda\phi^c$-radical dans M, mais il n'est pas sûr qu'il existe un réduit de M dont l'interprétation serait le réduit de l'interprétation de M.

Cela signifie que l'on ne peut pas implanter n'importe quelle stratégie du λ-calcul pur. Cela n'est pas grave, puisque tout ce qui nous intéresse est de prouver que si $[\![T]\!]_\infty$ contient un radical, alors T aussi.

Théorème 1 (`nf_sound`)

$$[\![T]\!]_\infty \triangleright_\beta u \Rightarrow \exists U.\, T \to_{\lambda\phi^c} U$$

Ce résultat assure que si l'on calcule la forme $\lambda\phi^c$-normale de T, son interprétation de niveau ∞ est une forme normale.

3.6.4 Normalisation faible

Pour montrer la normalisation de $\lambda\phi^c$ lorsque l'on ne réduit pas sous les substitutions, nous allons définir une autre fonction d'interprétation $[\![]\!]^{\lambda\phi^c}$.

Définition 3.24 (`interp`) *On interprète un terme par la suite de ses interprétation de niveau i. La deuxième composante servira à tenir compte des ϕ^c-réductions (propagation des substitutions).*

$$[\![M]\!]^{\lambda\phi^c} \ \stackrel{\text{def}}{\equiv} \ (i \mapsto [\![M]\!]_i, M)$$

Définition 3.25 (prédicat `ord`**)** *On ordonne ces interprétations selon l'ordre lexicographique (voir l'annexe A pour une définition de $\prec_i^R$).*

$$(f_1, M) \ll^i (f_2, N) \ \stackrel{\text{def}}{\equiv} \ (f_2 \prec_i^{\triangleright_\beta} f_1) \vee ((\forall k.\, f_2(k) \triangleright_\beta^* f_1(k)) \wedge N \to_{[\phi^c]} M)$$

Un élément (f, M) est accessible pour $\ll^i$ à partir du moment où tous les f_j tels que $j < i$ sont accessibles pour $\triangleright_\beta^{-1}$ (i.e. ils sont fortement normalisables), en utilisant le résultat A.3. Il nous reste à trouver, pour tout terme M une borne i pour lequel $\ll^i$ décroît strictement pour toute réduction d'un de ses réduits. Le lemme 3.21 va nous aider. On peut en déduire que les ϕ^c-réductions font décroître n'importe quel $\ll^i$ (selon la deuxième clause, à partir du moment où M est invariant). Il suffit de trouver un majorant au nombre d'abstractions des réduits de M pour être sûr que toutes les $\bar{\beta}$-réductions feront décroître strictement l'interprétation (selon la première clause).

Définition 3.26 (`maj_depth`) *Nous définissons les niveaux d'interprétations idempotents comme la hauteur maximale (en ne comptant que les abstractions, notation $|t|_\lambda$) que peut avoir un réduit. Cette définition concerne les termes purs.*

$$\mathrm{Maj}(t) \stackrel{\text{def}}{\equiv} \left\{ n \;\middle|\; \forall u.\, t \triangleright_\beta^* u \;\Rightarrow\; |u|_\lambda \leq n \right\}$$

$$|n|_\lambda = 0 \qquad |(m\ n)|_\lambda = \max(|m|_\lambda, |n|_\lambda) \qquad |\lambda t|_\lambda = 1 + |t|_\lambda$$

On appelle ces entiers les niveaux d'interprétations idempotents car on peut montrer que pour tout terme M, si m et n appartiennent à $\mathrm{Maj}([\![M]\!]_0)$, alors $[\![M]\!]_m = [\![M]\!]_n$.

Lemme 3.27 (`maj_depth_ex`) *Pour tout terme pur fortement normalisable, cet ensemble n'est pas vide, puisque ces termes ont un nombre fini de réduits.*

$$\forall t \in \mathcal{SN}^\beta_\cdot\, \exists n.\, n \in \mathrm{Maj}(t)$$

Lemme 3.28 (`sn_intf`) *Tout terme invariant M tel que son interprétation est accessible pour l'ordre considéré à un rang idempotent est fortement normalisable :*

$$n \in \mathrm{Maj}([\![M]\!]_0) \;\wedge\; [\![M]\!]^{\lambda\phi^c} \in \mathrm{Acc}_{\ll^{n+1}} \;\wedge\; M \in \mathcal{I} \Rightarrow M \in \mathcal{SN}^{\lambda\phi^c}$$

Preuve Il suffit de montrer que chaque pas de $\lambda\phi^c$-réduction fait décroître l'ordre en question, ce que nous avons déjà prouvé. ■

Théorème 2 (`sn_invar`) *Terminaison du système (aucun pas de réduction dans les substitutions) :*

$$\mathcal{I} \subseteq \mathcal{SN}^{\lambda\phi^c}$$

Preuve Il suffit d'utiliser le lemme précédent. Le fait que M est invariant nous assure que $[\![M]\!]_0$ est fortement normalisable, et donc il existe un niveau d'interprétation idempotent pour M. Le lemme A.3 permet de conclure puisque nous pouvons prouver que pour tout i, l'interprétation $[\![M]\!]_i$ est accessible pour $\triangleright_\beta^{-1}$. ■

3.6.5 Résultat principal

Il nous reste à justifier que pour normaliser un terme pur t, il suffit de prendre l'interprétation de rang ∞ de la forme normale de $[\mathrm{id}]t$. En effet, le terme de départ s'interprète bien par t :

Lemme 3.29 (`intf_id`)

$$[\![[\mathrm{id}]t]\!]_i = t$$

Preuve On prouve en fait que $[\![[\Uparrow^n \mathrm{id}]t]\!]_i = t$, par récurrence sur t, sinon le cas de l'abstraction ne passe pas la récurrence. ■

Lemme 3.30 (`init_invar`) *Le terme $[\mathrm{id}]t$ vérifie l'invariant à partir du monment où t est fortement normalisable :*

$$t \in \mathcal{SN}^\beta \;\Rightarrow\; [\mathrm{id}]t \in \mathcal{I}$$

On a montré que $\bar{\beta}\phi^c$ simulait la β-réduction, donc la forme $\lambda\phi^c$-normale de $[\mathrm{id}]t$ est un réduit de t. De plus, il est en forme β-normale car s'il se réduisait, le terme T serait lui aussi réductible (lemme 1), ce qui est absurde. On sait de plus que le terme T ne contient plus aucune fermeture : il ne reste donc plus qu'à calculer les relocations grâce à $[\![]\!]_\infty$.

Algorithme 3.31 (`reduction_function`) *Si on a un algorithme de normalisation des termes normalisables pour $\bar{\beta}\phi^c$, alors on peut en déduire un algorithme de normalisation des termes purs fortement normalisables.*

$$\forall T \in \mathcal{SN}^{\lambda\phi^c}_\cdot\, \exists^\star U.\, T \to_{\lambda\phi^c} U \;\wedge\; U \in \mathcal{NF}^{\lambda\phi^c}$$
$$\Rightarrow \forall t \in \mathcal{SN}^\beta.\, \exists^\star u.\, t \triangleright_\beta^* u \;\wedge\; u \in \mathcal{NF}^\beta$$

Il est très facile d'écrire un algorithme pour $\lambda\phi^c$-normaliser les termes puisque $\lambda\phi^c$ est un système de premier ordre : on détecte les radicaux par simple filtrage, et les réduits peuvent se calculer de manière atomique (en temps constant). Nous n'avons pas à nous soucier de la terminaison puisque nous ne réduisons que des termes fortement normalisables (en excluant les réductions sous les substitutions).

Dans la section suivante, on va décrire comment est implanté la fonction de normalisation des termes avec fermetures. La difficulté sera de produire une fonction récursive terminale, et de gérer le partage à l'aide d'effets de bord.

3.7 Description de la machine abstraite

Note : cette section, qui décrit plus précisément l'implantation qui se trouve en annexe B.6, mérite d'être améliorée. Les détails des machines qui y sont présentées sont susceptibles d'être légèrement modifiées.

3.7.1 Parcours récursif terminal des termes

On va devoir écrire des fonctions qui parcourent des termes. La solution la plus simple est d'utiliser la récursivité, mais ce n'est pas très bon vis-à-vis de la pile. Il est en général meilleur d'écrire des fonctions récursives terminales, qui peuvent être compilées de façon à ne pas consommer de place sur la pile. En fait, cela revient à remplacer la pile d'appels récursifs par une structure appelée zipper (voir [33]).

Définition 3.32 *Nous définissons le type des contextes de termes. Dans le cas des zippers, le chemin part de l'occurrence et se dirige vers la racine.*

$$Z \quad := \quad \Box \mid \lambda_{\langle S,t\rangle}\Box :: Z \mid (\Box \cdot T) :: Z \mid (T \cdot \Box) :: Z \mid \uparrow^n \Box :: Z$$

Le zipper indique où on se trouve dans le terme (une autre façon est de considérer qu'il permet de reconstruire le terme entier) : $\Box$ signifie qu'on est au sommet, $(\Box \cdot T) :: Z$ signifie qu'on est descendu à gauche d'une application dont le sous-terme droit est T, et que cette application est elle même dans le contexte Z.

On interprète un zipper par une fonction qui reconstruit un certain contexte autour d'un terme M :

$$\begin{aligned}
[\![\Box]\!](M) &= M \\
[\![\lambda_{\langle S,t\rangle}\Box :: Z]\!](M) &= [\![Z]\!](\lambda_{\langle S,t\rangle} M) \\
[\![(\Box \cdot N) :: Z]\!](M) &= [\![Z]\!]((M \cdot N)) \\
[\![(N \cdot \Box) :: Z]\!](M) &= [\![Z]\!]((N \cdot M)) \\
[\![\uparrow^n \Box :: Z]\!](M) &= [\![Z]\!](\uparrow^n M)
\end{aligned}$$

Les zippers permettent de faire des parcours récursifs terminaux des termes. Comme premier exemple, les règles de transition de la figure 3.7 permettent de faire une copie d'un terme, en parcourant les arbres de la gauche vers la droite. Pour faire une copie d'un terme t, il suffit de faire toutes les transitions possibles à partir de $(t, \Box)$, le parcours étant déterministe. La flèche indique si nous sommes en train de descendre ou remonter dans le terme.

Un exemple plus intéressant est la traduction finale des termes avec *shifts* explicites vers la structure des termes purs. On adapte la machine précédente sur plusieurs points :

$$
\begin{array}{rcl}
(\downarrow,\ (A \cdot B),\ Z) & \to & (\downarrow,\ A,\ (\Box \cdot B) :: Z) \\
(\downarrow,\ \lambda_{\langle S,t\rangle} T,\ Z) & \to & (\downarrow,\ T,\ \lambda_{\langle S,t\rangle} \Box :: Z) \\
(\downarrow,\ \overline{0},\ Z) & \to & (\uparrow,\ \overline{0},\ Z) \\
(\uparrow,\ T,\ (\Box \cdot B) :: Z) & \to & (\downarrow,\ B,\ (T \cdot \Box) :: Z) \\
(\uparrow,\ T,\ (A \cdot \Box) :: Z) & \to & (\uparrow,\ (A \cdot T),\ Z) \\
(\uparrow,\ T,\ \lambda_{\langle S,t\rangle} \Box :: Z) & \to & (\uparrow,\ \lambda_{\langle S,t\rangle} T,\ Z)
\end{array}
$$

FIG. 3.7: Parcours récursif terminal d'un terme

$$
\begin{array}{rcl}
(\downarrow,\ (L, (A \cdot B)),\ Z) & \to & (\downarrow,\ (L, A),\ (\Box \cdot (L, B)) :: Z) \\
(\downarrow,\ (L, \lambda_{\langle S,t\rangle} T),\ Z) & \to & (\downarrow,\ (\Uparrow L, T),\ \lambda_{\langle S,t\rangle} \Box :: Z) \\
(\downarrow,\ (L, \uparrow^k T),\ Z) & \to & (\downarrow,\ (L \circ \uparrow^k, T),\ Z) \\
(\downarrow,\ (L, \overline{0}),\ Z) & \to & (\uparrow,\ L(0),\ Z) \\
(\uparrow,\ t,\ (\Box \cdot (L, B)) :: Z) & \to & (\downarrow,\ (L, B),\ (t \cdot \Box) :: Z) \\
(\uparrow,\ t,\ (a \cdot \Box) :: Z) & \to & (\uparrow,\ (a\ t),\ Z) \\
(\uparrow,\ t,\ \lambda_{\langle S,t'\rangle} \Box :: Z) & \to & (\uparrow,\ \lambda t,\ Z)
\end{array}
$$

FIG. 3.8: Calcul des relocations explicites

- on ajoute un argument supplémentaire aux termes : c'est la composition des relocations rencontrées depuis le sommet du terme ;
- le zipper contient des données de types hétérogènes car on a en entrée un terme et une relocation, alors qu'en sortie, on a un terme pur. Quand on descend à gauche d'une application, on n'a pas encore relogé le terme de droite, donc on met dans le zipper le terme et la relocation. Quand on redescend à droite, on a déjà relogé le sous-terme gauche, donc on met ce terme relogé dans le zipper.

Ces modifications donnent lieu à la déclaration de type `stk` de l'annexe B.5.

Cela nous donne la machine de la figure 3.8 (voir la fonction `to_lambda`, annexe B.6). Les relocations sont des fonctions des entiers vers les entiers dont les opérations sont définies par :

$$
\begin{array}{rcl}
\Uparrow L(0) & = & 0 \\
\Uparrow L(k{+}1) & = & 1 + L(k) \\
L \circ \uparrow^k (n) & = & L(k+n) \\
\mathrm{id}(n) & = & n
\end{array}
$$

On démarre avec $(\downarrow,\ (\mathrm{id}, M),\ \Box)$, et si le terme ne comporte aucune fermeture, la machine s'arrête dans l'état $(\uparrow,\ t,\ \Box)$, t étant le résultat recherché. Si le terme initial contient des fermetures, la machine se bloque. Elle est censée n'être appelée que lorsqu'on a terminé

la réduction, et alors il n'y a plus de fermetures dans le terme : il ne reste plus que les opérateurs du λ-calcul et les shifts.

3.7.2 Les machines abstraites

Le principe de la machine est de partir du sommet du terme, avec le zipper $\Box$. On descend dans le terme, en accumulant les arguments des applications dans le zipper à l'aide du constructeur $(\Box \cdot T)$, et en réduisant les radicaux rencontrés, comme cela se fait avec la machine de Krivine. Lorsqu'on arrive à une feuille en forme normale, on essaie de reconstruire le terme. Sauf si l'un des voisins n'est pas encore normalisé, auquel cas on recommence avec ce terme. À la fin, on se retrouve au sommet du terme. Comme nous n'avons reconstruit que des termes en forme normale, nous savons que nous avons calculé la forme normale du terme initial.

L'état de la machine KLN est un couple formé d'un terme et d'un zipper. On raffine un peu en séparant le cas où le terme est une fermeture (ce qui nous évite d'allouer une fermeture que nous décomposons immédiatement après). La conséquence est l'introduction d'une autre machine KLNT, dont l'état est un triplet (substitution, terme pur, zipper). Nous avons vu que l'on avait besoin de rajouter un booléen (la flèche dans les exemples précédents) indiquant si l'on est en train de remonter ou de descendre. Cela se fait en introduisant encore une autre machine ZIP servant à remonter dans le terme, alors que KLN et KLNT servent à descendre.

Nous allons présenter les transitions de la machine KLN, qui calcule la forme normale de tête faible d'une fermeture. Il y a trois types de transitions : celles qui font de la β réduction, celles qui font avancer les substitutions, et celles qui font avancer dans le parcours du terme.

Comme pour la machine KN de Crégut, nous faisons l'optimisation de ne jamais enfermer une simple variable dans une fermeture, en allant directement chercher la valeur dans la substitution. Non seulement cela améliore la complexité en temps, comme nous l'avons déjà remarqué, mais en plus cela permet au GC d'Objective Caml de mieux travailler, puisqu'au lieu d'avoir une fermeture qui contient un pointeur vers la substitution entière, nous n'avons qu'un pointeur vers l'un des éléments de la susbstitution. La définition de cette fonction est la suivante :

$$\begin{aligned}
\langle m, S, \lambda t\rangle &= \uparrow^m [S]\lambda t \\
\langle m, S, (M\ N)\rangle &= \uparrow^m [S](M\ N) \\
\langle m, \mathrm{id}, n\rangle &= \uparrow^{m+n}\overline{0} \\
\langle m, \uparrow^k S.\, T, 0\rangle &= \uparrow^m T \\
\langle m, \uparrow^k S.\, T, n{+}1\rangle &= \langle m+k, S, n\rangle
\end{aligned}$$

L'entier sert à accumuler les relocations pour éviter d'engendrer plusieurs *shifts*, qu'il faudra ensuite simplifier.

La première règle de KLN (figure 3.9, les règles se lisant comme un filtrage) donne la main à KLNT lorsque l'on arrive sur une fermeture. Les quatre règles suivantes gèrent la descente dans le terme avec simplification des relocations. Les deux dernières règles font la β-réduction, avec ou sans *shift* intercalé. La machine KLN s'arrête sur un terme normal ou sur une abstraction non appliquée.

La machine KLNT ne se bloque jamais, et elle finit toujours par rappeler KLN. Les 2 premières règles font la propagation de la substitution et donnent immédiatement la main à KLN. La troisième règle traite le cas des variables. L'expansion des variables a déjà été traitée lorsque l'on crée une fermeture quelconque. Il suffit de s'en servir.

$$
\begin{array}{rcl}
(S,\, \lambda t,\, Z)_{\mathrm{klnt}} & \to & (\lambda_{\langle S,t\rangle}\, \langle 0, \Uparrow^1 S, t\rangle\,,\, Z)_{\mathrm{kln}} \\
(S,\, (M\ N),\, Z)_{\mathrm{klnt}} & \to & (S,\, M,\, (\Box \cdot \langle 0, S, N\rangle) :: Z)_{\mathrm{klnt}} \\
(S,\, n,\, Z)_{\mathrm{klnt}} & \to & (\langle 0, S, n\rangle\,,\, Z)_{\mathrm{kln}} \\
 & & \\
([S]t,\, Z)_{\mathrm{kln}} & \to & (S,\, t,\, Z)_{\mathrm{klnt}} \\
(\uparrow^n T,\, \uparrow^m \Box :: Z)_{\mathrm{kln}} & \to & (T,\, \uparrow^{m+n} \Box :: Z)_{\mathrm{kln}} \\
(\uparrow^0 T,\, Z)_{\mathrm{kln}} & \to & (T,\, Z)_{\mathrm{kln}} \\
(\uparrow^n T,\, Z)_{\mathrm{kln}} & \to & (T,\, \uparrow^n \Box :: Z)_{\mathrm{kln}} \\
((M \cdot N),\, Z)_{\mathrm{kln}} & \to & (M,\, (\Box \cdot N) :: Z)_{\mathrm{kln}} \\
(\lambda_{\langle S,t\rangle} T,\, (\Box \cdot M) :: Z)_{\mathrm{kln}} & \to & (\uparrow^0 S.\, M,\, t,\, Z)_{\mathrm{klnt}} \\
(\lambda_{\langle S,t\rangle} T,\, \uparrow^n \Box :: (\Box \cdot M) :: Z)_{\mathrm{kln}} & \to & (\uparrow^n S.\, M,\, t,\, Z)_{\mathrm{klnt}}
\end{array}
$$

FIG. 3.9: Les machines KLN et KLNT

Dans l'implantation, nous annotons nos termes avec un booléen indiquant si l'on a déjà atteint la forme normale d'un terme (voir plus bas), ce qui évite de le reparcourir pour s'en assurer. Cela donne lieu à des règles qui permettent de couper court :

$$
\begin{array}{rcll}
((M \cdot N),\, Z)_{\mathrm{kln}} & \to^* & ((M \cdot N),\, Z)_{\uparrow} & \text{si } (M \cdot N) \text{ normal} \\
(\lambda_{\langle S,t\rangle} T,\, Z)_{\mathrm{kln}} & \to^* & (\lambda_{\langle S,t\rangle} T,\, Z)_{\uparrow} & \text{si } T \text{ normal} \\
(M,\, (\Box \cdot N) :: Z)_{\uparrow} & \to^* & ((M \cdot N),\, Z)_{\uparrow} & \text{si } N \text{ normal}
\end{array}
$$

On pourrait vérifier que chacune de ces règles s'expanse en la succession de transitions de la machine. Ce sont en quelque sorte des règles dérivées.

Les règles décrites ci-dessus permettent de calculer la forme normale de tête faible d'un terme (elle s'arrête sur les abstractions non appliquées et les variables libres). Pour poursuivre de la réduction sous les abstractions, et initier la reconstruction du terme, nous introduisons les règles :

$$
\begin{array}{rcl}
(\lambda_{\langle S,t\rangle} T,\, Z)_{\mathrm{kln}} & \to & (T,\, \lambda_{\langle S,t\rangle} \Box :: Z)_{\mathrm{kln}} \\
(\overline{0},\, Z)_{\mathrm{kln}} & \to & (\overline{0},\, Z)_{\uparrow} \\
(M,\, (\Box \cdot N) :: Z)_{\uparrow} & \to & (N,\, (M \cdot \Box) :: Z)_{\mathrm{kln}} \\
(M,\, (N \cdot \Box) :: Z)_{\uparrow} & \to & ((N \cdot M),\, Z)_{\uparrow} \\
(M,\, \uparrow^k \Box :: Z)_{\uparrow} & \to & (\uparrow^k M,\, Z)_{\uparrow} \\
(M,\, \lambda_{\langle S,t\rangle} \Box :: Z)_{\uparrow} & \to & (\lambda_{\langle S,t\rangle} M,\, Z)_{\uparrow}
\end{array}
$$

La première règle (que l'on applique uniquement si $Z \neq (\Box \cdot M) :: Z'$ et $Z \neq \uparrow^n \Box :: (\Box \cdot M) :: Z'$, auquel cas on applique les règles du premier tableau) sert à descendre sous les abstractions. La troisième sert à normaliser le sous-terme droit d'une application une fois que le sous-terme gauche a été normalisé sans faire apparaître d'abstraction. Enfin, les trois dernières reconstruisent le terme, lorsque tous ses sous-termes ont été normalisés.

On utilise l'interprétation suivante pour déduire la terminaison de l'algorithme à partir de la terminaison du calcul de substitutions :

$$\begin{aligned}
[\![(S,\, t,\, Z)_{\mathrm{klnt}}]\!] &= [\![Z]\!]([S]t) \\
[\![(M,\, Z)_{\mathrm{kln}}]\!] &= [\![Z]\!](M) \\
[\![(M,\, Z)_{\uparrow}]\!] &= [\![Z]\!](M)
\end{aligned}$$

Le résultat est que chaque transition décrite dans cette section correspond soit à une réduction de l'interprétation, soit à avancer dans le zipper. Comme ce dernier ordre termine et commute avec la réduction de l'interprétation, notre algorithme termine.

3.7.3 Implantation avec effets de bord

L'implantation utilise le partage et fait des effets de bord. Une autre amélioration consiste à annoter chaque nœud du terme avec l'état du terme correspondant : en forme normale, ou susceptible d'être réduit. Voir le source complet, annexe B.6 (moins de 200 lignes, tout compris).

Définition 3.33 *Termes modifiables (le #) avec relocations et substitutions explicites :*

$$\begin{aligned}
T &:= \#a = (T_0,\, F) \\
T_0 &:= \overline{0} \mid \lambda_{\langle S,t\rangle} T \mid (T \cdot T') \mid \uparrow^n T \mid [S]t \\
S &:= \mathrm{id} \mid S.\, T \mid \uparrow^n S \mid \Uparrow^n S
\end{aligned}$$

avec $n \in \mathbb{N}$ *et* $F \in \{\mathrm{Red}, \mathrm{Norm}\}$.

Red signifie que le terme peut contenir des radicaux, et Norm qu'il est en forme normale.

Pour modéliser les termes sur lesquels on peut faire des effets de bord, nous introduirons des notations pour les pointeurs : $\#a = (T,\, F)$ désigne l'adresse a, dont le contenu est la paire $(T,\, F)$. L'expression $a := (T,\, F)$ est l'affectation du terme T à l'adresse a.

Il suffit d'ajouter les règles suivantes à KLN (KLNT est inchangée) :

$$\begin{aligned}
(\#a = (M,\, \mathrm{Red}),\, Z)_{\mathrm{kln}} &\to (M,\, \#a :: Z)_{\mathrm{kln}} && \mathsf{si}\ M \neq \lambda \\
(\#a = (\lambda_{\langle S,t\rangle} T,\, \mathrm{Red}),\, Z)_{\mathrm{kln}} &\to (\lambda_{\langle S,t\rangle} T,\, Z)_{\mathrm{kln}} && \\
(\#a = (M,\, \mathrm{Norm}),\, Z)_{\mathrm{kln}} &\to (M,\, Z)_{\mathrm{kln}} && \\
(M = \lambda_{\langle S,t\rangle} T,\, \#a :: Z)_{\mathrm{kln}} &\to (M,\, Z)_{\mathrm{kln}} && a := (M,\, \mathrm{Red}) \\
(M = \lambda_{\langle S,t\rangle} T,\, \uparrow^k \Box :: \#a :: Z)_{\mathrm{kln}} &\to (M,\, \uparrow^k \Box :: Z)_{\mathrm{kln}} && a := (\uparrow^k M,\, \mathrm{Red}) \\
(M,\, \#a :: Z)_{\uparrow} &\to (M,\, Z)_{\uparrow} && a := (M,\, \mathrm{Norm})
\end{aligned}$$

Lorsque l'on déréference un terme qui n'est pas encore en forme normale de tête faible, on insère une marque de mise à jour dans le zipper, comme cela est fait dans la thèse de Crégut [17].

On n'engendre une marque de mise à jour uniquement lorsque le terme peut ne pas être en forme normale de tête. Dès que la machine KLN obtient une forme normale de tête, i.e. une abstraction ou une variable, on effectue les mises à jour. Ainsi, on est certain de ne pas faire plus de mises à jour que l'on ne crée de fermetures (ce qui est loin d'être le cas dans l'implantation actuelle).

Quelques variantes

Les modifications à apporter au code correspondant aux variantes décrites dans cette section se trouvent dans l'annexe B.5.1.

Nous remarquons que la correction ne repose pas sur le fait qu'il y a partage. Seule l'efficacité en dépend. Si l'on supprime les mises à jour, l'algorithme reste correct. On peut commuter vers une version totalement fonctionnelle en ne modifiant que très peu de code. Si l'on supprime les effets de bords, on réduit selon une stratégie d'appel par nom.

Une autre variante d'implantation serait d'utiliser les *weak pointers* d'Objective Caml (le GC s'autorise à libérer la mémoire occupée par des structures référencées uniquement par ces *weak pointers*) pour les marques de mises à jour stockées dans la pile (le zipper). En effet, si une référence n'est présente que dans la pile, il est inutile de la mettre à jour, puisqu'aucun autre sous-terme n'est susceptible d'en profiter. Les *weak pointers* nous dispensent en quelque sorte d'écrire un GC à l'intérieur de notre interpréteur en utilisant celui d'Objective Caml.

Dans l'exemple de la multiplication d'entiers de Church, sur certaines exécutions, plus de 85% des mises à jour faites sans cette modification peuvent être évitées. Ce chiffre est susceptible de varier d'une exécution à l'autre puisque le comportement du GC n'est pas déterministe.

Bien que cette modification permette de faire beaucoup moins de mises à jour et de mieux gérer l'occupation mémoire, l'algorithme s'avère plus lent car cela requiert toujours un test supplémentaire pour savoir si le pointeur est toujours valide et doit être mis à jour. En revanche, lorsque l'exécution requiert un espace mémoire important, cette optimisation peut être utile. Il pourrait être très avantageux de faire une analyse préliminaire du terme à normaliser pour déterminer les sous-termes qui ne sont pas partagés et pour lesquels il est inutile de faire de mise à jour. Le problème est que cela risque de compliquer l'implantation, et cela nous rapproche de la compilation.

3.8 Conclusions

On a essayé de présenter de manière modulaire un système simple, mais proche de l'implantation. En particulier, nous avons cherché au maximum à séparer la définition d'un système de fermetures de son implantation. Même dans la description de l'implantation à l'aide de machines à environnement, nous avons d'abord présenté des règles totalement fonctionnelles, ce qui donne lieu à une implantation dont la preuve de correction faite dans ce chapitre s'applique formellement.

Les optimisations habituelles, et qui rendent le système vraiment efficace, se sont greffées a posteriori. Cela n'est pas tout à fait exact historiquement, puisque l'implantation fonctionnait par effet de bord dès ses premières versions. En revanche, le parcours exclusivement récursif terminal des termes n'est pas encore implanté dans le système Coq, et laisse espérer encore une amélioration de l'efficacité de la réduction pour les prochaines versions de Coq.

Un point important lors du développement de l'implantation est que l'on a toujours tenu à ce que le système soit correct indépendemment de l'utilisation d'effets de bords qui donnent lieu à des preuves de correction plus complexes, et souvent dépendantes du langage de programmation employé.

La preuve de correction et de terminaison par rapport au λ-calcul fait appel à des techniques de preuve qui mériteraient d'être simplifiées. Il serait intéressant de voir comment adopter la présentation uniforme des preuves de correction exposée dans [29]. Nous rencontrons tout de même un sérieux problème pour définir la fonction de décompilation, du fait du dédoublement de l'abstraction.

C'est un système qui semble avoir des qualités pour effectuer des calculs, mais on ne peut pas vraiment s'en servir pour représenter les termes d'un vérificateur de preuves généraliste à cause des *shifts* explicites qui rendraient la vie impossible lorsque l'on veut faire du filtrage sur les termes.

Deuxième partie

Certification d'un système de type générique

Chapitre 4

Systèmes de Types Purs

Nous allons maintenant aborder la partie centrale de notre objectif, à savoir formaliser le noyau d'un système de preuves. Nous rappelons qu'il s'agit essentiellement de prouver la décidabilité du typage du système de type souhaité, ce que nous n'avons pas encore choisi. Le formalisme doit être suffisamment puissant pour exprimer les arguments logiques courants en mathématiques (raisonnements par récurrence, etc.), mais ce système doit pouvoir s'implanter de manière efficace et être assez commode d'utilisation pour ne pas dérouter ses utilisateurs. Comme nous voulons l'étudier formellement, il faut aussi qu'il ne soit pas trop complexe de présentation.

Les systèmes de types purs, que nous avons présenté section 2.2.3 sont un assez bon compromis entre tous ces critères. Ils permettent d'exprimer des logiques d'ordre supérieur, éventuellement imprédicatives (ce qui permet de définir quelques structures de données simples), et leur présentation est très concise. Les anciennes versions de Coq rentrent dans ce cadre. Ces systèmes se prêtent bien à la formalisation. Les travaux de Pollack sur ce sujet en témoignent [56, 58].

Notre démarche consistera à étendre ces résultats en définissant et formalisant une nouvelle classe de systèmes de types, encore plus vaste que les PTS, et possédant donc plus de paramètres. L'idée est de généraliser la règle de conversion en paramétrant par la relation utilisée pour la conversion. La première utilité de ce paramétrage est de permettre plus facilement l'ajout de la δ-réduction dans le langage. La δ-réduction consiste à remplacer le nom d'une constante par sa définition. On omet souvent cette règle dans les études théoriques en disant simplement que l'on travaille toujours sur les formes totalement expansées. Cela n'est pas un argument en pratique, puisque l'expansion de toutes les définitions serait très inefficace, et nous obligerait à revérifier la validité de ces définitions à chaque utilisation.

Mais il est possible de généraliser en considérant la règle de conversion comme une règle introduisant du sous-typage dans le système (voir fin de section 2.2, page 27). Dans son étude des PTS, Pollack avait remarqué que beaucoup de résultats ne faisaient appel qu'à très peu de propriétés de la β-réduction. En particulier, la symétrie de cette relation ne servait que très peu. Nous allons préciser ce point en déterminant des conditions que devra vérifier notre paramètre supplémentaire, que nous n'appellerons plus règle de conversion, mais règle de sous-typage. C'est pourquoi la classe de systèmes que nous définissons ici s'appelle *Systèmes de Types Purs avec Sous-Typage*.

Le fait d'introduire le sous-typage ne sera pas beaucoup exploité dans cette thèse. La principale utilisation sera de prendre en compte la cumulativité, qui permet d'inclure des sortes les unes dans les autres, ainsi que le marquage des types inductifs afin d'assurer la terminaison des fonctions définies par point fixe.

Lorsque nous aurons terminé cette étude et déterminé des conditions que doit vérifier le sous-typage pour préserver la décidabilité du typage de nos systèmes, nous développerons quelques exemples d'instantiation possible du sous-typage. Le premier exemple sera de l'instancier par la fermeture réflexive symétrique transitive et congruente (i.e. la réduction peut avoir lieu dans n'importe quel sous-terme) d'une règle de réduction. Un autre exemple introduira les systèmes de types cumulatifs, où le sous-typage est la règle de cumulativité, qui sera paramétrée par la règle de réduction que l'on souhaite considérer (β, δ ou autres). Là encore nous établirons des conditions à vérifier pour assurer la décidabilité du typage.

4.1 Méthodologie

Nous allons concevoir notre développement en Coq[1] comme nous faisons les preuves, c'est-à-dire de manière dirigé par le but. Nous détaillerons la manière de développer les preuves lorsque nous prouverons la décidabilité du typage. Pour l'instant, nous en donnerons simplement un aperçu.

La méthode habituelle lorsque l'on attaque une formalisation est de commencer par définir toutes les composantes du système, puis de prouver les résultats métathéoriques dans l'ordre direct, i.e. que chaque théorème repose sur des résultats déjà démontrés. L'idée ici est de s'attaquer immédiatement au théorème que nous souhaitons prouver. Dans notre cas, il s'agit de la décidabilité du sous-typage, ou plus précisément la spécification du noyau de notre système de preuves. La preuve ne peut évidemment pas être complétée, mais elle permet de dégager les théorèmes plus simples à prouver, que nous admettons dans un premier temps. Cela permet de boucler la preuve du théorème principal modulo d'importantes hypothèses. Nous raffinons ensuite ces hypothèses pas à pas. C'est en cela que l'on reproduit au niveau du développement ce que que nous faisions déjà au niveau des preuves grâce aux tactiques.

Lorsque l'ensemble d'hypothèses faites (logiques ou algorithmiques) forme un ensemble cohérent, nous les regroupons tout simplement au sein d'une structure, et les preuves faites jusqu'ici forment un foncteur transformant des structures élémentaires en structures plus complexes, ultimement le noyau. Cela permet de produire une preuve modulaire dont les structures créées forment des interfaces à instancier. Cela permet de dégager l'architecture non seulement de l'implantation, mais aussi de sa preuve de correction. La construction du programme n'est alors qu'un simple emboîtement de ces différents foncteurs.

L'un des inconvénients de la méthode des preuves dirigées par le but est que si l'on n'emploie pas les bonnes tactiques, on peut se retrouver dans une *impasse*, i.e. avec un sous-but impossible à prouver (ce qui n'implique pas nécessairement que le but initial ne l'est pas). Il se produit le même phénomène avec notre méthode.

Un danger qui nous guette est de ne pas se rendre compte qu'une des hypothèses est incompatible avec les autres. Si l'on est un minimum vigilant, il arrivera rarement que l'on fasse une hypothèse qui soit absurde en elle-même, mais il peut arriver qu'une hypothèse soit incompatible avec les autres, c'est-à-dire que la conjonction des hypothèses faites devient absurde. Le problème est que l'on peut ne pas se rendre compte immédiatement de cette incohérence, et c'est uniquement au moment où l'on voudra instancier la structure que nous nous rendrons compte de l'impossibilité de la tâche.

Un bon moyen d'éviter cette mésaventure est de toujours travailler avec un modèle en tête. Dans notre cas, sachant que notre classe de systèmes inclut le Calcul des Constructions, ou le Calcul des Constructions Étendu qui ont déjà été formalisés, il est suffisant de toujours

[1] Le développement correspondant à ce chapitre est disponible à partir de l'URL `http://pauillac.inria.fr/~barras/pts_proofs`.

s'assurer que l'hypothèse que l'on veut rajouter s'applique effectivement au système de types qui sert de modèle. De cette manière, nous ne pouvons échouer, et nous sommes sûrs que nous finirons par dégager des conditions satisfiables. Dans le pire des cas, nous aurons à nous restreindre de manière tellement drastique que seul notre modèle satisfiera les conditions. Cela peut être considéré comme un échec de la méthode et il serait alors préférable d'adopter un style de présentation direct.

Il peut encore nous arriver quelques désagréments : il se peut qu'on fasse des suppositions qui soient vraies, mais à un endroit où il est difficile de les prouver. Par exemple, un résultat comme la normalisation forte utilise en général l'auto-réduction. Si l'on se trouvait dans une situation où le module censé fournir le résultat d'auto-réduction nécessite en entrée le résultat de normalisation forte (éventuellement de manière indirecte), alors il serait difficile d'utiliser ce module, puisqu'il faudrait d'abord prouver l'auto-réduction, puis la normalisation forte. L'application de foncteur permettrait alors d'avoir les résultats accompagnant celui d'auto-réduction, mais ce dernier serait complètement inutile puisqu'il avait fallu le prouver bien avant.

Ce style de développement peut paraître un peu forcé par moments, car par sécurité, nous avons fait certaines parties du raisonnement de manière directe. Cela signifie en fait que ce raisonnement n'est ni pire, ni meilleur que le style habituel. Cependant, il peut être utile dans les cas où l'on a déjà une preuve qu'on souhaite généraliser, comme c'était notre cas avec CC et les PTS.

4.2 Définition du système

Dans cette section, nous définissons une classe de systèmes de types. Tout d'abord, il faut définir le type des termes qui permettent d'écrire les programmes, les propositions et les preuves ; il faut aussi définir une opération particulièrement importante : la substitution.

Puis nous décrirons ce que sont les paramètres de notre classe, parmi lesquels la relation de sous-typage évoquée dans la présentation informelle de ce début de chapitre. À partir de ces paramètres, les règles de typage du système seront introduites.

Seules des propriétés élémentaires des objets définis seront énoncées ici. Les résultats métathéoriques plus conséquents feront l'objet de la section suivante.

4.2.1 Syntaxe des termes

Comme dans les présentations courantes des PTS, on n'introduit qu'une seule classe syntaxique pour tous les objets de notre calcul (preuves, propositions, programmes et types). Ces objets, que l'on appelle λ-termes car ils reprennent les constructeurs du λ-calcul, sont représentés par des arbres ayant deux espèces de feuilles et trois espèces de nœuds.

L'une des espèces de feuilles est l'ensemble des sortes, le premier paramètre des PTS, noté SORT. Comme il a déjà été précisé, les sortes sont des types un peu particuliers. La seule hypothèse que nous faisons sur cet ensemble est que l'égalité de deux sortes est décidable, ce qui ne servira que plus tard, lorsque nous voudrons démontrer la décidabilité du typage de notre système.

Définition 4.1 (type `term`) *Un terme est une sorte, une variable, un produit, une abstraction ou une application. Plus précisément, le type des termes est défini selon la grammaire suivante :*

$$T_1, T_2 : \text{TERM} \quad := \quad s \mid \natural n \mid \Pi\, T_1.\, T_2 \mid \lambda T_1.\, T_2 \mid (T_1\ T_2)$$

avec $s \in$ SORT *et* $n \in \mathbb{N}$.

Dans le développement formel en Coq, cette définition s'écrit à l'aide d'une déclaration inductive similaire aux types concrets d'Objective Caml :

```
Inductive Set term :=
      Srt: sort->term
    | Ref: nat->term
    | Abs: term->term->term
    | App: term->term->term
    | Prod: term->term->term.
```

Cette définition correspond en grande partie à ce que nous avons vu section 2.2.3 : nous retrouvons les sortes, le produit dépendant. La principale différence est l'absence de nom sur les opérateurs lieurs (abstraction et produit), car nous avons opté pour une présentation en notations de de Bruijn, comme dans la section 2.4. Ainsi $\lambda A.\, M$ dénote une fonction dont le domaine est le type A, et dont l'image est M, dans lequel la variable 0 est celle introduite par cette abstraction.

Le choix des indices de de Bruijn nous évite l'épineux problème des renommages. Le prix à payer semble moins grand car au lieu d'avoir à se soucier des renommages à chaque fois que l'on fait une substitution, il suffit en général de montrer que l'on n'introduit que des notions invariantes par renommage. En revanche, la formalisation de Pollack [56] utilise la notation avec noms. Rappelons que notre objectif est de formaliser une implantation, et que les notations de de Bruijn ont été conçues pour simplifier le traitement par l'ordinateur des liaisons de variables, et que cela ne semble pas poser de problèmes d'efficacité. Notre choix paraît donc raisonnable.

Pour définir le jugement de typage, on aura besoin de garder les types des variables dans un contexte. Le contexte sert d'une part à définir des objets *globaux*, comme les différentes constantes de la théorie que l'on souhaite formaliser, et d'autre part à accumuler les variables locales introduites par les lieurs.

Définition 4.2 (types `decl`, `env`) *Les déclarations sont soit une variable dont on donne le type, soit une définition, dont on donne la valeur et le type. Les contextes sont des listes de déclarations :*

$$\begin{array}{rcl} \delta, \gamma : \textsc{Decl} & := & [T] \mid [M : T] \\ \Gamma, \Delta : \textsc{Ctx} & := & [] \mid \Gamma\delta \end{array}$$

avec $M, T \in \textsc{Term}$.

Les notations utilisées seront les suivantes : $|\Gamma|$ est la longueur de la liste Γ, $\Gamma(n)$ désigne le n-ième élément de Γ, le plus à droite ayant l'indice 0. Le type d'une déclaration δ sera noté $\delta.\mathrm{T}$.

Noter que bien que les contextes permettent d'introduire des définitions (où l'on donne un nom à un terme, contrairement aux variables qui supposent qu'un type est habité), la syntaxe des termes ne permet pas de faire de définitions locales, car il n'y a pas de constructeur comme le `let ... in ...` que l'on trouve dans les langages de programmation de la famille ML.

4.2.2 Substitution

À plusieurs reprises dans la suite de développement, nous ferons appel à la notion de *substitution*, qui consiste à remplacer toutes les occurrences d'une variable par un terme. Comme nous avons choisi une présentation basée sur des indices de de Bruijn, la substitution fait elle-même appel à la *relocation*, qui consiste à décaler les indices, pour éviter les captures

Relocation	Substitution
$\uparrow_k^n s = s$	$s\{k\backslash N\} = s$
$\uparrow_k^n \natural i = \begin{cases} \natural(i+n) & \text{si } i \geq k \\ \natural i & \text{si } i < k \end{cases}$	$\natural i\{k\backslash N\} = \begin{cases} \natural(i-1) & \text{si } i > k \\ \uparrow_0^k N & \text{si } i = k \\ \natural i & \text{si } i < k \end{cases}$
$\uparrow_k^n \lambda T.\, M = \lambda \uparrow_k^n T.\ \uparrow_{k+1}^n M$	$(\lambda T.\, M)\{k\backslash N\} = \lambda T\{k\backslash N\}.\, M\{k+1\backslash N\}$
$\uparrow_k^n (A\ B) = (\uparrow_k^n A\ \uparrow_k^n B)$	$(A\ B)\{k\backslash N\} = (A\{k\backslash N\}\ B\{k\backslash N\})$
$\uparrow_k^n \Pi T.\, U = \Pi\ \uparrow_k^n T.\ \uparrow_{k+1}^n U$	$(\Pi T.\, U)\{k\backslash N\} = \Pi T\{k\backslash N\}.\, U\{k+1\backslash N\}$

FIG. 4.1: Définition de la relocation et de la substitution

de variables qui auraient lieu lorsqu'un terme apparaît dans un contexte différent de celui où il apparaissait à l'origine.

Définition 4.3 (`lift_rec`, `lift`, `subst_rec`) *La fonction de relocation d'un terme M de n variables à partir du rang k est notée $\uparrow_k^n M$. Cette fonction augmente de n les variables libres supérieures ou égales à k. La relocation au rang 0 sera notée $\uparrow^n M$. La substitution dans M de la variable k par le terme N est notée $M\{k\backslash N\}$. Les équations définissant ces fonctions sont regroupées figure 4.1.*

Dans la définition formelle de la relocation qui suit, il faut indiquer au système que c'est une définition récursive sur la structure du terme `t`, ce qui garantit que l'on définit une fonction qui termine. En termes mathématiques, cela signifie que la fonction est totale.

```
Fixpoint lift_rec [n:nat; t:term]: nat -> term :=
  [k]Cases t of
       (Srt s) => (Srt s)
     | (Ref i) => (Ref Cases (le_gt_dec k i) of
                         (* k<=i *) (left _)  => (plus n i)
                       | (* k>i  *) (right _) => i
                      end)
     | (Abs a m)  => (Abs (lift_rec n a k) (lift_rec n m (S k)))
     | (App u v)  => (App (lift_rec n u k) (lift_rec n v k))
     | (Prod a b) => (Prod (lift_rec n a k) (lift_rec n b (S k)))
     end.
```

La fonction `le_gt_dec` est un théorème de la librairie standard de Coq qui permet de comparer deux entiers k et i : elle retourne soit `left` accompagné d'une preuve que $k \leq i$, soit `right` et une preuve que $k > i$.

En observant le rang avec lequel sont faits les appels récursifs sur les sous-termes, on en déduit la manière dont sont liées les variables de de Bruijn. Ainsi, on voit que le produit et l'abstraction lient une variable dans leur sous-terme droit (l'appel récursif se fait avec le rang $k+1$), alors qu'ils n'en lient pas dans leur sous-terme gauche (appel récursif avec k). L'application ne lie aucune variable.

Définition 4.4 (`lift_decl`, `subst_decl`) *Les opérations de relocation et de substitution s'étendent directement aux déclarations, et l'on utilisera les notations $\uparrow_k^n \delta$ et $\delta\{k\backslash N\}$ pour toute déclaration δ.*

La relocation et la substitution vérifient un certain nombre de propriétés algébriques, comme par exemple la distributivité de la substitution sur elle-même.

Lemme 4.5 *Les opérations de relocation et de substitution ont les propriétés suivantes :*

$$\begin{aligned}
\uparrow_k^0 M &= M \\
\uparrow_i^p \uparrow_k^n M &= \uparrow_k^{p+n} M && \textit{si } k \leq i \leq k+n \\
\uparrow_i^p \uparrow_k^n M &= \uparrow_{p+k}^n \uparrow_i^p M && \textit{si } i \leq k \\
(\uparrow_k^{n+1} M)\{p\backslash N\} &= \uparrow_k^n M && \textit{si } k \leq p \leq n+k \\
\uparrow_k^n (M\{p\backslash N\}) &= (\uparrow_k^n M)\{n+p\backslash N\} && \textit{si } k \leq p \\
\uparrow_{p+k}^n (M\{p\backslash N\}) &= (\uparrow_{p+k+1}^n M)\{p\backslash \uparrow_k^n N\} \\
M\{p\backslash N\}\{p+n\backslash P\} &= M\{p+n+1\backslash P\}\{p\backslash N\{n\backslash P\}\}
\end{aligned}$$

Ces résultats bien connus avaient déjà été prouvés sous cette forme par Huet dans [32], une formalisation en Coq du λ-calcul pur.

En utilisant le système de grammaires extensibles de Coq, on arrive à formuler les théorèmes de façon plus synthétique, et avec des notations assez proches des notations informelles. Ainsi, on pourra écrire :

```
Lemma commut_lift_subst_rec: (n:nat; N,M:term; p,k:nat)
    (le k p)->[: ^(n/k){p:=N}M :]=[: {!plus n p!:=N}^(n/k)M :].
```

Les délimiteurs `[: :]` et `! !` servent à passer de la notation Coq habituelle à la notation spécifique et réciproquement.

En notation Coq standard, ce lemme serait formulé ainsi :

```
Lemma commut_lift_subst_rec: (n:nat; N,M:term; p,k:nat)
    (le k p)->(lift_rec n (subst_rec N M p) k)
            = (subst_rec N (lift_rec n M k) (plus n p)).
```

Il existe des opérations symétriques de la relocation et de la substitution pour les contextes, mais celles-ci ne sont pas totales, puisqu'il n'est pas possible d'insérer une déclaration à un rang supérieur à la longueur du contexte. Formellement, nous définissons une fonction partielle à l'aide d'une relation (fonctionnelle) qui associe à un contexte son image par relocation.

Définition 4.6 (prédicat `ins_in_env`) *L'insertion d'une déclaration δ dans un contexte est une fonction partielle dont le graphe est la plus petite relation close par application des deux clauses suivantes :*

$$\text{(Ins-0)}\ \frac{}{\mathsf{InsInEnv}(\Gamma_0, \delta, 0, \Gamma_0, \Gamma_0\delta)} \qquad \text{(Ins-S)}\ \frac{\mathsf{InsInEnv}(\Gamma_0, \delta, n, \Gamma, \Gamma')}{\mathsf{InsInEnv}(\Gamma_0, \delta, n+1, \Gamma\delta', \Gamma \uparrow_n^1 \delta')}$$

Un prédicat peut se définir en Coq grâce à une fonction qui retourne une proposition. Il est en général plus commode d'utiliser les prédicats inductifs, ce qui permet de procéder comme en Prolog. La correspondance entre la définition mathématique ci-dessus et la déclaration suivante est assez flagrante.

```
Inductive ins_in_env [g:env; d1:decl]: nat->env->env->Prop :=
   ins_0: (ins_in_env g d1 0 g (cons d1 g))
 | ins_S: (n:nat)(e,f:env)(d:decl)
       (ins_in_env g d1 n e f)
     ->(ins_in_env g d1 (S n) (cons d e) (cons (lift_decl (S 0) d n) f)).
```

Pour comprendre cette définition, on peut dire que $\mathsf{InsInEnv}(\Gamma_0, \delta, n, \Gamma, \Gamma')$ signifie que Γ' est le contexte Γ dans lequel on a inséré la déclaration δ au rang n, Γ_0 étant le contexte dans lequel se trouve inséré δ.

Cette opération est symétrique de la relocation au sens où si un terme M est défini dans le contexte Γ, et que $\mathsf{InsInEnv}(\Gamma_0, \delta, k, \Gamma, \Gamma')$, alors pour éviter les captures de variable, M devra être relogé en $\uparrow_k^1 M$ lorsqu'il apparaît dans le contexte Γ'.

Informellement, il est peut-être plus simple de séparer la relocation des déclarations et l'insertion proprement dite. Ainsi, pour un contexte $\Gamma_0\Delta$, l'insertion au rang $k = |\Delta|$ de δ pourra se noter $\Gamma_0\delta(\Delta^+)$, la relocation d'une portion de contexte Δ^+ se définissant par récurrence sur la longueur de Δ :

$$[]^+ = [] \qquad (\Delta\delta)^+ = \Delta^+ \uparrow^1_{|\Delta|} \delta$$

Pour définir la substitution dans un contexte, il faut prendre garde lorsque l'on substitue une variable dont la déclaration est une définition. En effet, la déclaration $[M : T]$ signifie que la variable a déjà une valeur. Il serait incohérent d'autoriser à substituer cette variable par autre chose que M. On définit l'ensemble des valeurs par lesquelles une variable peut être substituée :

Définition 4.7 (prédicat `val_ok`) *Un axiome peut être substitué par n'importe quel terme, alors qu'une définition ne peut l'être que par la valeur qui lui est affectée dans le contexte.*

$$\mathsf{ValOk}(\delta) \stackrel{\text{def}}{\equiv} \begin{cases} \textsc{Term} & \textit{si } \delta = [T] \\ \{M\} & \textit{si } \delta = [M : T] \end{cases}$$

On dira que M est une valeur adaptée à δ.

Cette notion est non typée. Une version typée serait qu'on ne peut substituer un axiome que par un terme dont le type est T. Ceci sera établi par le lemme de substitution. Comme nous n'avons pas encore défini le typage, nous nous contenterons de cette version non typée.

Définition 4.8 (prédicat `sub_in_env`) *Comme pour la relocation, la substitution dans un contexte est définie formellement à l'aide de deux clauses :*

$$(\textsc{Sub-0}) \quad \frac{N \in \mathsf{ValOk}(\delta)}{\mathsf{SubInEnv}(\Gamma_0, \delta, N, 0, \Gamma_0\delta, \Gamma_0)}$$

$$(\textsc{Sub-S}) \quad \frac{\mathsf{SubInEnv}(\Gamma_0, \delta, N, k, \Gamma, \Gamma')}{\mathsf{SubInEnv}(\Gamma_0, \delta, N, k+1, \Gamma\gamma, \Gamma(\gamma\{k\backslash N\}))}$$

La proposition $\mathsf{SubInEnv}(\Gamma_0, \delta, N, k, \Gamma, \Gamma')$ signifie que Γ' est égal au contexte Γ dans lequel on a substitué la k-ième variable (dont la déclaration est δ) par N. Γ_0 est le contexte de δ.

Autrement dit, cela signifie que N est une valeur correcte pour la déclaration δ et qu'il existe une portion de contexte Δ de longueur k telle que $\Gamma = \Gamma_0\delta\Delta$ et $\Gamma' = \Gamma(\Delta\,\{N\})$, la définition de $\Delta\,\{N\}$ étant :

$$[]\,\{N\} \stackrel{\text{def}}{\equiv} [] \qquad (\Delta\delta)\,\{N\} \stackrel{\text{def}}{\equiv} \Delta\,\{N\}\,(\delta\{|\Delta|\backslash N\})$$

4.2.3 Règles de sous-typage abstraites

Dans les présentations habituelles, on définit ensuite le jugement de convertibilité ou de sous-typage à partir de la β-réduction. Dans cette présentation, on ne figera pas tout de suite la règle de sous-typage employée. On raisonnera sur une relation de sous-typage abstraite,

en introduisant un nouveau paramètre. Cela nous permettra de mettre en évidence et de mieux comprendre les propriétés importantes que doit vérifier ce nouveau paramètre.

Néanmoins, il est important d'avoir en tête ce par quoi on veut instancier ce paramètre pour être certain que l'on pourra bien le faire le moment voulu. Habituellement, la relation de conversion ou de sous-typage est binaire : $A \leq B$ indique que A est un sous-type de B. Si nous voulons pouvoir traiter le cas de δ-réduction (le remplacement d'une constante par sa définition), il est nécessaire que cette relation dépende aussi du contexte, là où sont rangées les définitions.

Une règle de sous-typage est une relation ternaire R dont le domaine est $\textsc{Ctx} \times \textsc{Term} \times \textsc{Term}$ et vérifiant quelques conditions que nous allons définir. L'expression $R(\Gamma, A, B)$ signifie que dans le contexte Γ, le type A est plus petit que B, au sens de R.

Comme il vient d'être dit, n'importe quelle relation n'est pas une règle de sous-typage. Les deux premières propriétés requises sont que le sous-typage doit être stable par relocation et par substitution. Intuitivement, si A est un sous-type de B, alors $\uparrow_k^n A$ doit aussi être un sous-type de $\uparrow_k^n B$, et $A\{k\backslash N\}$ un sous-type de $B\{k\backslash N\}$. La définition formelle est un peu plus complexe car il faut tenir compte des relocations et substitutions dans le contexte.

Définition 4.9 (prédicat `R_lift`) *Une relation R est stable par relocation si R est préservé par ajout de nouvelles déclarations dans le contexte :*

$$\mathsf{Rlift} \stackrel{\text{def}}{\equiv} \left\{ R \;\middle|\; \forall \Gamma, \Delta, \delta, A, B.\, R(\Gamma\Delta, A, B) \;\Rightarrow\; R(\Gamma\delta(\Delta^+), \uparrow^1_{|\Delta|} A, \uparrow^1_{|\Delta|} B) \right\}$$

Définition 4.10 (prédicat `R_rt`) *On note R^* la fermeture réflexive transitive de la relation R ; elle est définie par les règles suivantes :*

$$\frac{R(\Gamma, A, B)}{R^*(\Gamma, A, B)} \qquad \frac{}{R^*(\Gamma, A, A)} \qquad \frac{R^*(\Gamma, A, B) \qquad R^*(\Gamma, B, C)}{R^*(\Gamma, A, C)}$$

Définition 4.11 (prédicat `R_subst`) *Une relation R est dite stable par substitution si R est préservé par substitution par une valeur adaptée :*

$$\mathsf{Rsubstwk} \stackrel{\text{def}}{\equiv} \left\{ R \;\middle|\; \begin{array}{l} \forall \Gamma, \Delta, \delta, A, B.\, \forall M \in \mathsf{ValOk}(\delta). \\ \qquad R(\Gamma\delta\Delta, A, B) \;\Rightarrow\; R^*(\Gamma(\Delta\,\{M\}), A\{|\Delta|\backslash M\}, B\{|\Delta|\backslash M\}) \end{array} \right\}$$

Pour justifier l'utilisation de R^* dans cette définition, on peut prendre comme exemple la δ-réduction. Dans un contexte où la variable 0 est définie avec la valeur M, on a la réduction suivante :

$$[M : T] \vdash \natural 0 \to_\delta \uparrow^1 M$$

Mais si l'on substitue la variable 0 par M, la réduction ne se fait plus en un pas, mais en zéro :

$$[\,] \vdash M \to_\delta M$$

En revanche, on verra que δ vérifie bien la condition Rsubstwk.

Définition 4.12 (prédicats `red_decl`, `R_in_env`) *Un contexte Γ est plus petit que Γ', ce qui sera noté $(\Gamma)\ R\ (\Gamma')$, si l'un des types de Γ est un sous-type de son correspondant dans Γ'.*

$$\frac{R(\Gamma, A, B)}{(\Gamma\,[A])\ R\ (\Gamma\,[B])} \qquad \frac{R(\Gamma, A, B)}{(\Gamma\,[M : A])\ R\ (\Gamma\,[M : B])} \qquad \frac{(\Gamma)\ R\ (\Gamma')}{(\Gamma\delta)\ R\ (\Gamma'\delta)}$$

La propriété $(\Gamma)\ R\ (\Gamma')$ signifie que pour un certain rang n, on a $R(\Gamma_0, \Gamma(n).\mathrm{T}, \Gamma'(n).\mathrm{T})$, Γ_0 étant le contexte de la n-ième déclaration (i.e. le plus grand préfixe de Γ ne contenant pas $\Gamma(n)$). Les autres déclarations et les valeurs associées aux variables sont inchangées. Si R est une relation de sous-typage, cela signifie que Γ est égal à Γ' dans lequel un des types de variables est plus petit au sens de R.

La troisième propriété des règles de sous-typage est que le sous-typage ne peut pas dépendre des types présents dans le contexte, mais uniquement des valeurs. On essaiera de raffiner la définition de cette propriété dans le prochain chapitre, ce qui devrait permettre de profiter de l'information qu'est le type d'une variable pour autoriser certaines réductions. Comme nous le verrons, si ce raffinement ne pose pas de problème particulier pour la métathéorie élémentaire, il en va autrement quand il s'agit de prouver la décidabilité des jugements.

Définition 4.13 (prédicat `R_stable_env`)

$$\mathsf{Rstable} \stackrel{\text{def}}{\equiv} \{R \mid \forall R', \Gamma, \Gamma', A, B.\ R(\Gamma, A, B)\ \wedge\ (\Gamma')\ R'\ (\Gamma)\ \Rightarrow\ R(\Gamma', A, B)\}$$

La définition ci-dessus est imprédicative puisqu'elle utilise une quantification sur toutes les relations R'. Celle-ci est en fait inutile puisque $(\Gamma')\ R'\ (\Gamma)$ est monotone par rapport à R'. On obtiendrait une définition équivalente en considérant uniquement la relation toujours vraie pour R'.

Certains lemmes porteront sur des relations qui ne vérifient que certaines de ces propriétés, mais les théorèmes importants ne feront pas autant de détails et porteront sur des relations vérifiant ces trois propriétés.

Définition 4.14 (type `Basic_rule`) *Une relation ternaire R est* invariante *si elle vérifie les trois propriétés ci-dessus.*

$$\mathcal{BR} \stackrel{\text{def}}{\equiv} \mathsf{Rlift} \cap \mathsf{Rsubstwk} \cap \mathsf{Rstable}$$

Il est maintenant possible de définir ce qu'est une relation de sous-typage :

Définition 4.15 (type `Subtyping_rule`) *Une relation invariante R est une règle de sous-typage si pour tout contexte Γ, R partiellement appliquée à Γ est un préordre (i.e. une relation réflexive et transitive).*

$$\textsc{SubRule} \stackrel{\text{def}}{\equiv} \{R \in \mathcal{BR} \mid R^* \subseteq R\}$$

Nous verrons que ces conditions sont suffisantes pour pouvoir prouver un certain nombre de propriétés métathéoriques, comme les lemmes d'affaiblissement, de substitution et la correction des types. On remarquera tout d'abord une propriété des règles de sous-typage :

Lemme 4.16 (`sub_sort_env_indep`) *Soit R une règle de sous-typage. La restriction de R aux sortes est indépendante du contexte :*

$$R(\Gamma, s_1, s_2) \Rightarrow R(\Gamma', s_1, s_2)$$

Preuve Si $R(\Gamma, s_1, s_2)$, il suffit d'appliquer la propriété Rsubstwk pour chaque variable de Γ, d'où $R^*([], s_1, s_2)$. R étant transitive, on a aussi $R([], s_1, s_2)$. Il ne reste plus qu'à appliquer Rlift pour déduire $R(\Gamma', s_1, s_2)$ pour n'importe quel contexte Γ'. ■

Dans ce cas, on se permettra de ne pas mentionner le contexte.

4.2.4 Jugements de typage

Comme pour les PTS, nous regroupons les paramètres de notre classe de systèmes de types dans une structure. Un PTS avec sous-typage aura les mêmes paramètres que les PTS, plus une règle de sous-typage.

Définition 4.17 (type `PTS_sub_spec`) *La spécification d'un PTS avec sous-typage est un triplet :*

$$\begin{array}{lll} \langle\ \textsc{Axiom}: & \mathcal{P}(\textsc{Sort}\times\textsc{Sort}); & \textit{(* Typage des sortes *)} \\ \quad \textsc{Rule}: & \mathcal{P}(\textsc{Sort}\times\textsc{Sort}\times\textsc{Sort}); & \textit{(* Formation des produits *)} \\ \quad \leq: & \textsc{SubRule} & \textit{(* Règle de sous-typage *)} \\ \rangle & & \end{array}$$

L'ensemble des spécifications de PTS avec sous-typage est noté $\mathcal{PTS}$.

La déclaration correspondante en Coq utilise les enregistrements, qui sont équivalents à des types inductifs avec un seul constructeur :

```
Record PTS_sub_spec: Type := {
  pts_axiom: (relation sort);
  pts_rules: sort->sort->sort->Prop;
  pts_le_type: Subtyping_rule
}.
```

En plus du type `PTS_sub_spec`, cette commande définit `Build_PTS_sub_spec`, qui construit une structure à partir de ses trois composantes, ainsi que les trois projections `pts_axiom`, `pts_rules` et `pts_le_type` qui extraient l'une des trois composantes.

Le premier paramètre des PTS, l'ensemble des sortes, à déjà été introduit. Ce paramètre n'est pas inclus dans cette structure car nous avons dû l'introduire avant de pouvoir définir les termes. Comme la définition d'une relation de sous-typage fait appel à celle des termes, il faudrait soit définir les termes en même temps que la structure, soit définir les termes de manière paramétrée par l'ensemble des sortes. La première solution serait très lourde car en fait, toutes les définitions vues précédemment se trouveraient mutuellement dépendantes. La deuxième solution semble aisée, mais formellement, il serait assez pénible de devoir rappeler le paramètre à chaque nœud des termes.

Nous rappellons rapidement ce qui a déjà été dit section 2.2.3 : Axiom est une relation qui indique quels sont les types des sortes (par convention, les types des sortes ne peuvent être que des sortes). Si $(s_1, s_2) \in \textsc{Axiom}$, alors s_1 a pour type s_2. Rule indique comment se forment les produits. Si $(s_1, s_2, s_3) \in \textsc{Rule}$ et les termes T et U ont pour type respectivement s_1 et s_2, alors le type produit $\Pi\, T.\, U$ a le type s_3.

Dans tout le reste de ce chapitre, nous travaillerons sur un PTS $\langle \textsc{Axiom}; \textsc{Rule}; \leq\rangle \in \mathcal{PTS}$. On notera $\Gamma \vdash A \leq B$ au lieu de $\leq(\Gamma, A, B)$. Il faut prendre garde au fait que la relation de sous-typage ne dépend pas du typage, contrairement à certaines présentations de systèmes de types. Le jugement $\Gamma \vdash A \leq B$ ne sous-entend pas que A et B soient bien formés. Bien entendu, un tel jugement n'aura de sens que lorsque A et B sont bien typés.

Définition 4.18 (prédicats `wf`, `typ`) *Les jugements de typage des Systèmes de Types Purs avec sous-typage sont définis par les règles des figures 4.2 et 4.3.*

Le jugment $\Gamma \vdash$ se lit "le contexte Γ est bien formé", ce qui signifie qu'il associe à chaque variable une déclaration bien typée, c'est-à-dire dont le type est lui-même typé par une sorte. Le jugement $\Gamma \vdash M : T$ signifie que dans le contexte Γ, le terme M a le type T, ou de manière équivalente : sous les hypothèses Γ, M est une preuve de la proposition T.

$$\text{(WF-[ ])}\ \frac{}{[\,]\vdash} \qquad \text{(WF-VAR)}\ \frac{\Gamma\vdash T:\ s}{\Gamma\,[T]\vdash}(s\in \textsc{Sort})$$

$$\text{(WF-DEF)}\ \frac{\Gamma\vdash M:\ T \qquad \Gamma\vdash T:\ s}{\Gamma\,[M:T]\vdash}(s\in \textsc{Sort})$$

FIG. 4.2: Règles de typage des contextes

$$\text{(SRT)}\ \frac{\Gamma\vdash \qquad (s_1,s_2)\in \textsc{Axiom}}{\Gamma\vdash s_1:\ s_2} \qquad \text{(REL)}\ \frac{\Gamma\vdash \qquad \Gamma(n)=\delta}{\Gamma\vdash \natural n:\uparrow^{n+1}\delta.\mathrm{T}}$$

$$\text{(PROD)}\ \frac{\Gamma\vdash T:\ s_1 \qquad \Gamma\,[T]\vdash U:\ s_2}{\Gamma\vdash \Pi\,T.\,U:\ s_3}\ (s_1,s_2,s_3\in \textsc{Rule})$$

$$\text{(LAM)}\ \frac{\Gamma\,[T]\vdash M:\ U \qquad \Gamma\vdash \Pi\,T.\,U:\ s}{\Gamma\vdash \lambda T.\,M:\ \Pi\,T.\,U}(s\in \textsc{Sort})$$

$$\text{(APP)}\ \frac{\Gamma\vdash M:\ \Pi\,T.\,U \qquad \Gamma\vdash N:\ T}{\Gamma\vdash (M\ N):\ U\{0\backslash N\}}$$

$$\text{(CONV)}\ \frac{\Gamma\vdash M:\ U \qquad \Gamma\vdash V:\ s \qquad \Gamma\vdash U\leq V}{\Gamma\vdash M:\ V}\ (s\in \textsc{Sort})$$

$$\text{(CONV-SRT)}\ \frac{\Gamma\vdash M:\ U \qquad \Gamma\vdash U\leq s}{\Gamma\vdash M:\ s}\ (s\in \textsc{Sort})$$

FIG. 4.3: Règles de typage des PTS

Avant de commenter la règle de conversion, nous introduisons la notion de type bien formé. Elle diffère de celle de terme typé par une sorte car il peut y avoir des sortes n'ayant pas de type, et nous considérons tout de même que n'importe quelle sorte est un type bien formé, puisqu'elle peut être habitée.

Définition 4.19 (prédicat `wf_type`) *Un type bien formé est un terme qui est soit une sorte, soit typé par une sorte. L'ensemble des types bien formés dans le contexte* Γ *est noté* $\mathcal{WT}_\Gamma$.

$$\mathcal{WT}_\Gamma \stackrel{\text{def}}{\equiv} \textsc{Sort} \cup \{T \mid \exists s \in \textsc{Sort}.\, (\Gamma \vdash T : s)\}$$

Nous verrons que les types bien formés sont les termes susceptibles d'apparaître à droite dans les jugements de typage. Ce résultat s'appelle lemme de correction des types.

La règle de conversion des PTS de notre formalisme se démarque des présentations habituelles sur plusieurs points. Fondamentalement, c'est désormais une règle de sous-typage, puisque la relation $\leq$ n'est pas nécessairement symétrique. Par rapport à certaines présentations, on ne requiert pas que les types U et V aient le même type (la même sorte puisque ce sont des types), car cela n'est pas vrai lorsque l'on considère la cumulativité. Enfin, nous ne demandons pas que U soit un type bien formé, car cela est assuré à partir du moment où l'on peut prouver le lemme de correction des types évoqué ci-dessus.

La dernière différence est qu'en fait il y a deux règles de conversions : l'une lorsque V est typé par une sorte, et l'autre lorsque V est une sorte. Nous aurions préféré pouvoir définir directement la règle de conversion à l'aide de la règle dérivée ci-dessous, mais Coq aurait engendré un principe de récurrence peu pratique à utiliser puisqu'il faudrait à chaque fois donner trois propriétés à prouver simultanément, alors que celle concernant $\mathcal{WT}$ se dérive de celle concernant le jugement de typage.

Lemme 4.20 (`typ_conv_wf`) *La règle suivante est dérivable :*

$$(\textsc{Conv-Wf})\ \frac{\Gamma \vdash M : U \qquad \Gamma \vdash U \leq V \qquad V \in \mathcal{WT}_\Gamma}{\Gamma \vdash M : V}$$

Le résultat qui suit montre que les produits que l'on peut effectivement former (au sens où si A a le type s_1 et si B a le type s_2, alors $\Pi\, A.\, B$ a le type s_3) peut être plus grand que ce que contient Rule, du fait du sous-typage.

Définition 4.21 (prédicat `rule_sat`) *Les règles de formation des produits complétées par le sous-typage se définit ainsi :*

$$\textsc{Rule}^* \stackrel{\text{def}}{\equiv} \{(s_1, s_2, s_3) \mid \exists s_1', s_2'.\, s_1 \leq s_1' \ \wedge\ s_2 \leq s_2' \ \wedge\ (s_1', s_2', s_3) \in \textsc{Rule}\}$$

Évidemment, $\textsc{Rule} \subseteq \textsc{Rule}^*$. On peut généraliser la règle de typage du produit de la manière suivante :

$$\frac{\Gamma \vdash T : s_1 \qquad \Gamma\,[T] \vdash U : s_2}{\Gamma \vdash \Pi\, T.\, U : s_3}\ (s_1, s_2, s_3 \in \textsc{Rule}^*)$$

Preuve Il suffit d'appliquer la règle Conv-Srt sur chacune des prémisses avant d'appliquer la règle du produit. ■

Cette propriété aurait été plus complexe si on n'avait pas la règle de conversion vers une sorte.

4.3 Métathéorie des PTS avec sous-typage

Le système étant maintenant complètement défini, nous nous fixons comme but de prouver la décidabilité des jugements de typage définis précédemment. Comme tous les PTS n'ont pas un typage décidable, il va falloir imposer des conditions sur les paramètres pour atteindre notre but. La méthode présentée au début de ce chapitre va nous permettre de dégager des ensembles de conditions suffisantes pour construire un algoritme de typage générique. Ces conditions porteront presque exclusivement sur les paramètres du PTS. Par exemple, nous supposerons que la relation de sous-typage est décidable.

Dans la section suivante, nous ferons quelques hypothèses encore plus restrictives sur le sous-typage, définissant ainsi la classe des PTS avec cumulativité, et nous raffinerons les conditions de cette section afin d'établir des hypothèses plus simples à satisfaire.

4.3.1 Interface du noyau

On a vu dans la formalisation de l'interface (section 2.4) que ce qui nous intéressait n'était pas exactement la décidabilité du typage (ce résultat est cependant une façon simple de résumer notre objectif), mais plutôt la correction des fonctions de typage (inférence ou vérification) dans un contexte bien formé.

Concrètement, nous souhaitons prouver les fonctions correspondant aux commandes de notre système de preuve. On définit donc une structure regroupant des spécifications de programmes pour chacune de nos commandes. En programmation fonctionnelle, on dirait que l'on déclare un type de signature de module. Cette signature forme l'interface du noyau. Tout le travail consistera à instancier cette signature.

Définition 4.22 (type `PTS_TC`) *L'interface du noyau est un type de structure regroupant des algorithmes dont les spécifications sont :*

$$
\begin{array}{llll}
\langle \ \textit{(* Nom *)} & & \textit{(* Précondition *)} & \textit{(* Résultat *)} \\
\quad \texttt{inf\_ppal}: & \forall \Gamma, M. & \Gamma \vdash & \Rightarrow \ \mathsf{InfPpal}(T \mid \Gamma \vdash M : T, \leq); \\
\quad \texttt{chk\_typ}: & \forall \Gamma, M, T. & \Gamma \vdash & \Rightarrow \ \mathsf{Dec}\,(\Gamma \vdash M : T)\,; \\
\quad \texttt{add\_typ}: & \forall \Gamma, T. & \Gamma \vdash & \Rightarrow \ \mathsf{Dec}\,(\Gamma\,[T] \vdash)\,; \\
\quad \texttt{add\_def}: & \forall \Gamma, M, T. & \Gamma \vdash & \Rightarrow \ \mathsf{Dec}\,(\Gamma\,[M{:}T] \vdash)\,; \\
\quad \texttt{chk\_wk}: & \forall \Gamma, T. & \Gamma \vdash & \Rightarrow \ \mathsf{Dec}\,(T \in \mathcal{WT}_\Gamma)\,; \\
\quad \texttt{chk\_wft}: & \forall \Gamma, M, T. & \Gamma \vdash \ \wedge \ T \in \mathcal{WT}_\Gamma & \Rightarrow \ \mathsf{Dec}\,(\Gamma \vdash M : T) \\
\rangle &&&
\end{array}
$$

Cette définition peut se voir soit comme la définition d'une signature de module appelée PtsTc*, soit comme le sous-ensemble des spécification de PTS pour lesquels une telle structure existe. La première vision est plutôt informatique, alors que la deuxième correspond à l'usage en mathématiques.*

L'opération dont on a réellement besoin est la vérification de types (`chk_typ`). Mais à cause de l'application, il faut aussi être capable d'inférer le type d'une expression : pour vérifier que $(M\ N)$ a le type T, il faut connaitre quel est le type de l'argument N. Rien ne nous permet de deviner ce type, il faut donc être capable d'inférer le type d'un terme.

Puisque la relation $\leq$ n'est pas symétrique, nous n'avons pas la propriété d'unicité du typage que nous avons dans le Calcul des Constructions : si on a $\Gamma \vdash M : T$ et $\Gamma \vdash M : T'$,

alors T et T' sont convertibles. Ce genre de résultat est utile pour montrer la complétude de l'algorithme de typage, car il restreint l'ensemble des types que peut avoir un terme. Par exemple, le simple fait qu'un terme M soit de type $\Pi\, A.\, B$, et N de type T, celui-ci n'étant pas un sous-type de A ne permet pas de conclure que l'application $(M\ N)$ est mal typée : il se pourrait que N ait aussi un type qui soit un sous-type de A. En revanche, dans un système ayant la propriété d'unicité du typage, la non-convertibilité de T et A entraine la non-typabilité de l'application.

Pour résoudre ce problème, il y a plusieurs possibilités. La plus simple est de faire en sorte que le système admette des types principaux. C'est-à-dire que parmi tous les types d'un terme, il en existe un qui permette de retrouver tous les autres. Dans notre cas, il s'agit de trouver le type le plus petit au sens de $\leq$, car la règle de sous-typage permettra de construire la dérivation pour n'importe quel type. Le système de types de `Coq` possède cette propriété, mais tous les PTS ne l'ont pas. L'autre solution, plus générale, consiste à avoir des schémas de types avec des variables, et considérer des contraintes sur ces variables de types. Ainsi l'ensemble des types d'un terme n'est pas représenté par un type du formalisme, mais par l'ensemble des instances d'un schéma vérifiant certaines contraintes. Ces problèmes ont étés traités pour les PTS, soit afin de gérer automatiquement la numérotation des univers dans [30], soit pour pouvoir décider le typage d'une plus grande classe de PTS ([59]). Cela complique considérablement la tâche, et aucune de ces deux propositions n'ont été vérifiées formellement à ce jour.

Notre choix de ne traiter que le cas où il existe un type principal nous permet d'utiliser un algorithme qui est une généralisation simple de celui employé dans le *Constructive Engine* de Huet [31]. La fonction centrale de notre noyau sera une fonction permettant de calculer le type principal d'un terme, ce que fait `inf_ppal`. Cette spécification correspond clairement à la commande `Infer` de l'interface (section 2.4), puisqu'une preuve de cette spécification s'extrairait vers une fonction qui calcule le plus petit type de M au sens du sous-typage. De même, `chk_typ` correspond à la commande `Check`. Les commandes d'extension du contexte `Axiom` et `Definition` sont implantées par les champs `add_typ` et `add_def`. Le programme `inf_ppal` est de loin le plus complexe car il contient l'algorithme d'inférence de type. Les autres spécifications ne sont que des corollaires assez simples montrés dans la section 4.3.3.

Avant de prouver ces résultats, nous commencerons par prouver des résultats métathéoriques simples.

4.3.2 Résultats métathéoriques

Comme il a déjà été dit dans l'introduction, les résultats métathéoriques des PTS ont été étudiés dans la littérature ([4, 22], entre autres), et parfois formellement, comme dans [56].

La présentation de la formalisation de cette section ne pose pas de problèmes particuliers car les raisonnements employés (principalement des récurrences structurelles) sont très proches des preuves informelles, si ce n'est que l'on doit aussi traiter les cas simples ou répétitifs, par manque d'automatisation.

Lemmes d'inversion

Les lemmes d'inversion peuvent se voir comme des conditions nécessaires pour qu'un jugement ait une chance d'être dérivable. Les conditions nécessaires servent à deux occasions : d'une part pour montrer que le terme est mal typé lorsque l'algorithme échoue (par contraposée). D'autre part pour montrer que l'on rend bien le type principal.

Calculatoirement, les lemmes d'inversion se comprennent comme des fonctions qui vont rechercher certaines sous-dérivations importantes d'une dérivation donnée en argument. Ce-

pendant, comme les dérivations ont été déclarées dans la sorte `Prop` de Coq, le contenu calculatoire de ces fonctions sera effacé au cours de l'extraction vers Objective Caml.

Lemme 4.23 (`typ_wf`) *Le contexte de n'importe quel jugement dérivable est bien formé ; la règle suivante est admissible :*

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash}$$

Preuve Immédiate, par récurrence sur le jugement de typage. ■

Par règle admissible, on entend que si les jugements en premisse sont dérivables, alors il existe une dérivation du jugement en conclusion. Elle ne doit pas être confondue avec une règle *dérivable*, qui signifie que l'on peut construire une dérivation de la conclusion en utilisant les dérivations des prémisses. La différence est que dans le cas des règles dérivées, les dérivations des prémisses sont des sous-dérivations de celle de la conclusion, et il est donc possible, dans un raisonnement par récurrence, d'utiliser l'hypothèse de récurrence sur les prémisses d'une règle dérivée.

Prouver que la dérivation de $\Gamma \vdash$ est une sous-dérivation de celle de $\Gamma \vdash M : T$ pourrait servir à faire des raisonnements par récurrence plus facilement qu'avec le simple schéma de récurrence primitif. Comme nous n'aurons pas besoin de ce genre de récurrence, on ne prendra pas la peine de prouver ce fait.

Lemme 4.24 (`wf_sort`) *Les types apparaissant dans un contexte bien formé sont typables par une sorte :*

$$\Gamma\delta\Delta \vdash \Rightarrow \exists s \in \textsc{Sort}.\, \Gamma \vdash \delta_{\mathrm{T}} : s$$

En observant les règles de typage, on remarque que mis à part les deux règles de conversion, il y a exactement une règle par constructeur de terme. Ainsi, pour chaque constructeur de terme, nous avons une caractérisation simple des dérivations dont le terme-preuve est bâti sur ce constructeur : la règle correspondant à ce dernier a été utilisée en dernier, éventuellement suivie de l'application un nombre arbitraire de fois des règles de conversion.

Lemme 4.25 (`inversion_lemma`) *Les énoncés des lemmes d'inversion sont regroupés figure 4.4.*

Preuve Ces lemmes sont prouvés simultanément par récurrence structurelle sur la dérivation. Il n'y a pas de difficulté notable. ■

Ces lemmes d'inversion vont rechercher les sous-dérivations de la règle correspondant au constructeur de tête. L'un des corollaires est donc qu'un terme typé a des sous-termes bien typés. Par contraposition, on en déduit que si l'un des sous-termes est mal typé, le terme entier est mal typé.

Lemme d'affaiblissement

Le lemme d'affaiblissement garantit que les jugements de typage sont préservés par ajout de nouvelles variables dans le contexte (du point de vue logique, une proposition reste vraie si l'on fait des hypothèses supplémentaires). Il faut juste tenir compte des relocations des indices de de Bruijn, et s'assurer que le nouveau contexte est bien formé.

Lemme 4.26 (`thinning_n`) *Le lemme d'affaiblissement peut s'énoncer sous forme de règle d'inférence admissible :*

$$\frac{\Gamma \vdash M : T \qquad \Gamma\Delta \vdash}{\Gamma\Delta \vdash \uparrow^{|\Delta|} M : \uparrow^{|\Delta|} T}$$

$$
\begin{array}{rcl}
\Gamma \vdash s_1 : T & \Rightarrow & \exists s_2 \in \textsc{Sort}. \begin{cases} (s_1, s_2) \in \textsc{Axiom} \\ \Gamma \vdash s_2 \leq T \end{cases} \\
\Gamma \vdash \natural n : T & \Rightarrow & \exists \delta \in \textsc{Decl}. \begin{cases} \Gamma(n) = \delta \\ \Gamma \vdash \uparrow^{n+1} \delta_{\mathrm{T}} \leq T \end{cases} \\
\Gamma \vdash \lambda A.\, M : T & \Rightarrow & \exists B \in \textsc{Term}. \exists s \in \textsc{Sort}. \begin{cases} \Gamma\,[A] \vdash M : B \\ \Gamma \vdash \Pi\, A.\, B : s \\ \Gamma \vdash \Pi\, A.\, B \leq T \end{cases} \\
\Gamma \vdash (M\ N) : T & \Rightarrow & \exists A, B \in \textsc{Term}. \begin{cases} \Gamma \vdash M : \Pi\, A.\, B \\ \Gamma \vdash N : A \\ \Gamma \vdash B\{0 \backslash N\} \leq T \end{cases} \\
\Gamma \vdash \Pi\, A.\, B : T & \Rightarrow & \exists (s_1, s_2, s_3) \in \textsc{Rule}. \begin{cases} \Gamma \vdash A : s_1 \\ \Gamma\,[A] \vdash B : s_2 \\ \Gamma \vdash s_3 \leq T \end{cases}
\end{array}
$$

FIG. 4.4: Lemmes d'inversion du jugement de typage

On peut prouver ce résultat par récurrence sur la longueur de Δ. Il suffit de le prouver dans le cas où l'on n'ajoute qu'une seule variable. Mais il faut renforcer ce résultat avec le cas où l'insertion ne se fait pas à la fin du contexte. Cela est nécessaire pour traiter le cas de l'abstraction et du produit.

Lemme 4.27 (`typ_weak, thinning`) *La règle*

$$\frac{\Gamma\Delta \vdash M : T \qquad \Gamma\delta(\Delta^+) \vdash}{\Gamma\delta(\Delta^+) \vdash \uparrow^1_{|\Delta|} M : \uparrow^1_{|\Delta|} T}$$

est admissible.

Preuve Par récurrence sur la dérivation de M. Il n'est pas la peine de faire une récurrence mutuelle avec le jugement de bonne formation des contextes car nous avons fait l'hypothèse $\Gamma\delta(\Delta^+) \vdash$ au lieu de $\Gamma\delta \vdash$. Le cas des règles de conversion utilise le fait que $\leq \in \mathsf{Rlift}$. ■

Lemme de substitution

Le typage est préservé par substitution d'une variable par un terme dont le type est celui déclaré pour la variable.

$$\frac{\Gamma\,[T] \vdash M : U \qquad \Gamma \vdash N : T}{\Gamma \vdash M\{0 \backslash N\} : U\{0 \backslash N\}}$$

Ici encore, il faut renforcer l'hypothèse de récurrence :

Lemme 4.28 (`typ_sub, substitution`) *La règle suivante est admissible :*

$$\frac{\Gamma\delta\Delta \vdash M : U \qquad \Gamma \vdash N : \delta_{\mathrm{T}} \qquad N \in \mathsf{ValOk}(\delta)}{\Gamma(\Delta\,\{N\}) \vdash M\{|\Delta| \backslash N\} : U\{|\Delta| \backslash N\}}$$

Preuve La preuve suit le même schéma que la preuve du lemme d'affaiblissement. Le cas des règles de conversion utilise $\leq\in$ Rsubstwk. ■

Sous-typage dans le contexte

On prouve un résultat de contravariance du contexte : un jugement reste dérivable si l'on diminue les types dans le contexte (dans la mesure où le contexte reste valide). On aura besoin de ce résultat pour montrer l'auto-réduction de β, ou pour montrer que la correction d'une règle de réduction passe au contexte.

Lemme 4.29 (`subtype_in_env`) *La règle suivante est admissible :*

$$\frac{\Gamma \vdash M : T \qquad (\Gamma') \leq (\Gamma) \qquad \Gamma' \vdash}{\Gamma' \vdash M : T}$$

Preuve On peut voir une preuve de ce théorème comme une fonction qui transforme une dérivation de $\Gamma \vdash M : T$ en une dérivation de $\Gamma' \vdash M : T$. La dérivation a la même forme sauf lorsque l'on arrive à la variable dont le type est différent entre Γ et Γ'. Comme on sait que le type de cette variable dans Γ' est plus grand que celui dans Γ, il suffit de rajouter l'application de la règle de conversion. Le cas des deux règles de conversion fait appel à $\leq\in$ Rstable. ■

Ce dernier résultat sert à plusieurs endroits : d'une part pour prouver que la propriété d'auto-réduction est stable par fermeture congruente d'une règle de réduction, et d'autre part pour inférer le type principal de l'abstraction.

Correction des types

Le lemme de correction des types exprime que le terme apparaissant à droite dans un jugement dérivable est lui-même bien typé.

Lemme 4.30 (`type_correctness`) *Seuls les types bien formés et les sortes sont habités : pour tous* Γ, M, T,

$$\Gamma \vdash M : T \;\Rightarrow\; T \in \mathcal{WT}_\Gamma$$

Preuve Par récurrence sur la dérivation. Seuls les cas des variables et de l'application ne sont pas triviaux.

variables : On a $\Gamma_0 \delta \Delta \vdash$ avec $|\Delta| = n$. D'après `inv_wf_sort`, il existe s tel que $\Gamma_0 \vdash \delta.\mathrm{T} : s$. Le lemme d'affaiblissement permet de conclure $\Gamma_0 \delta \Delta \vdash \uparrow^{n+1} \delta.\mathrm{T} : s$.

application : En utilisant l'hypothèse de récurrence sur $\Gamma \vdash M : \Pi T.U$ (notations de la figure 4.3) et en appliquant le lemme d'inversion, on en déduit qu'il existe une sorte s_2 telle que $\Gamma\,[T] \vdash U : s_2$. Comme $\Gamma \vdash N : T$, le lemme de substitution permet de conclure ce cas.

■

D'un point de vue logique, cela nous assure que seules les propositions bien formées seront prouvables. Mais cela a une importance pour la décidabilité du typage. Lors du typage d'une application, il faut vérifier que le type de l'argument est compatible avec le type de l'argument formel de la fonction. Pour pouvoir faire ce test, il faut que ces deux types soient bien formés. Si l'on n'avait pas le lemme de correction, il faudrait retyper le type de l'argument. Cela est d'une part inefficace, et d'autre part, spécifié ainsi, cela peut faire boucler l'algorithme de typage.

$\langle$ (* Nom *) (* Spécification *)

`least_sort:` $\forall\Gamma.\,\forall M \in \mathcal{WT}_\Gamma.\,\mathsf{InfPpal}(s \in \textsc{Sort} \mid \Gamma \vdash M \leq s, \leq)$;

`least_prod:` $\forall\Gamma.\,\forall M \in \mathcal{WT}_\Gamma.$
$\mathsf{InfPpal}(A, B \mid \Gamma \vdash M \leq \Pi\, A.\, B \ \wedge\ \Pi\, A.\, B \in \mathcal{WT}_\Gamma,$
$((A, B), (A', B')) \mapsto \Gamma \vdash \Pi\,\mathrm{A}.\,\mathrm{B} \leq \Pi\,\mathrm{A'}.\,\mathrm{B'})$;

`le_type_dec:` $\forall\Gamma.\,\forall A, B \in \mathcal{WT}_\Gamma.\,\mathsf{Dec}\,(\Gamma \vdash A \leq B)$;

`infer_axiom:` $\forall s_1.\,\mathsf{InfPpal}(s_2 \mid (s_1, s_2) \in \textsc{Axiom}, \leq)$;

`infer_rule:` $\forall s_1, s_2.\,\exists^{\mathsf{Ppal}}(s_3 \mid (s_1, s_2, s_3) \in \textsc{Rule}^*, \leq)$;

`topsort:` $\forall\Gamma, s_1, s, T.\,\Gamma \vdash s_1 \leq T \ \wedge\ \Gamma \vdash T : s \ \Rightarrow\ \exists s_2.\,(s_1, s_2) \in \textsc{Axiom}$;

`le_type_prod:` $\forall\Gamma, T, A, B.\,\Gamma\,[T] \vdash A \leq B \ \Rightarrow\ \Gamma \vdash \Pi\, T.\, A \leq \Pi\, T.\, B$;

`inv_prod:` $\forall\Gamma, A, A', B, B'.\,\Gamma \vdash \Pi\, A.\, B \leq \Pi\, A'.\, B'$
$\Rightarrow\ \Gamma \vdash A' \leq A \ \wedge\ \Gamma\,[A'] \vdash B \leq B'$

$\rangle$

FIG. 4.5: Conditions assurant la décidabilité du typage (PTSALGOS)

Exemple 4.31 *Supposons que nous voulions inférer le type de $(M\ N)$, et que nous ayons inféré par appels récursifs les jugements $\Gamma \vdash M : \Pi\, A.\, B$ et $\Gamma \vdash N : A'$. Tout ce que nous savons de A' est que c'est un type de N. La spécification n'interdit pas que le type inféré de N soit $(\lambda B\{0\backslash N\}.\ \uparrow^1 A'\ (M\ N))$ au lieu de A'. Si nous devions retyper le type de N pour pouvoir le comparer avec A, nous aurions à retyper $(M\ N)$, ce qui ferait boucler l'algorithme de typage.*

Généralement, l'étude métathéorique des systèmes de types se poursuit par la preuve de résultats comme la correction de la β-réduction (ou d'auto-réduction) et la normalisation forte. Cela n'aurait pas de sens ici, car nous n'avons pas encore défini ce qu'est la β-réduction. D'une manière générale, nous n'avons pas considéré la notion de réduction, opération entre termes, mais uniquement celle de sous-typage qui relie des types. En pratique, la relation de sous-typage est souvent construite à partir d'une ou plusieurs règles de réduction, ce que nous verrons section 4.4.

Cela ne fera évidemment pas disparaître le problème. Cela signifie que les hypothèses que nous allons faire nécessiteront d'avoir préalablement prouvé les résultats d'auto-réduction et de normalisation forte.

4.3.3 Foncteur de construction du noyau

Avec les résultats métathéoriques la section précédente, nous pouvons construire un algorithme de typage, modulo un ensemble de conditions que nous allons définir.

Définition 4.32 (type `PTS_algos`) *La structure regroupant les algorithmes et les propriétés est décrite figure 4.5. Comme pour* PTSTC*, ce type de structures (noté* PTSALGOS*) peut être vu soit comme une interface de module, soit comme un sous-ensemble de $\mathcal{PTS}$.*

Cette définition appelle de nombreux commentaires :

- les algorithmes `least_sort` et `least_prod` servent à faire du filtrage sur les types, afin de reconnaître à quelle classe appartient un type. Lorsque l'on veut typer le destructeur du produit (i.e. l'application), il faut pouvoir vérifier que le type de la fonction est bien un produit.
- Noter que `least_prod` demande de prouver que le produit retourné est bien typé. C'est ici que se cache le résultat d'auto-réduction. Le résultat de normalisation forte permet de remplacer les préconditions de bonne formation des types par la condition de normalisation forte.
- La décidabilité du sous-typage (`le_type_dec`) est une condition nécessaire, car le terme $\lambda A.\,(\lambda \uparrow^1 B.\,\natural 0\ \natural 0)$ n'est bien typé que si $A \leq B$. Cette condition est en principe la plus complexe à prouver, car elle spécifie un algorithme en géneral assez évolué.
- Pour simplifier, on se place dans le cas des PTS *full*, c'est-à-dire que selon `infer_rule`, pour toute paire de types, on peut former le type produit. L'algorithme employé ici s'adapterait très facilement au cas des PTS *semi-full* de Pollack (voir [56, 59]). Dans le développement du chapitre suivant, on verra une condition encore plus générale, mais plus complexe à exprimer. Cependant, il est très facile de prouver qu'elle généralise la condition *semi-full*.
- `topsort` permet de s'assurer que les hiérarchies de sortes définies par Axiom et $\leq$ vont dans le même sens, i.e. une sorte non typée ne peut pas être un sous-type d'un type bien formé. Du strict point de vue typage, il suffirait d'un résultat plus simple, à savoir qu'il est décidable de savoir si une sorte non typée admet un sur-type typable par une sorte. Cependant, pour que notre système soit cohérent, il faut une certaine correlation entre les hiérarchies de sortes que définissent Axiom, Rule et $\leq$.
- `le_type_prod` n'est pas exactement la réciproque de `inv_prod` : le produit doit être covariant à droite, mais il n'est pas nécessaire qu'il soit contravariant à gauche. De son côté `inv_prod` autorise la contravariance.

À partir de ces algorithmes, il est assez facile de construire un algorithme de typage générique.

Théorème 3 (`full_type_checker`) *Tout PTS vérifiant les conditions de la définition précédente admet des jugements de typage décidables :*

$$\textsc{PtsAlgos} \subseteq \textsc{PtsTc}$$

Cette inclusion s'interprète calculatoirement comme l'existence d'un foncteur FullPtsTc *tel que pour toute structure S implantant* PtsAlgos *(définition 4.32),* FullPtsTc(S) *implante la signature* PtsTc *(définition 4.22).*

Preuve On prouve successivement :

- `inf_ppal` : $\forall \Gamma, M.\, \Gamma \vdash \Rightarrow \mathsf{InfPpal}(T \mid \Gamma \vdash M : T, \leq)$
  Preuve par récurrence sur M. Comme nous aurons à faire une preuve similaire, dans un cadre plus général dans le prochain chapitre, nous ne détaillerons pas l'algorithme d'inférence de type, qui est assez classique.
- `chk_wk` : $\forall \Gamma, M, T.\, \Gamma \vdash\ \wedge\ T \in \mathcal{WT}_\Gamma\ \Rightarrow\ \mathsf{Dec}\,(\Gamma \vdash M : T)$
  inférer le type de M et appeler `le_type_dec` pour voir si c'est un sous-type de T.
- `add_typ` : $\forall \Gamma, T.\, \Gamma \vdash \Rightarrow\ \mathsf{Dec}\,(\Gamma\,[T] \vdash)$
  inférer le type de T, et vérifier que c'est un sous-type d'une sorte grâce à `least_sort`.
- `add_def` : $\forall \Gamma, M, T.\, \Gamma \vdash \Rightarrow\ \mathsf{Dec}\,(\Gamma\,[M : T] \vdash)$
  composer `add_typ` et `chk_wk` puisque $\Gamma\,[T] \vdash$ implique $T \in \mathcal{WT}_\Gamma$.

- `chk_wft` : $\forall \Gamma, T. \Gamma \vdash \Rightarrow \mathsf{Dec}\,(T \in \mathcal{WT}_\Gamma)$
  tester si T est une sorte ; sinon, appeler `add_typ`.
- `chk_typ` : $\forall \Gamma, M, T. \Gamma \vdash \Rightarrow \mathsf{Dec}\,(\Gamma \vdash M : T)$
  composer `chk_wft` et `chk_wk`.

■

Les deux dernières spécifications de PtsTc `chk_wf` et `chk_wft` ne correspondent à aucune commande de notre vérificateur de preuves. Les spécifications `chk_wk` et `chk_typ` ne diffèrent que par la précondition suppléementaire dans `chk_wk`. Celle-ci dispense `chk_wk` de typer l'argument T. Elles sont tout de même exportées car elles peuvent servir à écrire des tactiques. Il arrive assez souvent que l'on veuille faire une vérification de type sachant déjà que le type proposé est bien formé. Le fait que `chk_wk` soit utilisé plusieurs fois dans les fonctions du noyau montre en soi que c'est une fonction auxiliaire importante. Nous rappellerons tout de même qu'il ne faut pas trop recourrir à ce procédé qui trahit un peu l'algorithme employé dans le noyau.

Nous avons isolé un ensemble de conditions (PtsAlgos) pour que le typage soit décidable. Mais il reste des paramètres assez complexes à prouver comme par exemple la décidabilité du sous-typage. Il y a peu d'espoir de faire plus précis si on n'a pas plus d'informations sur le sous-typage. Il est temps de considérer des cas particuliers de sous-typage et dans chaque cas, reproduire notre méthode pour dégager des conditions simples pour que le sous-typage soit décidable. Le reste de ce chapitre se chargera d'étudier le cas de la cumulativité, ainsi que plusieurs hiérarchies de sortes.

Remarques méthodologiques complémentaires

Avant de poursuivre en raffinant les conditions de PtsAlgos, nous faisons quelques remarques sur la manière dont est apparue cette structure. Le point de départ est l'algorithme de typage du Calcul des Constructions prouvé dans un précédent développement [5]. Il est clair que cet algorithme peut s'étendre à "une certaine classe de PTS", sans que nous n'ayons d'idée précise sur ce qu'est exactement cette classe. Nous avons donc repris la preuve de décidabilité pour CC et à chaque fois qu'il se dégageait une condition générique, comme par exemple la décidabilité de l'inférence de Axiom, celle-ci était rajoutée à la liste des hypothèses. Un autre critère pour ajouter une condition est l'impossibilité de prouver un résultat du fait des paramètres choisis : la décidabilité du sous-typage ne peut évidemment pas se prouver pour une relation abstraite.

Une fois la décidabilité du typage prouvée, le travail consiste à analyser les hypothèses faites afin de déterminer si certaines ne peuvent pas se déduire à partir des autres. Par exemple, décider si une sorte est typée ou non est une conséquence facile de l'algorithme `infer_axiom`. Jusqu'ici, si l'on est assez rigoureux, on a toujours une condition nécessaire et suffisante pour que notre PTS soit décidable. Parfois, pour avoir une présentation plus digeste, on affaiblit les conditions, soit pour les rendre individuellement plus simples à comprendre, soit pour en "fusionner" plusieurs, en introduisant une propriété dont elles sont la conséquence. Comme exemple, nous citerons `infer_rule`, qui fusionne une condition similaire à `infer_axiom`, mais pour Rule, ainsi que la condition de PTS full. Une fois ce travail terminé, on collecte toutes les conditions restantes, et l'on forme la structure PtsAlgos.

Ainsi, le réel travail de conception et de choix entre différentes options ne se fait pas au moment de prouver le théorème qui nous intéresse, mais après. Cette technique nous est apparue très productive : elle permet de généraliser très largement un théorème avec un faible coût, puisqu'il s'agit plus de choisir quels seront les paramètres de notre problème plutôt que de découvrir de nouvelles techniques de preuves.

4.4 Règles de réduction

Pour avoir une idée de comment nous allons instancier le paramètre de sous-typage, voyons comment cette relation serait définie dans certains systèmes de types.

Dans le cas des PTS, ce serait simplement la fermeture réflexive, symétrique, transitive et congruente de la β-réduction :

$$\frac{}{\Gamma \vdash (\lambda T.\, M\ N) \equiv_\beta M\{0\backslash N\}} \qquad \frac{}{\Gamma \vdash M \equiv_\beta M}$$

$$\frac{\Gamma \vdash M \equiv_\beta N}{\Gamma \vdash N \equiv_\beta M} \qquad \frac{\Gamma \vdash M \equiv_\beta N \qquad \Gamma \vdash N \equiv_\beta P}{\Gamma \vdash M \equiv_\beta P}$$

$$\frac{\Gamma \vdash M \equiv_\beta M'}{\Gamma \vdash \lambda M.\, N \equiv_\beta \lambda M'.\, N} \qquad \frac{\Gamma\,[N] \vdash M \equiv_\beta M'}{\Gamma \vdash \lambda N.\, M \equiv_\beta \lambda N.\, M'}$$

$$\frac{\Gamma \vdash M_1 \equiv_\beta N_1}{\Gamma \vdash (M_1\ M_2) \equiv_\beta (N_1\ M_2)} \qquad \frac{\Gamma \vdash M_2 \equiv_\beta N_2}{\Gamma \vdash (M_1\ M_2) \equiv_\beta (M_1\ N_2)}$$

$$\frac{\Gamma \vdash M_1 \equiv_\beta N_1}{\Gamma \vdash \Pi\, M_1.\, M_2 \equiv_\beta \Pi\, N_1.\, M_2} \qquad \frac{\Gamma\,[M_1] \vdash M_2 \equiv_\beta N_2}{\Gamma \vdash \Pi\, M_1.\, M_2 \equiv_\beta \Pi\, M_1.\, N_2}$$

Le système de Coq comporte une notion assez limitée de sous-typage : la cumulativité qui consiste à rajouter des inclusions entre sortes. Il existe une relation $\prec$ entre les sortes telle que si $s_1 \prec s_2$, alors tout type de sorte s_1 est aussi un type de sorte s_2. La relation de sous-typage est alors la fermeture réflexive, transitive et congruente (pour les types) de la réunion de la β-conversion et de la cumulativité :

$$\frac{\Gamma \vdash M \equiv_\beta N}{\Gamma \vdash M \leq^*_{\mathrm{C}} N} \qquad \frac{}{\Gamma \vdash \mathrm{Type}(i) \leq^*_{\mathrm{C}} \mathrm{Type}(i{+}1)} \qquad \frac{}{\Gamma \vdash M \leq^*_{\mathrm{C}} M}$$

$$\frac{\Gamma \vdash M \leq^*_{\mathrm{C}} N \qquad \Gamma \vdash N \leq^*_{\mathrm{C}} P}{\Gamma \vdash M \leq^*_{\mathrm{C}} P} \qquad \frac{\Gamma \vdash A_2 \leq^*_{\mathrm{C}} A_1 \qquad \Gamma\,[A_2] \vdash B_1 \leq^*_{\mathrm{C}} B_2}{\Gamma \vdash \Pi\, A_1.\, B_1 \leq^*_{\mathrm{C}} \Pi\, A_2.\, B_2}$$

Si nous voulions intégrer dans notre système d'autres règles de réduction comme l'expansion des constantes, la réduction du filtrage ou des points fixes, il faudrait modifier ces définitions simplement en rajoutant la règle adéquate. Il apparaît que ces définitions suivent souvent le même schéma, et nous allons le traiter de manière générale, pour éviter que le développement formel ne souffre de ces répétitions.

Dans ces différents tableaux, on peut à chaque fois distinguer deux sortes de règles : d'une part celle qui introduisent de nouvelles fonctionalités, comme la β-réduction qui explique comment s'effectuent les appels de fonctions, ou la δ-réduction qui introduit l'expansion des abbréviations. D'autre part, il y a celles qui permettent simplement de propager les réductions dans les différents sous-termes, ou de les enchainer. Cette deuxième sorte se retrouve souvent inchangée suivant les règles de base (β, δ, etc.) que l'on incorpore au système.

Pour traiter ces schémas de manière générique, nous allons définir des *opérateurs de règles de réduction*, i.e. des règles de réductions paramétrées par une règle de base, et nous montrerons que ces opérateurs préservent toutes les propriétés attendues pour les règles de réduction.

Une nouvelle notion apparaît dans ces exemples : celle de règle de réduction. Une règle de réduction est une égalité que l'on n'utilise que dans un sens, qui généralement simplifie

$$\text{(Step)}\ \frac{\Gamma \vdash M \to_R N}{\Gamma \vdash M \triangleright_R N}$$

$$\text{(Abs-L)}\ \frac{\Gamma \vdash M \triangleright_R M'}{\Gamma \vdash \lambda M.\, N \triangleright_R \lambda M'.\, N}$$

$$\text{(Abs-R)}\ \frac{\Gamma\,[N] \vdash M \triangleright_R M'}{\Gamma \vdash \lambda N.\, M \triangleright_R \lambda N.\, M'}$$

$$\text{(App-L)}\ \frac{\Gamma \vdash M_1 \triangleright_R N_1}{\Gamma \vdash (M_1\ M_2) \triangleright_R (N_1\ M_2)}$$

$$\text{(App-R)}\ \frac{\Gamma \vdash M_2 \triangleright_R N_2}{\Gamma \vdash (M_1\ M_2) \triangleright_R (M_1\ N_2)}$$

$$\text{(Prod-L)}\ \frac{\Gamma \vdash M_1 \triangleright_R N_1}{\Gamma \vdash \Pi\, M_1.\, M_2 \triangleright_R \Pi\, N_1.\, M_2}$$

$$\text{(Prod-R)}\ \frac{\Gamma\,[M_1] \vdash M_2 \triangleright_R N_2}{\Gamma \vdash \Pi\, M_1.\, M_2 \triangleright_R \Pi\, M_1.\, N_2}$$

FIG. 4.6: Fermeture contextuelle d'une règle de réduction

la formule. La principale utilité est que cela permet de mettre des expressions en forme normale. Si la règle de réduction possède la propriété de confluence, on pourra alors décider de l'égalité de deux expressions simplement en comparant les formes normales, en supposant que l'on ne compare que des termes fortement normalisables.

Cette section définira donc aussi les propriétés des règles de réductions utiles, et fournira des outils génériques permettant de prouver les résultats capitaux concernant les règles de réduction : confluence, auto-réduction, décidabilité du sous-typage.

Dans la majeure partie de cette section, nous allons considérer un paramètre R, que nous appelerons règle de réduction. Il s'agit d'une relation de domaine $\text{Ctx} \times \text{Term} \times \text{Term}$ (comme pour les règles de sous-typage). Le jugement $\Gamma \vdash M \to_R N$ se lit : dans le contexte Γ, le terme M se réduit vers N selon la règle de réduction R.

4.4.1 Définition des opérateurs

La fermeture réflexive transitive R^* d'une relation R a déjà été définie section 4.2.3. Nous introduisons maintenant les autres opérateurs comme la réunion de deux règles, la fermeture contextuelle, ou la fermeture réflexive symétrique transitive.

Définition 4.33 (`ctxt`) *L'opérateur de fermeture contextuelle (ou congruente) de R permet d'appliquer une réduction dans n'importe quel sous-terme. Les règles de la figure 4.6 définissent $[R]$, l'image de R par cet opérateur. On notera $\Gamma \vdash M \triangleright_R N$ au lieu de $\Gamma \vdash M \to_{[R]} N$.*

Définition 4.34 (`reunion`) *L'opérateur de réunion de deux règles de réduction R_1 et R_2 est :*

$$\Gamma \vdash M \to_{R_1 \cup R_2} N \stackrel{\text{def}}{\equiv} \Gamma \vdash M \to_{R_1} N \vee \Gamma \vdash M \to_{R_2} N$$

Définition 4.35 (`transp`) *Le symétrique d'une relation R est définie par :*

$$\Gamma \vdash M \to_{R^{-1}} N \stackrel{\text{def}}{\equiv} \Gamma \vdash N \to_R M$$

Définition 4.36 (`R_rst, conv`) *La fermeture réflexive, symétrique et transitive de R est définie par*

$$R^{=} \stackrel{\text{def}}{\equiv} (R \cup R^{-1})^{*}.$$

De plus, la fermeture réflexive, symétrique, transitive et congruente de R (i.e. $[R]^{=}$) sera notée $\equiv_R$.

Nous montrons ensuite que ces opérateurs préservent toutes les propriétés de stabilité requises pour former une règle de sous-typage.

Lemme 4.37 *Les ensembles de règles* Rlift, Rsubstwk *et* Rstable *sont clos par les opérateurs de réunion, de fermeture réflexive, symétrique, transitive ou congruente :*

$$\begin{array}{l} R \in \mathsf{Rlift} \Rightarrow R^{*} \in \mathsf{Rlift} \\ R \in \mathsf{Rlift} \Rightarrow R^{=} \in \mathsf{Rlift} \\ R \in \mathsf{Rlift} \Rightarrow [R] \in \mathsf{Rlift} \\ \left.\begin{array}{l} R_1 \in \mathsf{Rlift} \\ R_2 \in \mathsf{Rlift} \end{array}\right\} \Rightarrow R_1 \cup R_2 \in \mathsf{Rlift} \end{array} \qquad \begin{array}{l} R \in \mathsf{Rsubstwk} \Rightarrow R^{*} \in \mathsf{Rsubstwk} \\ R \in \mathsf{Rsubstwk} \Rightarrow R^{=} \in \mathsf{Rsubstwk} \\ R \in \mathsf{Rsubstwk} \Rightarrow [R] \in \mathsf{Rsubstwk} \\ \left.\begin{array}{l} R_1 \in \mathsf{Rsubstwk} \\ R_2 \in \mathsf{Rsubstwk} \end{array}\right\} \Rightarrow R_1 \cup R_2 \in \mathsf{Rsubstwk} \end{array}$$

$$\begin{array}{l} R \in \mathsf{Rstable} \Rightarrow R^{*} \in \mathsf{Rstable} \\ R \in \mathsf{Rstable} \Rightarrow R^{=} \in \mathsf{Rstable} \\ R \in \mathsf{Rstable} \Rightarrow [R] \in \mathsf{Rstable} \\ \left.\begin{array}{l} R_1 \in \mathsf{Rstable} \\ R_2 \in \mathsf{Rstable} \end{array}\right\} \Rightarrow R_1 \cup R_2 \in \mathsf{Rstable} \end{array}$$

Pour résumer, on peut dire que si R est invariante ($R \in \mathcal{BR}$), alors les relations de réduction $[R]$,R^{*} et $R^{=}$ le sont aussi. De plus, R^{*} et $R^{=}$ sont des règles de sous-typage ($R^{*} \in$ SubRule) car elle sont réflexives et transitives.

4.4.2 Propriétés des règles de réduction

L'une des hypothèses de la signature PtsAlgos est la décidabilité du sous-typage. Dans le cadre de cette section, il s'agit d'écrire un algorithme générique permettant de décider $\equiv_R$, pour les types bien formés. Comme nous supposerons que notre calcul est fortement normalisable (tout terme bien typé est fortement normalisable), il suffit de savoir décider la convertibilité sur l'ensemble des termes fortement normalisables.

Nous ne traiterons ici que les cas simples, où notre réduction est Church-Rosser et admet un algorithme de mise en forme normale de tête. Lorsque l'on sort de cette grande voie bien balisée, il serait préférable de se tourner vers des solutions plus élaborées comme celles que l'on peut trouver dans la thèse de Hullot [34].

Définition 4.38 (prédicat `confluent`) *La propriété de confluence forte, que l'on définit souvent à l'aide d'un simple diagramme de commutation, s'énonce :*

$$\textsc{Confl} \stackrel{\text{def}}{\equiv} \left\{ R \;\middle|\; \begin{array}{r} \forall \Gamma, A, B, C.\ \Gamma \vdash A \to_R B \ \wedge\ \Gamma \vdash A \to_R C \\ \Rightarrow\ \exists D.\ \Gamma \vdash B \to_R D \ \wedge\ \Gamma \vdash C \to_R D \end{array} \right\}$$

Formellement, on définit la confluence forte en termes de commutation :

$$\textsc{Confl} = \left\{ R \;\middle|\; R^{-1}; R \subseteq R; R^{-1} \right\}$$

Attention : cette définition correspond en fait à la confluence forte. La confluence faible de R est la confluence forte de R^{*}.

Définition 4.39 (prédicat `church_rosser`) *Une règle de réduction R vérifie la propriété de Church-Rosser si pour toute paire de termes convertibles M et N, il existe un réduit commun T :*

$$\mathcal{CR} \stackrel{\text{def}}{\equiv} \{R \mid \forall \Gamma, M, N.\, \Gamma \vdash M \equiv_R N \;\Rightarrow\; \exists T.\, \Gamma \vdash M \triangleright_R^* T \;\wedge\; \Gamma \vdash N \triangleright_R^* T\}$$

Cette propriété est très importante car elle permet de conclure à l'unicité des formes normales (dans la mesure où elles existent).

Lemme 4.40 (`confl_church_rosser`) *Toute relation possédant la propriéte de confluence vérifie aussi celle de Church-Rosser :*

$$\textsc{Confl} \subseteq \mathcal{CR}$$

Un bon nombre de preuves de confluence peuvent se montrer suivant le même schéma. Nous allons donc montrer une version abstraite des preuves de confluence à la Tait-Martin-Löf :

Théorème 4 (`TML_Church_rosser`) *Soit deux relations R et R' telles que*

$$R \subseteq R' \subseteq R^* \qquad R' \in \textsc{Confl}$$

Alors R est confluente : $R^* \in \textsc{Confl}$, et donc elle vérifie la propriété de Church-Rosser grâce au lemme précédent.

Pour montrer la confluence de la β-réduction, on peut utiliser ce théorème avec $R = \beta$ et R' la β-réduction *parallèle*, qui est fortement confluente.

Définition 4.41 (prédicat `normal`) *L'ensemble $\mathcal{NF}_\Gamma^R$ des formes normales d'une relation R dans le contexte Γ est l'ensemble des termes n'ayant aucun radical au sommet :*

$$\mathcal{NF}_\Gamma^R \stackrel{\text{def}}{\equiv} \{M \in \textsc{Term} \mid \forall N.\, \Gamma \vdash M \to_R N \;\Rightarrow\; \bot\}$$

En général, on dit que M est normal pour R s'il ne contient aucun radical, ce qui se représente par la proposition $M \in \mathcal{NF}_\Gamma^{[R]}$. Il faut noter que $M \in \mathcal{NF}_\Gamma^R$ signifie autre chose, à savoir qu'il n'existe pas de radical au sommet du terme M. Cette distinction est utile pour définir la notion de forme normale de tête de manière générique.

Définition 4.42 (prédicat `head_normal`) *Soit R une relation ternaire, et Γ un contexte. M est en forme normale de tête ($M \in \mathcal{HNF}_\Gamma^R$) si aucun radical n'apparaît au sommet d'un réduit de M :*

$$\mathcal{HNF}_\Gamma^R \stackrel{\text{def}}{\equiv} \left\{M \in \textsc{Term} \;\middle|\; \forall N.\, \Gamma \vdash M \triangleright_R^* N \;\Rightarrow\; N \in \mathcal{NF}_\Gamma^R\right\}$$

Cette propriété signifie qu'il ne peut y avoir de radicaux que dans les sous-termes de M, et ces réductions ne peuvent pas faire apparaître de radical au sommet du terme. C'est une notion un peu différente de la définition habituelle dans le cas de β (où les formes normales de tête sont soit les abstractions, soit les applications dont le sous-terme en tête d'application est une variable). Par exemple, pour nous, $(\lambda T.\, \natural 0\ \natural 0\ \natural 0)$ est en forme normale de tête alors qu'elle ne l'est pas au sens usuel, puisque la tête de l'application est une abstraction. Le fait est qu'en général, pour savoir si un terme est en forme normale de tête, il faut le réduire jusqu'à obtenir la forme normale de tête au sens usuel. Notre définition a l'avantage d'être indépendante de la relation de réduction choisie (il faudrait adapter la notion de forme normale de tête si l'on considérait la δ-réduction, en disant qu'en plus la variable de tête n'est pas associée à une définition).

Au vu de cette définition, il est très facile de voir que l'ensemble des formes normales de tête est stable par réduction, i.e. pour tout réduit d'un terme en forme normale de tête est aussi en forme normale de tête.

Définition 4.43 (prédicat `sn`) *L'ensemble des termes fortement normalisables dans le contexte Γ, vis-à-vis d'une règle de réduction R, est défini à l'aide du prédicat d'accessibilité, comme nous l'avons déjà vu :*

$$\mathcal{SN}_\Gamma^R \stackrel{\text{def}}{\equiv} \mathrm{Acc}_{R_\Gamma^{-1}}$$

Cet ensemble est lui aussi stable par réduction.

4.4.3 Préservation de l'auto-réduction par fermeture contextuelle

La propriété d'auto-réduction a un intérêt en soi : elle prouve que les calculs se produiront sans erreur (par exemple, un objet non fonctionnel ne se retrouvera jamais appliqué) ; elle joue aussi un rôle dans la preuve de cohérence. Mais du point de vue de la décidabilité du typage, on n'en a besoin que pour montrer que l'on sait décider si un type est convertible avec un produit bien formé.

Définition 4.44 (prédicat `rule_sound`) *L'ensemble des règles vérifiant l'auto-réduction est noté $\mathcal{SR}$. Sa définition est :*

$$\mathcal{SR} \stackrel{\text{def}}{\equiv} \{R \mid \forall \Gamma, M, N, T. \Gamma \vdash M : T \ \wedge\ \Gamma \vdash M \to_R N \ \Rightarrow\ \Gamma \vdash N : T\}$$

On prouve que les opérateurs définis dans cette section préservent la notion d'auto-réduction. C'est évident pour les opérateurs de réunion, de fermeture transitive (éventuellement réflexive). Le cas de la fermeture contextuelle est plus complexe, et nécessite quelques conditions annexes.

Lemme 4.45 (`ctxt_sound`) *Soit R une relation ternaire telle que*

$$R \in \mathsf{Rlift} \qquad [R] \ \subseteq \leq^{-1}$$

Sous ces conditions, la propriété d'auto-réduction est préservée par fermeture contextuelle :

$$R \in \mathcal{SR} \ \Rightarrow\ [R] \in \mathcal{SR}$$

La première hypothèse est directement héritée du lemme de réduction dans l'environnement (lemme 4.29). La seconde sert à affaiblir les hypothèses $(\Gamma) \to_{[R]} (\Gamma')$ en $(\Gamma') \leq (\Gamma)$ lorsque la réduction se fait dans le sous-terme gauche d'un produit ou d'une abstraction, ce qui permet d'utiliser le lemme 4.29 pour montrer que le sous-terme droit reste bien typé dans le nouveau contexte.

4.4.4 Décidabilité de la convertibilité

La spécification du test de convertibilité est la décidabilité de la relation de conversion :

$$\forall \Gamma. \forall M, N \in \mathcal{SN}_\Gamma^{[R]}. \mathsf{Dec}\,(\Gamma \vdash M \equiv_R N)$$

Cela signifie que nous allons construire une fonction qui étant donné deux termes M et N, construit une dérivation de conversion entre M et N, en n'utilisant que la réflexivité, la transitivité ou l'une des règles de R, ou bien échoue. Le problème est de choisir la bonne règle à appliquer. En particulier, la règle de transitivité peut toujours s'appliquer, indépendamment de la forme de M et N. La transitivité pose le problème supplémentaire d'introduire une inconnue (le terme intermédiaire ne peut pas se déduire de la forme de M et N).

Nous allons chercher un système équivalent à $\equiv_R$ pour lequel les conclusions des différentes règles n'ont pas d'intersection commune, ce qui fait qu'au plus une règle peut s'appliquer, lorsque l'on connait la forme de M et N. On dit qu'il est dirigé par la syntaxe.

Supposer que notre règle de réduction possède la propriété de Church-Rosser simplifie beaucoup le problème car il signifie que toute dérivation de conversion peut se mettre sous la forme de deux chemins de réduction à partir de M et N aboutissant au même terme (un réduit commun). Cette fois, nous n'avons plus à déviner de terme puisque la réduction est déterministe (une fois choisie la stratégie).

Cet algorithme, qui consiste à comparer les formes normales de M et N, peut être raffiné. Au lieu de normaliser les termes, on les met en forme normale de tête, on compare les constructeurs en tête de terme, et l'on vérifie récursivement la convertibilité des sous-termes. En cas de réussite, cet algorithme n'est pas plus efficace, car l'on finit quand même par calculer (implicitement) la forme normale de nos termes de départ[2]. En revanche, un echec peut être détecté plus rapidement.

Nous commençons par définir un prédicat d'inversion de $\equiv_R$. Nous montrerons qu'il est équivalent à la relation d'origine sur les formes normales de tête, et qu'il est dirigé par la syntaxe. Décider ce système est beaucoup plus direct.

Définition 4.46 (prédicat `conv_hn_inv`) *Le prédicat d'inversion de $\equiv_R$ défini par les règles ci-dessous permet de ne faire des pas de conversion que sous au moins un constructeur.*

$$\frac{}{\Gamma \vdash s \equiv_R^{\text{inv}} s}\ (s \in \text{SORT}) \qquad \frac{}{\Gamma \vdash \natural n \equiv_R^{\text{inv}} \natural n}$$

$$\frac{\Gamma \vdash A \equiv_R A' \qquad \Gamma\,[A'] \vdash M \equiv_R M'}{\Gamma \vdash \lambda A.\, M \equiv_R^{\text{inv}} \lambda A'.\, M'}$$

$$\frac{\Gamma \vdash M \equiv_R M' \qquad \Gamma \vdash N \equiv_R N'}{\Gamma \vdash (M\ N) \equiv_R^{\text{inv}} (M'\ N')}$$

$$\frac{\Gamma \vdash A \equiv_R A' \qquad \Gamma\,[A'] \vdash B \equiv_R B'}{\Gamma \vdash \Pi\, A.\, B \equiv_R^{\text{inv}} \Pi\, A'.\, B'}$$

Attention, ce prédicat n'est pas défini récursivement. Les prémisses des règles font appel à la relation de conversion définie précédemment. Il s'agit d'une technique de preuve assez courante en Coq et nous nous en servirons à chaque fois que nous aurons à prouver la décidabilité de la fermeture transitive d'une relation.

Lemme 4.47 (`conv_hn_inv_sym`, `conv_hn_inv_trans`) *La relation d'inversion de la conversion est symétrique et transitive :*

$$\forall \Gamma, M, N.\ \Gamma \vdash M \equiv_R^{\text{inv}} N \ \Rightarrow\ \Gamma \vdash N \equiv_R^{\text{inv}} M$$
$$\forall \Gamma, M, N, P.\ \Gamma \vdash M \equiv_R^{\text{inv}} N \ \wedge\ \Gamma \vdash N \equiv_R^{\text{inv}} P \ \Rightarrow\ \Gamma \vdash M \equiv_R^{\text{inv}} P$$

Preuve La preuve de ces deux résultat font appel à l'hypothèse $R \in$ Rstable pour traiter les cas de l'abstraction et du produit. ■

Lemme 4.48 (`inv_conv_conv`, `inv_conv_hn`) *Soit R une règle de réduction vérifiant la propriété de Church-Rosser. La relation $\equiv_R$ est équivalente à $\equiv_R^{\text{inv}}$ sur l'ensemble des termes en forme normale de tête :*

$$\Gamma \vdash M \equiv_R^{\text{inv}} N \ \Rightarrow\ \Gamma \vdash M \equiv_R N$$
$$R \in \mathcal{CR} \ \wedge\ M, N \in \mathcal{HNF}_\Gamma^R \ \wedge\ \Gamma \vdash M \equiv_R N \ \Rightarrow\ \Gamma \vdash M \equiv_R^{\text{inv}} N$$

[2] Dans le cas où notre réduction inclut la δ-réduction, une amélioration consiste à n'expanser les constantes que lorsque cela est nécessaire.

Preuve Le premier résultat est évident. Pour le deuxième, on commence par prouver que si un terme M en forme normale de tête se réduit vers N, alors on a $\Gamma \vdash M \equiv_R^{\text{inv}} N$. Le résultat est alors une conséquence de la propriété de Church-Rosser. ■

Nous allons maintenant définir les ordres selon lesquels nous ferons des appels récursifs dans les algorithmes de mise en forme normale de tête et de conversion. Il nous faut pouvoir faire des appels récursifs avec un réduit (lorsque l'on a détecté un radical), mais aussi avec un sous-terme (lorsque nous avons obtenu la forme normale de tête, il faut recommencer avec les sous-termes). La réunion de ces deux ordres (que nous appelerons ordre de normalisation) est bien fondée sur l'ensemble des termes fortement normalisables.

Comme notre règle de réduction peut dépendre du contexte, et que celui-ci varie au cours des appels récursifs, notre relation ne portera pas sur des termes, mais sur des couples contexte-terme.

Définition 4.49 (prédicat `subterm`) *L'ordre de sous-terme $\subset_{st}$ est défini ainsi :*

$$(\Gamma, M) \subset_{st} (\Gamma, (M\ N)) \qquad (\Gamma, N) \subset_{st} (\Gamma, (M\ N))$$
$$(\Gamma, A) \subset_{st} (\Gamma, \lambda A.\, M) \qquad (\Gamma\,[A'], M) \subset_{st} (\Gamma, \lambda A.\, M)$$
$$(\Gamma, A) \subset_{st} (\Gamma, \Pi\, A.\, B) \qquad (\Gamma\,[A'], B) \subset_{st} (\Gamma, \Pi\, A.\, B)$$

Noter que dans le cas des lieurs, le contexte peut être étendu avec n'importe quelle valeur.

Définition 4.50 (prédicat `ord_norm`) *L'ordre de réduction est la réunion de l'ordre associé à la réduction R avec l'ordre des sous-termes. Il sera noté $\prec_R$.*

$$\frac{\Gamma \vdash M \to_R N}{(\Gamma, N) \prec_R (\Gamma, M)} \qquad \frac{V_1 \subset_{st} V_2}{V_1 \prec_{V2}}$$

Lemme 4.51 (`sn_acc_ord_norm1`) *Tout terme fortement normalisable est accessible pour l'ordre de normalisation :*

$$M \in \mathcal{SN}_\Gamma^R \ \Rightarrow\ (\Gamma, M) \in \mathrm{Acc}_{\prec_R}$$

Preuve La relation de sous-terme commute avec n'importe quelle congruence d'une relation appartenant à Rstable. Le théorème A.8 de l'annexe permet de conclure. ■

Un algorithme de mise en forme normale de tête est une fonction qui prend en entrée un terme M fortement normalisable et qui retourne un réduit de M qui soit en forme normale de tête. La précondition est évidemment nécessaire puisque certains termes non normalisables n'ont pas de forme normale de tête.

Algorithme 4.52 (`CR_WHNF_conv_dec`) *Soit R une relation invariante vérifiant la propriété de Church-Rosser ayant un algorithme de mise en forme normale de tête. Alors, la relation $\equiv_R$ est décidable sur l'ensemble des termes fortement normalisables.*

$$\forall \Gamma.\, \forall M, N \in \mathcal{SN}_\Gamma^{[R]}.\, \mathsf{Dec}\,(\Gamma \vdash M \equiv_R N)$$

Preuve L'algorithme est simple : soit M et N deux termes fortement normalisables à comparer. On met ces deux termes en forme normale de tête (resp. M' et N'). Le lemme précédent nous dit que M et N seront convertibles si et seulement si on a $\Gamma \vdash M' \equiv_R^{\text{inv}} N'$. On vérifie donc que M' et N' ont le même constructeur de tête (ce qui nécessite que l'égalité sur les sortes soit décidable) et l'on vérifie récursivement que les sous-termes sont deux à deux convertibles. Il reste à vérifier que ces appels récursifs sont légaux. Ceux-ci portent sur des sous-termes d'un réduit du terme avec lequel la fonction a été appelée. Cela correspond

à la fermeture transitive de l'ordre de réduction, qui est bien fondé sur l'ensemble des termes fortement normalisables. ■

Si nous voulions définir un PTS avec comme règle de sous-typage $\equiv_R$, nous aurions besoin du résultat d'inversion du produit pour instancier le champ `inv_prod` de PtsAlgos.

Lemme 4.53 (`inv_prod`) *Soit R une règle de réduction. Si les produits sont des formes normales pour R (i.e. pas de réduction au sommet du terme), et si R est Church-Rosser :*

$$\forall \Gamma, A, B. \, \Pi\, A.\, B \in \mathcal{NF}^R_\Gamma \qquad R \in \mathcal{CR}$$

alors nous avons la propriété d'inversion du produit

$$\Gamma \vdash \Pi\, A.\, B \equiv_R \Pi\, A'.\, B' \;\Rightarrow\; \Gamma \vdash A' \equiv_R A \;\wedge\; \Gamma\,[A'] \vdash B \equiv_R B'$$

Preuve En utilisant la confluence, on en déduit qu'il existe M, un réduit commun de $\Pi\, A.\, B$ et $\Pi\, A'.\, B'$. Comme les produits sont des formes normales (au sommet du terme), les réductions ne peuvent avoir lieu que dans les sous-termes. Ainsi, ces deux produits sont en forme normale de tête. Le lemme 4.48 permet d'en déduire $\Gamma \vdash \Pi\, A.\, B \equiv^{\text{inv}}_R \Pi\, A'.\, B'$, d'où le résultat, par inversion de cette propriété. ■

À ce point, nous pourrions conclure en introduisant une signature regroupant les hypothèses permettant la décidabilité du typage des PTS sans sous-typage (i.e. muni de $\equiv_R$). Cependant, dans la section suivante, nous allons définir une classe plus vaste de systèmes, et dont les conditions à satisfaire sont de complexité tout à fait similaire. L'étude de cette section ne se justifie que parce qu'elle sera utilisée pour définir les PTS cumulatifs.

4.5 Les PTS Cumulatifs (CTS)

Dans cette section, on n'étudiera qu'un seul exemple de sous-typage (qui comportera tout de même des paramètres), la cumulativité. Cela englobe évidemment le cas où il n'y a pas de sous-typage, puisqu'il suffit de considérer la relation vide pour l'inclusion entre sortes.

Le principe de la cumulativité est d'introduire des inclusions entre sortes à l'aide d'une relation $\prec$. Si l'on a $s_1 \prec s_2$, alors tout type de sorte s_1 sera, sans coercion explicite, un type de sorte s_2. Le sous-typage que nous allons considérer est engendrée par cette relation aiinsi qu'une règle de réduction R.

De plus, le sous-typage "passera au produit", c'est-à-dire que si A' est un sous-type de A et B un sous-type de B', alors $\Pi\, A.\, B$ sera un sous-type de $\Pi\, A'.\, B$. Cela fait apparaître le fait que le produit est covariant à droite (B et B' sont dans le même odre que les produits), alors qu'il est contravariant à gauche (A et A' sont en sens inverse).

La covariance du produit est utile pour que l'inférence de type se fasse simplement. En effet, sans la covariance, un terme M de type $\Pi\, A.\, B$ aura le type $\Pi\, A.\, B'$ s'il est sous forme de produit (en appliquant la règle de sous-typage avant celle de l'abstraction), mais pas si c'est une variable. Avec la covariance, l'application de la règle de sous-typage peut être retardée après l'abstraction.

La contravariance n'est pas aussi importante. La présentation originale de Luo [38] ne la considérait pas. Elle serait en revanche nécessaire si l'on considérait l'η-réduction : $\lambda A'.\, (M\ \natural 0)$ a le type $\Pi\, A'.\, B$, même sans la contravariance, alors que sa forme η-réduite n'aurait que le type $\Pi\, A.\, B$.

Les CTS sont donc des systèmes de types dont les paramètres sont les mêmes que les PTS, sauf que la relation de sous-typage est remplacée par les paramètres $\prec$ et R :

$$\frac{\Gamma \vdash A \equiv_R B}{\Gamma \vdash A \leq^*_{\mathrm{C}} B} \qquad \frac{s_1 \prec s_2}{\Gamma \vdash s_1 \leq^*_{\mathrm{C}} s_2}$$

$$\frac{\Gamma \vdash A' \leq^*_{\mathrm{C}} A \qquad \Gamma\,[A'] \vdash B \leq^*_{\mathrm{C}} B'}{\Gamma \vdash \Pi\, A.\, B \leq^*_{\mathrm{C}} \Pi\, A'.\, B'}$$

FIG. 4.7: Définition de la relation de cumulativité

Définition 4.54 (type `CTS_spec`) *Le type de structures regroupant les paramètres d'un CTS est :*

$$\begin{array}{lll}
\langle\ \textsc{Axiom}: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort}); & \text{(* Typage des sortes *)} \\
\quad \textsc{Rule}: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort} \times \textsc{Sort}); & \text{(* Formation des produits *)} \\
\quad R: & \mathcal{BR}; & \text{(* Règle de réduction *)} \\
\quad \prec: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort}) & \text{(* Inclusion entre sortes *)} \\
\rangle &&
\end{array}$$

On notera ce type $\mathcal{CTS}$.

Dans la suite, nous considérons $C = \langle \textsc{Axiom}, \textsc{Rule}, R, \prec \rangle$ un CTS quelconque. À partir de $\prec$ et R, nous allons définir la relation de cumulativité comme expliqué au début de cette section.

Définition 4.55 (prédicat `cumul`) *La cumulativité est définie figure 4.7.*

Si l'on considère la relation de cumulativité vide ($\prec = (s_1, s_2) \mapsto \bot$), nous nous retrouvons exactement dans le cas de la section précédente car $\leq^*_{\mathrm{C}} = \equiv_R$.

Lemme 4.56 (`cumul_rule`) *La relation de cumulativité est invariante, et donc sa fermeture réflexive transitive est une règle de sous-typage :*

$$\leq_{\mathrm{C}} \in \mathcal{BR} \qquad \leq^*_{\mathrm{C}} \in \textsc{SubRule}$$

On peut donc définir un PTS avec $\leq^*_{\mathrm{C}}$ comme règle de sous-typage.

Définition 4.57 (`cts_pts_functor`) *Soit $C = \langle \textsc{Axiom}, \textsc{Rule}, R, \prec \rangle \in \mathcal{CTS}$ un CTS, on définit alors* $\mathrm{CtsToPts}(C)$ *comme le PTS suivant :*

$$\mathrm{CtsToPts}(C) \stackrel{\text{def}}{\equiv} \langle \textsc{Axiom}, \textsc{Rule}, \leq^*_{\mathrm{C}} \rangle \in \mathcal{PTS}$$

$\mathrm{CtsToPts}(C)$ *est une spécification de PTS.*

Les PTS et les CTS sont formellement des structures différentes. Mais on peut considérer que la définition ci-dessus injecte les CTS dans les PTS. Dans le langage mathématique usuel, on dit que les CTS *sont* des PTS. (Il existe en Coq un mécanisme de coercions implicites qui rend compte de cela).

4.5.1 Décidabilité de la cumulativité

Nous suivrons le même schéma de preuve que pour les PTS sans sous-typage. La seule différence est que lorsque l'on compare deux sortes, il ne faut pas utiliser l'égalité mais la relation de cumulativité, et comme le produit est contravariant à gauche, il faudra échanger l'ordre des arguments, ce qui ne sera pas sans nous poser des problèmes pour prouver la terminaison.

Définition 4.58 (prédicat `cumul_hn_inv`) *Le prédicat d'inversion de la cumulativité généralise le prédicat* $\equiv_R^{\text{inv}}$ *:*

$$\frac{s_1 \prec^* s_2}{\Gamma \vdash s_1 \leq_{\text{C}}^{\text{inv}} s_2} \qquad \frac{}{\Gamma \vdash \natural n \leq_{\text{C}}^{\text{inv}} \natural n}$$

$$\frac{\Gamma \vdash A \equiv_R A' \qquad \Gamma\,[A'] \vdash M \equiv_R M'}{\Gamma \vdash \lambda A.\, M \leq_{\text{C}}^{\text{inv}} \lambda A'.\, M'}$$

$$\frac{\Gamma \vdash M \equiv_R M' \qquad \Gamma \vdash N \equiv_R N'}{\Gamma \vdash (M\ N) \leq_{\text{C}}^{\text{inv}} (M'\ N')}$$

$$\frac{\Gamma \vdash A' \leq_{\text{C}}^{*} A \qquad \Gamma\,[A'] \vdash B \leq_{\text{C}}^{*} B'}{\Gamma \vdash \Pi\, A.\, B \leq_{\text{C}}^{\text{inv}} \Pi\, A'.\, B'}$$

On remarquera que le sous-typage n'est invoqué que dans les prémisses du cas du produit, qui est le seul opérateur dont connaissons la variance des sous-termes.

Lemme 4.59 (`inv_cumul_trans`) *La relation* $\leq_{\text{C}}^{\text{inv}}$ *est transitive.*

Preuve Ce résultat n'est pas aussi trivial qu'il paraît. Les cas de l'abstraction et du produit sont plus compliqués : il faut que la conversion et la cumulativité soient invariants par conversion dans le contexte, et pas simplement par renforcement du contexte. C'est-à-dire, si on a $\Gamma \vdash A \leq_{\text{C}}^{*} B$, alors on doit avoir $\Gamma' \vdash A \leq_{\text{C}}^{*} B$ pour tout Γ' tel que $(\Gamma) \leq_{\text{C}}^{*} (\Gamma')$ ou $(\Gamma') \leq_{\text{C}}^{*} (\Gamma)$ (et pas simplement pour $(\Gamma') \leq_{\text{C}}^{*} (\Gamma)$). C'est ce qui a motivé la définition 4.13.

En effet, on a pour l'abstraction l'hypothèse suivante : $\lambda A.\, M \leq_{\text{C}} \lambda A'.\, M' \leq_{\text{C}} \lambda A''.\, M''$, ce qui donne par inversion : $\Gamma \vdash A \equiv_R A'$, $\Gamma\,[A'] \vdash M \equiv_R M'$, $\Gamma \vdash A' \equiv_R A''$ et $\Gamma\,[A''] \vdash M' \equiv_R M''$. Il faut prouver $\Gamma\,[A''] \vdash M \equiv_R M''$, ce qui est facile si l'on a $\Gamma\,[A''] \vdash M \equiv_R M'$. ■

Lemme 4.60 (`inv_cumul_cumul`, `inv_cumul_hn`) *Si la relation* R *est confluente et telle que les sortes et les produits sont des formes normales de tête pour* R*, alors la relation* $\leq_{\text{C}}^{\text{inv}}$ *est équivalente à* $\leq_{\text{C}}^{*}$ *sur l'ensemble de formes normales de tête de* R *:*

$$\Gamma \vdash M \leq_{\text{C}}^{\text{inv}} N \Rightarrow \Gamma \vdash M \leq_{\text{C}}^{*} N$$
$$M, N \in \mathcal{HNF}_\Gamma^R \ \wedge\ \Gamma \vdash M \leq_{\text{C}}^{*} N \Rightarrow \Gamma \vdash M \leq_{\text{C}}^{\text{inv}} N$$

Nous regroupons maintenant un ensemble de propriétés qui permettront de construire un algorithme décidant la relation de cumulativité.

Définition 4.61 (type `subtype_dec_CTS`) *Nous définissons* CtsSubDec *le sous-ensemble des CTS vérifiant les propriétés suivantes :*

$\langle$ *(* Nom *)* *(* Spécification *)*

- `scts_cr` : $R \in \mathcal{CR}$;
- `scts_hn_sort`: $\text{SORT} \subseteq \mathcal{NF}_\Gamma^R$;
- `scts_hn_prod`: $\forall A, B.\, \Pi\, A.\, B \in \mathcal{NF}_\Gamma^R$;
- `scts_whnf`: $\forall \Gamma.\, \forall M \in \mathcal{SN}_\Gamma^{[R]}.\, \exists^\star N.\, \Gamma \vdash M \triangleright_R^* N \ \wedge\ N \in \mathcal{HNF}_\Gamma^R$;
- `scts_rt_univ_dec`: $\forall s_1, s_2.\, \mathsf{Dec}\,(s_1 \prec^* s_2)$

$\rangle$

Les quatre premiers champs portent exclusivement sur la relation de réduction R, qui doit être Church-Rosser, posséder un algorithme de mise en forme normale de tête, et telle que les sortes et les produits soient des forme normales de tête. On peut alors utiliser l'algorithme de conversion générique `CR_WHNF_conv_dec`. La dernière condition (`scts_rt_univ_dec`) requiert que la fermeture réflexive transitive de l'inclusion entre sorte ($\prec^*$) soit décidable.

Les champs `scts_hn_sort` et `scts_hn_prod` traduisent le fait que les sortes et le produit sont des constructeurs de types canoniques, et donc ne peuvent pas donner lieu à d'autres simplifications.

Ce qui est remarquable est que l'on a découplé les hypothèses portant sur les paramètres décrivant la hiérarchie de sortes (Axiom, Rule et $\prec$) et la réduction R. Cela signifie que nous pourrons étudier des hiérarchies de sortes différentes sans que cela interfère avec la réduction.

Pour montrer la terminaison de l'algorithme de conversion sur les termes fortement normalisables, il nous faut introduire un nouvel ordre. Comme la contravariance nous fait parfois échanger les arguments, il nous faut tenir compte de la décroissance des deux termes.

Définition 4.62 (prédicat `ord_conv`)

$$\frac{V \prec_R V' \qquad W \prec_R W'}{(V,W) \precsim_R (V',W')} \qquad \frac{V \prec_R V' \qquad W \prec_R W'}{(W,V) \precsim_R (V',W')}$$

Le lemme suivant nous garantit que l'algorithme de conversion termine lorsqu'appelé avec des termes fortement normalisables.

Lemme 4.63 (`sn_acc_ord_conv`) *Si M et N sont fortement normalisables, alors la quantité $((\Gamma, M), (\Gamma, N))$ est accessible pour l'ordre de normalisation :*

$$M, N \in \mathcal{SN}_\Gamma^{[R]} \;\Rightarrow\; ((\Gamma, M), (\Gamma, N)) \in \mathrm{Acc}_{\precsim_R}$$

Il aurait probablement été plus simple de définir directement l'ordre dont nous avons besoin. Pourtant, le définir en utilisant des définitions de la bibliothèque standard nous évite de devoir refaire des preuves de bonne fondation qui sont toujours un peu délicates.

Algorithme 4.64 (`CR_WHNF_cumul_dec`) *Pour tout CTS appartenant à* CtsSubDec*, la relation $\leq_{\mathrm{C}}^*$ est décidable sur l'ensemble des termes fortement normalisables.*

Lemme 4.65 (`cumul_inv_prod`) *Pour tout CTS appartenant à* CtsSubDec*, la relation $\leq_{\mathrm{C}}^*$ possède la propriété d'inversion du produit.*

Preuve Comme les produits sont des formes normales de tête, il suffit d'employer le lemme `inv_cumul_hn`. ■

4.5.2 Décidabilité du typage des CTS

La structure suivante regroupe les hypothèses complémentaires que nous ferons pour prouver la décidabilité du typage.

Définition 4.66 (`norm_sound_CTS`)

$\langle$ *(* Nom *)* *(* Spécification *)*

`ncts_sr`: $R \in \mathcal{SR}$;

`ncts_typ_sn`: $\mathcal{WT}_\Gamma \subseteq \mathcal{SN}_\Gamma^{[R]}$;

`ncts_axiom`: $\forall s_1.\, \mathsf{InfPpal}(s_2 \mid (s_1, s_2) \in \textsc{Axiom}, \prec^*)$;

`ncts_rule`: $\forall s_1, s_2.\, \exists^{\mathsf{Ppal}}(s_3 \mid \exists s_1', s_2'.\, s_1 \prec^* s_1' \wedge s_2 \prec^* s_2' \wedge (s_1', s_2', s_3) \in \textsc{Rule}, \prec^*)$;

`ncts_hierarchy`: $\forall s_1, s_2, s_1'.\, s_1' \prec^* s_1 \wedge (s_1, s_2) \in \textsc{Axiom} \Rightarrow \exists s_2'.\, (s_1', s_2') \in \textsc{Axiom}$

$\rangle$

Les conditions `ncts_axiom` et `ncts_rule` viennent du fait que $\leq_{\mathrm{C}}^*$ restreint aux sortes et $\prec^*$ se confondent. Pour éviter les paradoxes, il est préférable que sous-typage et cumulativité aillent dans le même sens (i.e. leur réunion ne doit pas avoir de cycles), bien que cela ne soit pas une condition nécessaire (cf Système F cyclique). La condition `ncts_hierarchy` (que nous utiliserons pour le typage de l'abstraction) se justifie ainsi.

Ici encore, si l'on exclut l'hypothèse de normalisation forte `ncts_typ_sn`, on a découplé les propriétés portant sur la règle de réduction et la hiérarchie de sortes.

Précisons qu'à ce point précis du developpement, tous les résultats des sections 4.4 et précédentes sont acquis. Cela signifie que nous disposons de résultats métathéoriques comme le lemme de substitution, ce qui nous permettrait de prouver le résultat d'auto-réduction d'une règle de réduction comme β, après avoir prouvé qu'il s'agit bien d'une règle de réduction invariante et confluente. Nous pourrions alors faire la preuve de normalisation forte requise dans la structure que nous venons de définir. Si en revanche, on souhaite admettre la normalisation forte, en se reposant sur un résultat déjà prouvé dans la littérature, comme la preuve de normalisation forte générique de Melliès et Werner [41], il serait intéressant de décrire explicitement une structure regroupant les hypothèses de ce théorème.

On montre d'abord les algorithmes de test de convertibilité vers une sorte ou un produit (cf champs `least_sort` et `least_prod` de la définition PtsAlgos), puis les autres algorithmes ne posent aucune difficulté notable.

Lemme 4.67 (`cts_prim_algos`) *Tout CTS vérifiant les conditions des définitions 4.61 et 4.66 appartient à* PtsAlgos *:*

$$C \in \mathrm{CtsSubDec} \cap \mathrm{CtsNormSound} \Rightarrow \mathrm{CtsToPts}(C) \in \textsc{PtsAlgos}$$

Théorème 5 (`full_cts_type_checker`) *Tout CTS vérifiant les conditions du lemme précédent admet des jugements de typage décidables.*

$$C \in \mathrm{CtsSubDec} \cap \mathrm{CtsNormSound} \Rightarrow \mathrm{CtsToPts}(C) \in \textsc{PtsTc}$$

Preuve C'est une conséquence immédiate du lemme précédent et du résultat 3. ■

4.6 Les règles de réduction classiques

La règle de réduction que l'on considèrera sera souvent la réunion d'un certain nombre de règles plus élémentaires comme β ou δ (les seules règles que nous formaliserons), mais

on peut vouloir aussi ajouter des règles spécifiques pour certaines constantes. Le fait d'avoir isolé un nombre assez faible de conditions à vérifier est bon pour la modularité.

Il faut remarquer que bien que nous ayons fait des hypothèses relativement classiques sur R, cela ne permettrait pas de traiter la η-réduction. En effet, celle-ci n'est pas confluente sur les termes à la Church mal typés. La technique qui permet de montrer la confluence sur l'ensemble des termes bien typés ne marche pas car la confluence est perdue même sur les termes bien typés, à cause du sous-typage éventuel. En effet, $\lambda A.\,(\lambda B.\,M\ \natural 0)$ est bien typé lorsque A est un sous-type de B, ce qui empêche dans le cas général de refermer la paire critique entre η et β.

4.6.1 Définition de β et δ

Définition 4.68 (`beta, beta_rule`) *La β-réduction se définit ainsi :*

$$\frac{}{\Gamma \vdash (\lambda T.\,M\ \ N) \rightarrow_\beta M\{0 \backslash N\}}\ (\beta)$$

Nous pouvons prouver qu'elle est invariante : $\beta \in \mathcal{BR}$.

La règle δ dépend effectivement du contexte. Il ne faut pas oublier de reloger la valeur, de la même manière que l'on relogeait le type pour le typage.

Définition 4.69 (`delta, delta_rule`)

$$\frac{\Gamma(n) = [M : T]}{\Gamma \vdash \natural n \rightarrow_\delta \uparrow^{n+1} M}\ (\delta)$$

Nous pouvons aussi prouver qu'elle est invariante : $\delta \in \mathcal{BR}$.

Dans la suite, on distinguera les propriétés à satisfaire suivant qu'elles peuvent se montrer séparément ou non. Notamment, la confluence ne peut être montrée séparément, puisque la réunion de deux relations confluentes ne l'est pas forcément. En revanche, l'auto-réduction se montre pour chacune des composantes.

4.6.2 Formes normales de tête

Les sortes et les produits sont des formes normales de tête pour β et δ. En fait, il s'agit d'une conséquence des champs `ncts_hn_sort` et `ncts_hn_prod` que nous devons pourvoir.

Algorithme 4.70 (`beta_delta_whnf`) *La règle $\beta\delta$ possède un algorithme de mise en forme normale de tête.*

Preuve L'algorithme simple utilisé n'est pas très efficace, car il δ-normalise systématiquement les termes. Il serait préférable de calculer la β-forme normale de tête faible, et n'expanser les constantes de tête que lorsque cette stratégie échoue. ■

4.6.3 Confluence

Pour montrer la confluence on utilise la réduction parallèle, dont la fermeture réflexive transitive est équivalente à celle de la $\beta\delta$-réduction habituelle. Il suffit d'adapter la preuve de Tait Martin-Löf. C'est pour cela que nous avons formalisé une version abstraite de ce genre de preuves (théorème 4).

Définition 4.71 (prédicat `bd_par_red1`) *On utilise la $\beta\delta$-réduction parallèle définie figure 4.8.*

$$(\textsc{Step-}\beta)\ \frac{\Gamma\,[T] \vdash M \to_{\beta\delta_{/\!/}} M' \qquad \Gamma \vdash N \to_{\beta\delta_{/\!/}} N'}{\Gamma \vdash (\lambda T.\, M\ N) \to_{\beta\delta_{/\!/}} M'\{0\backslash N'\}}$$

$$(\textsc{Step-}\delta)\ \frac{\Gamma(n) = [M : T] \qquad \Gamma \vdash \uparrow^{n+1} M \to_{\beta\delta_{/\!/}} M'}{\Gamma \vdash \natural n \to_{\beta\delta_{/\!/}} M'}$$

$$(\textsc{Srt})\ \frac{}{\Gamma \vdash s \to_{\beta\delta_{/\!/}} s}\ (s \in \textsc{Sort}) \qquad (\textsc{Rel})\ \frac{}{\Gamma \vdash \natural n \to_{\beta\delta_{/\!/}} \natural n}$$

$$(\textsc{Abs})\ \frac{\Gamma \vdash M \to_{\beta\delta_{/\!/}} M' \qquad \Gamma\,[M'] \vdash N \to_{\beta\delta_{/\!/}} N'}{\Gamma \vdash \lambda M.\, N \to_{\beta\delta_{/\!/}} \lambda M'.\, N'}$$

$$(\textsc{App})\ \frac{\Gamma \vdash M \to_{\beta\delta_{/\!/}} M' \qquad \Gamma \vdash N \to_{\beta\delta_{/\!/}} N'}{\Gamma \vdash (M\ N) \to_{\beta\delta_{/\!/}} (M'\ N')}$$

$$(\textsc{Prd})\ \frac{\Gamma \vdash M \to_{\beta\delta_{/\!/}} M' \qquad \Gamma\,[M'] \vdash N \to_{\beta\delta_{/\!/}} N'}{\Gamma \vdash \Pi\, M.\, N \to_{\beta\delta_{/\!/}} \Pi\, M'.\, N'}$$

FIG. 4.8: Définition de la $\beta\delta$-réduction parallèle

On montre facilement que sa fermeture réflexive transitive est équivalente à $\beta\delta$.

Lemme 4.72 (`bd_par_red1_subst_rec`) *La $\beta\delta$-réduction parallèle est préservée par substitution, à gauche comme à droite.*

$$\frac{\beta\delta \vdash A \to_{\Gamma_{/\!/}} B \qquad \Gamma\,[T] \vdash M \to_{\beta\delta_{/\!/}} N}{\Gamma \vdash M\{0\backslash A\} \to_{\beta\delta_{/\!/}} N\{0\backslash B\}}$$

Preuve Comme d'habitude, il faut généraliser la proposition aux cas où la substitution se fait à un niveau quelconque.

$$\frac{\beta\delta \vdash A \to_{\Gamma_{/\!/}} B \qquad \Gamma\delta\Delta \vdash M \to_{\beta\delta_{/\!/}} N \qquad A \in \mathsf{ValOk}(\delta)}{\Gamma(\Delta\,\{A\}) \vdash M\{|\Delta|\backslash A\} \to_{\beta\delta_{/\!/}} N\{|\Delta|\backslash B\}}$$

■

Lemme 4.73 (`confluence_bd_par_red1`) *La $\beta\delta$-réduction parallèle est fortement confluente.*

Preuve Il suffit d'énumérer tous les cas. La preuve est relativement longue car elle comporte de nombreux cas, mais ils sont tous simples, car il n'y a pas de paires critiques entre β et δ. ■

Théorème 6 (`church_rosser_beta_delta`) *La règle $\beta\delta$ est confluente.*

Preuve Il suffit d'appliquer le théorème 4, page 112. ■

Lemme 4.74 *La règle $\beta\delta$ est décidable sur les termes fortement normalisables.*

Lemme 4.75 *Nous avons prouvé la confluence de $\beta\delta$, ainsi que l'existence d'un algorithme de mise en forme normale de tête. Le lemme 4.52 permet de construire un algorithme de conversion pour $\beta\delta$.*

4.6.4 Auto-réduction

Nous dissocions le fait qu'une règle soit correcte du fait qu'elle soit incluse dans le sous-typage. Par exemple, l'η-réduction est correcte (au sommet du terme, grâce à la contravariance), mais on ne la considère pas. Cependant, à partir du moment où on veut que la fermeture congruente soit correcte, il faut que la réduction soit incluse dans le sous-typage (lemme 4.45).

Les résultats suivants sont assez faciles à démontrer. Il suffit de faire la preuve pour une réduction au sommet du terme, car nous pourrons utiliser le théorème qui prouve que l'auto-réduction passe au contexte.

Nous considérons un PTS quelconque $\langle$Axiom; Rule; $\leq\rangle$.

Lemme 4.76 (`beta_sound`) *La règle β vérifie la propriété d'auto-réduction si le sous-typage du PTS $\leq$ considéré vérifie la propriété d'inversion des produits.*

$$\Gamma \vdash \Pi\, A.\, B \leq \Pi\, A'.\, B' \;\Rightarrow\; \Gamma \vdash A' \leq A \;\wedge\; \Gamma\,[A'] \vdash B \leq B'$$

Preuve Supposons $\Gamma \vdash (\lambda A.\, M\ N) : T$. En appliquant deux fois les lemmes d'inversion, nous en déduisons qu'il existe une sorte s et trois termes B, A' et B' tels que

- $\Gamma \vdash B'\{0\backslash N\} \leq T$
- $\Gamma \vdash N : A'$
- $\Gamma \vdash \lambda A.\, M : \Pi\, A'.\, B'$
- $\Gamma \vdash \Pi\, A.\, B \leq \Pi\, A'.\, B'$
- $\Gamma\,[A] \vdash M : B$
- $\Gamma \vdash \Pi\, A.\, B : s$

Nous dérivons successivement :

- $\Gamma \vdash A' \leq A$ et $\Gamma\,[A'] \vdash B \leq B'$ par inversion du produit,
- $\Gamma \vdash N : A$ par la règle de conversion,
- $\Gamma \vdash M\{0\backslash N\} : B\{0\backslash N\}$ par le lemme de substitution,
- $\Gamma \vdash M\{0\backslash N\} : T$ par la règle de conversion, $\Gamma \vdash B\{0\backslash N\} \leq B'\{0\backslash N\}$ étant obtenu par la substitutivité du sous-typage.

■

Le schéma des preuves d'auto-réduction est toujours le même : on suppose qu'une instance du membre gauche de la règle de réduction est typé par T ; on applique le lemme d'inversion sur cette hypothèse, et l'on reconstruit le jugement de typage du membre droit.

Lemme 4.77 (`delta_sound`) *La règle δ vérifie la propriété d'auto-réduction, sans aucune condition.*

Donc la règle $\beta\delta$ vérifie l'auto-réduction modulo la condition d'inversion du produit. Or, le lemme `inv_prod` permet de le prouver à partir du moment où notre relation $\beta\delta$ est confluente et telle que les produits sont des formes normales pour $\beta\delta$ (au sommet du terme), ce que nous avons déjà prouvé.

4.7 Hiérarchies de sortes

Il ne nous reste plus qu'à définir le système de sortes que nous souhaitons employer. Nous en présenterons deux, chacun ayant été implanté dans Coq. Le changement a eu lieu lors du passage de la version 5 à la version 6, sans que beaucoup d'utilisateurs n'en soient gêné. La modularité de notre développement est telle qu'elle permet de passer de l'un à l'autre aussi facilement que les utilisateurs.

4.7.1 Calcul des Constructions Étendu, Coq version 5.10

La hiérarchie de sortes utilisée dans la version V5.10 de Coq s'inspire de celle du Calcul des Constructions Étendu (ECC) défini par Luo dans sa thèse [38]. Comme dans le Calcul des Constructions, nous disposons d'une sorte imprédicative Prop. La différence est qu'au dessus de cette sorte, nous avons désormais une infinité de sortes Type, indexées par un entier. Ces sortes sont prédicatives, sinon il est possible de coder la paradoxe de Girard-Coquand [12], qui est une optimisation du paradoxe de Burali-Forti.

La seule différence entre les sortes de Coq V5.10 et ECC est que l'on a deux hiérarchies de sortes : l'une calculatoire et l'autre logique. Ainsi, Prop donne lieu a deux sortes Prop^- et Prop^+, et $\mathrm{Type}(i)$ se scinde en $\mathrm{Type}(-,i)$ et $\mathrm{Type}(+,i)$.

Définition 4.78 (type `calc,srt_ecc`) *Les sortes de la version 5.10 de* Coq *sont :*

$$\textsc{Sort}^{\mathrm{ECC}} \quad := \quad \mathrm{Prop}^{\epsilon} \mid \mathrm{Type}(\epsilon, n)$$

avec $\epsilon \in \{+,-\}$ *et* $n \in \mathbb{N}$.

Définition 4.79 (prédicat `univ_ecc`) *La relation de cumulativité ne concerne que les sortes prédicatives :*

$$\mathrm{Type}(\epsilon, i) \prec^{\mathrm{ECC}} \mathrm{Type}(\epsilon, j)$$

pour i *et* j *tels que* $i \leq j$.

Définition 4.80 (prédicat `axiom_ecc`) *Les règles de typage des sortes de la version 5.10 de* Coq *sont :*

$$(\mathrm{Prop}^{\epsilon}, \mathrm{Type}(\epsilon, i)) \in \textsc{Axiom}^{\mathrm{ECC}} \qquad (\mathrm{Type}(\epsilon, i), \mathrm{Type}(\epsilon, j)) \in \textsc{Axiom}^{\mathrm{ECC}}$$

pour i *et* j *tels que* $i < j$.

Définition 4.81 (prédicat `rules_ecc`) *Les règles de formation des produits de la version 5.10 de* Coq *sont :*

$$(\mathrm{Prop}^{\epsilon}, s, s) \in \textsc{Rule}^{\mathrm{ECC}} \qquad (s, \mathrm{Prop}^{\epsilon}, \mathrm{Prop}^{\epsilon}) \in \textsc{Rule}^{\mathrm{ECC}}$$
$$(\mathrm{Type}(\epsilon_1, i), \mathrm{Type}(\epsilon_2, j), \mathrm{Type}(\epsilon_2, k)) \in \textsc{Rule}^{\mathrm{ECC}}$$

pour i, j *et* k *tels que* $i \leq k$ *et* $j \leq k$.

Cette définition montre que les sortes Prop^{ϵ} sont imprédicatives, alors que les $\mathrm{Type}(\epsilon, i)$ sont prédicatives.

Toutes ces relations sont décidables et satisfont les conditions requises dans la section 4.5.2.

Une définition alternative, que l'on rencontre parfois dans la littérature, serait :

$$(\mathrm{Prop}^{\epsilon}, \mathrm{Type}(\epsilon, 0)) \in \textsc{Axiom}^{\mathrm{ECC}} \qquad (\mathrm{Type}(\epsilon, i), \mathrm{Type}(\epsilon, i+1)) \in \textsc{Axiom}^{\mathrm{ECC}}$$
$$(\mathrm{Prop}^{\epsilon}, s, s) \in \textsc{Rule}^{\mathrm{ECC}} \qquad (s, \mathrm{Prop}^{\epsilon}, \mathrm{Prop}^{\epsilon}) \in \textsc{Rule}^{\mathrm{ECC}}$$
$$(\mathrm{Type}(\epsilon_1, i), \mathrm{Type}(\epsilon_2, i), \mathrm{Type}(\epsilon_2, i)) \in \textsc{Rule}^{\mathrm{ECC}}$$

Compte tenu de la cumulativité, ces deux définitions donnent lieu aux mêmes jugements de typage. L'avantage de la première est qu'elle rend explicite toutes les façons de typer les sortes et tous les produits que l'on peut effectivement former (elle est *complétée* par la relation de cumulativité, voir page 100).

4.7.2 Coq version 6.2

Dans la version 6.2, Les hiérarchies Type$(-,n)$ et Type$(+,n)$ ont été fusionnées. Ainsi, Prop$^-$ et Prop$^+$ ont tous deux le même type Type(0). Enfin, la cumulativité est étendue puisque les deux sortes Prop$^\epsilon$ sont incluses dans Type(0).

Définition 4.82 (type `srt_v6`) *Les sortes de la version 6.2 de* Coq *sont :*

$$\textsc{Sort}^{\text{V6}} \quad := \quad \text{Prop}^\epsilon \mid \text{Type}(n)$$

avec les mêmes conventions que dans la section précédente. Pour reprendre les notations de Coq*,* Prop *désignera* Prop$^-$ *et* Set *sera une abbréviation pour* Prop$^+$*.*

Définition 4.83 (prédicat `univ_v6`) *La relation de cumulativité de la version 6.2 de* Coq *est :*

$$\text{Prop}^\epsilon \prec^{\text{V6}} \text{Type}(i) \qquad \text{Type}(i) \prec^{\text{V6}} \text{Type}(j)$$

pour i et j tels que $i \leq j$.

Définition 4.84 (prédicat `axiom_v6`) *Les règles de typage des sortes de la version 6.2 de* Coq *sont :*

$$(\text{Prop}^\epsilon, \text{Type}(i)) \in \textsc{Axiom}^{\text{V6}} \qquad (\text{Type}(i), \text{Type}(j)) \in \textsc{Axiom}^{\text{V6}}$$

pour i et j tels que $i \leq j$.

Définition 4.85 (prédicat `rules_v6`) *Les règles de formation des produits de la version 6.2 de* Coq *sont :*

$$(\text{Prop}^\epsilon, s, s) \in \textsc{Rule}^{\text{V6}} \qquad (s, \text{Prop}^\epsilon, \text{Prop}^\epsilon) \in \textsc{Rule}^{\text{V6}}$$
$$(\text{Type}(i), \text{Type}(j), \text{Type}(k)) \in \textsc{Rule}^{\text{V6}}$$

pour i, j et k tels que $i \leq k$ et $j \leq k$.

Ce système de sortes vérifie lui aussi toutes les propriétés attendues.

4.8 Construction du noyau

Il ne reste plus qu'à assembler les différents modules pour former le noyau que l'on peut extraire. Nous allons décrire cette construction pas à pas.

Définition 4.86 (`ecc`, `v6`) *Nous définissons les deux CTS* ECC *et* V6 *à l'aide des définitions des sections précédentes :*

$$\text{ECC} \quad \stackrel{\text{def}}{\equiv} \quad \langle \textsc{Sort}^{\text{ECC}}, \textsc{Axiom}^{\text{ECC}}, \textsc{Rule}^{\text{ECC}}, \beta\delta, \prec^{\text{ECC}} \rangle$$
$$\text{V6} \quad \stackrel{\text{def}}{\equiv} \quad \langle \textsc{Sort}^{\text{V6}}, \textsc{Axiom}^{\text{V6}}, \textsc{Rule}^{\text{V6}}, \beta\delta, \prec^{\text{V6}} \rangle$$

Comme tous les CTS s'injectent dans les PTS, nous bénéficions désormais de tous les résultats métathéoriques de la section 4.3.2 (lemmes d'inversion, de substitution, correction des types, etc.). Dans les sections précédentes, nous avons montré que ces CTS remplissaient toutes les conditions de CtsSubDec, ce qui nous permet de construire l'algorithme de décidabilité du sous-typage. Nous pouvons aussi prouver l'auto-réduction de $\beta\delta$, puisque c'est la réunion de deux règles qui ont cette propriété (pour β, il suffit d'utiliser le lemme 4.65).

À ce point, nous aurions assez de résultats métathéoriques pour nous lancer dans la preuve de normalisation forte de nos systèmes de type. La preuve de normalisation forte de ECC a été faite par Luo dans sa thèse.

On avait fait formellement la preuve pour le Calcul des Constructions, mais maintenant, il se peut que nous tombions sous le coup du deuxième théorème d'incomplétude de Gödel, et qu'il soit impossible de faire la preuve de normalisation forte de ECC dans le Calcul des Constructions Inductives. Nous admettrons donc ces résultats. Nous avons déjà discuté ce point dans l'introduction.

Axiome 1 (`ecc_normalise, v6_normalise`) *Les systèmes de types* ECC *et* V6 *(version 5.10 et 6.2 de* Coq*) sont fortement normalisants.*

Ceci nous permet de compléter la signature CtsNormSound, et il ne reste plus qu'à employer le théorème 5 pour former le noyau.

Théorème 7 (`ecc_algorithms, v6_algorithms`) *Les systèmes de types* ECC *et* V6 *ont des jugements de typage décidables :*

$$\text{ECC} \in \textsc{PtsTc} \qquad \text{V6} \in \textsc{PtsTc}$$

D'après la description faite, on voit que la construction du noyau ne diffère que dans le choix initial des ingrédients qui constituent le CTS. En fait, nous avons aussi produit une version de V6 sans la δ-réduction. Les étapes sont les mêmes, sauf pour montrer l'auto-réduction, où l'on évite simplement de montrer l'auto-réduction de δ. Les fichiers d'assemblage des différents modules sont courts (moins de 150 lignes à chaque fois). Reconstruire une variante d'un noyau est donc très facile. Par exemple, rédiger le fichier d'assemblage du noyau sans δ se fait en quelques minutes.

4.8.1 Extraction

Avant extraction, l'algorithme de typage est en fait une fonction qui, étant donné un terme, retourne en cas de succès son type, ainsi qu'une preuve qu'il est bien typé, i.e. la dérivation de typage complète. Potentiellement, nous pourrions calculer la dérivation, mais nous ne le ferons pas car cela ruinerait l'efficacité du programme. En effet, la dérivation de $\Gamma \vdash$ est au moins de taille exponentielle par rapport à la taille de Γ (la dérivation de $\Gamma\,[T]$ contient au moins autant de dérivations de $\Gamma \vdash$ que T a de feuilles).

Heureusement, nous ne souhaitons pas garder les dérivations, mais simplement les termes-preuves. L'extraction permet d'enlever tout ce qui concerne la construction de la dérivation, puisque le prédicat `typ` a été défini dans Prop.

4.8.2 Réalisation d'un système de preuves

Il resterait à incorporer les messages d'erreur (section 2.4) pour en faire un noyau réellement utilisable. Cela a été fait formellement, mais n'a pas été présenté ici pour ne pas alourdir la présentation.

Nous avons programmé (sans vérification formelle) une petite interface permettant l'analyse grammaticale de termes, ainsi qu'un interpréteur de commandes similaire à celui formalisé dans la section 2.4.

On a pu formaliser le lemme de Newman dans ce système. Le système étant très rudimentaire, on n'a pas développé la preuve dans ce système, mais en Coq ! Une fois la preuve faite en Coq, il n'y a plus qu'à afficher le contexte sous une forme que comprend notre vérificateur. Comme nous avons choisi une syntaxe compatible avec celle de Coq, il suffit

d'afficher le terme preuve à la fin de chaque preuve. Ce script peut être revérifié par notre système.

On a fait cette transformation de manière totalement manuelle, mais il est envisageable de le faire automatiquement. Évidemment, le système logique n'étant pas aussi puissant que celui de Coq, principalement parce que nous n'avons pas formalisé les types inductifs. Seuls les développements n'utilisant pas les types inductifs seront acceptés.

4.8.3 Efficacité

Le code extrait est raisonnablement efficace. Il s'avère légèrement plus rapide que Coq sur des exemples de taille moyenne, comme la preuve du lemme de Newman (fichier de 6Ko). Ce n'est, tout compte fait, pas étonnant, car le code extrait correspond à ce qu'on aurait écrit à la main, surtout si l'on utilise la tactique `Program`. Les différences sont que nous avons dû programmer de manière totalement applicative, ce qui n'est en fait pas un handicap du point de vue efficacité. Aussi, pour ce coup d'essai, les algorithmes employés sont vraiment peu raffinés.

En poursuivant l'analyse, on se rend compte qu'en fait, la δ-réduction est très mal traitée dans notre système. En particulier, le test de conversion expanse systématiquement les définitions, ce qui ruine complètement les performances. Pire, ce handicap peut croître très rapidement avec la taille des développements (au moins de manière exponentielle). La gestion de l'expansion des définitions est un problème récurrent dans les systèmes de preuves. Cette situation est probablement due au fait que la δ-réduction semble tellement anodine d'un point de vue théorique, que peu d'effort est fait pour l'étudier sérieusement.

Pour évaluer l'effet de la δ-réduction, nous avons reconstruit le système muni uniquement de la δ-réduction, ce qui est très simple à faire grâce à la modularité de nos preuves. Pour être acceptée, la preuve du lemme de Newman doit être modifiée (à l'aide de Coq, bien sûr) de manière à rendre explicite dans le terme preuve les expansions de constantes nécessaires. Avec ces modifications, notre système extrait est quatre fois plus rapide que Coq, ce qui met bien en évidence à quel point l'implantation de l'algorithme de conversion est sensible. Il serait sûrement très bénéfique de poursuivre la formalisation du chapitre 3 afin de l'intégrer à notre système.

4.8.4 Autres conclusions et perspectives

Lorsque l'on a bien compris la sémantique de Heyting, il est facile de faire les preuves qui correspondent à l'algorithme qu'on a en tête, même sans employer la tactique `Program`. Réciproquement, nous avons à quelques occasions expliqué certains résultats logiques (par exemple les lemmes d'inversion du jugement de typage) en termes calculatoires. Il nous est apparu que ce genre d'explications, qui sont à la base de l'isomorphisme de Curry-Howard, étaient susceptibles de nous aider à mieux concevoir les preuves.

A posteriori, nous aurions pu regrouper les conditions à satisfaire différemment : d'un côté les propriétés portant sur la hiérarchie de sortes et celles portant sur la règle de réduction, la normalisation forte ayant un statut particulier.

Le bilan de ce développement est clairement encourageant, et nous donne suffisamment de confiance pour nous lancer dans la formalisation du Calcul des Constructions Inductives (il "suffit" pour cela de rajouter les types inductifs), avec comme objectif ultime le *bootstrap* de ce système. Une solution serait de reprendre le développement avec exactement les même idées sauf que l'on définirait des PTS avec types inductifs. Pour briser la monotonie, nous préférerons une présentation plus générale, en concevant des systèmes de types dans lesquels nous pouvons rajouter des opérateurs à l'aide d'une signature.

Chapitre 5

PTS avec opérateurs

5.1 Opérateurs

Notre objectif est maintenant de formaliser un système de types ayant une puissance équivalente à celui de Coq. La tâche la plus complexe qu'il reste à accomplir est la formalisation des types inductifs. Ceux-ci ont la réputation d'être assez complexes, et leur présentation est souvent fastidieuse. De plus, l'ajout de cette facilité a été incorporée relativement récemment à la Théorie des Types et le formalisme même évolue d'année en année.

Reprendre la formalisation de CCI avec la même méthode que celle décrite dans le chapitre précédent, simplement en ajoutant les constructeurs de termes nécessaires ferait reposer un bon nombre de résultats métathéoriques sur une base mouvante.

La solution que nous apporterons est d'étudier des systèmes de types dans lesquels le type des termes est en quelque sorte extensible.

Une solution serait de rajouter dans la syntaxe des termes un simple constructeur représentant des constantes que l'on rajoute au calcul. Martin-Löf [39] a proposé un système ouvert dans lequel nous avons la possibilité d'introduire de nouvelles constantes. Le comportement de ces constantes serait défini dans une *signature*, spécifiant notamment le type de la constante, mais aussi les règles de réductions associées à cette constante. Cette signature pourrait être étendue par l'utilisateur, modulo quelques vérifications pour s'assurer de la cohérence de l'extension proposée.

Par exemple, pour l'égalité, nous pourrions rajouter trois constantes eq, refl et subst dont les types seraient :

$$\begin{array}{rcl} \text{eq} & : & \Pi A{:}\text{Set}.\, A \to A \to \text{Prop} \\ \text{refl} & : & \Pi A{:}\text{Set}.\, \Pi x{:}A.\, (\text{eq}\ A\ x\ x) \\ \text{subst} & : & \Pi A{:}\text{Set}.\, \Pi x{:}A.\, \Pi P{:}(A \to \text{Prop}).\, (P\ x) \to \Pi y{:}A.\, (\text{eq}\ A\ x\ y) \to (P\ y) \end{array}$$

La constante subst correspond à l'égalité de Leibniz, qui permet de remplacer x par y lorsque l'on a prouvé l'égalité de x et de y. Enfin, nous introduirions la règle de réduction :

$$(\text{subst}\ A\ x\ P\ H\ x\ (\text{refl}\ A\ x)) \to H$$

Le problème d'une telle approche est que les constantes possèdent un bon nombre d'arguments que nous aimerions ne pas faire figurer explicitement dans les preuves. Dans l'exemple

ci-dessus, le type de A peut en général être inféré[1]. Cela permettrait à nos constantes d'être d'arité respectives 2, 1 et 3 (dans le cas de subst, on ne peut se passer du prédicat puisque le filtrage d'ordre supérieur est indécidable), au lieu de respectivement 3, 2 et 6.

Pour éviter cette abondance d'arguments, nous allons considérer des constantes dont le type est trop complexe pour pouvoir toujours être exprimé par un type du système logique. On associera à ces constantes un *schéma de type*. Il sera possible d'associer un type du système uniquement à la constante une fois appliquée à un certain nombre d'arguments, spécifiés par le schéma de type. C'est pourquoi nous les appelerons plutôt *opérateurs*.

Un schéma de type est tout simplement une relation qui associe aux valeurs et types des arguments appliqués à l'opérateur, le type de l'objet ainsi formé. Évidemment, pour que le système ait de bonnes propriétés métathéoriques, il faudra restreindre les schémas autorisés d'une certaine manière, ce que nous verrons plus tard.

Pour reprendre l'exemple de l'égalité, nous associerons à eq le schéma suivant :

$$(M : A;\ N : A) \to \mathrm{Prop}$$

ce qui signifie que le schéma de type de eq est la relation qui à tout quadruplet qui est filtré par le membre gauche ci-dessus associe le membre droit. Les termes M et N représentent les valeurs des arguments de eq (qui sera donc d'arité deux), et ils sont accompagnés de leur type.

De même, les autres constantes auront les schémas suivants :

$$\begin{array}{rcl} \mathrm{refl} & : & (M : A) \to (\mathrm{eq}\ M\ M) \\ \mathrm{subst} & : & (P : (A \to \mathrm{Prop});\ H : (P\ x);\ M : (\mathrm{eq}\ x\ y)) \to (P\ y) \end{array}$$

et la règle de réduction associée sera tout simplement :

$$(\mathrm{subst}\ P\ H\ (\mathrm{refl}\ x)) \to H$$

On se rend compte alors que le typage de l'application peut s'exprimer à l'aide de nos schémas de types. On peut considérer l'application comme un opérateur @ non primitif qui prend deux arguments M et N, l'un de type $\Pi x{:}A.\,B$ et l'autre de type A, le type de cette instance de l'opérateur est alors $B\{0\backslash N\}$. Pour cela, on associe à l'opérateur @ le schéma de type suivant :

$$@ : (M : \Pi x{:}A.\,B;\ N : A) \to B\{0\backslash N\}.$$

À cet opérateur, on associe la règle de réduction

$$@(\lambda x{:}A.\,M,\ N) \to_\beta M\{x\backslash N\}$$

ce qui achève de définir les PTS dans notre cadre. Noter que l'on n'a pas besoin de donner les types A et B dans l'opérateur, car ces types peuvent être inférés[2]. On voit donc que les opérateurs sont un peu plus commodes que de simplement offrir la possibilité de rajouter des constantes avec un type exprimé uniquement à l'aide des produits du systèmes : @ devrait avoir le type

$$\Pi A{:}s_1.\,\Pi B{:}A \Rightarrow s_2.\,(\Pi x{:}A.\,(B\ x)) \to \Pi y{:}A.\,(B\ y)$$

et cela nous oblige à rajouter une telle constante pour chaque $(s_1, s_2, s_3) \in \textsc{Rule}$.

[1] La situation est un peu plus complexe avec les systèmes possédant du sous-typage. On pourra toujours inférer le type A si l'on dispose d'un algorithme décidant si deux types admettent un sur-type commun.

[2] D'ailleurs, dans la présentation usuelle de l'application comme constructeur de terme, les types A et B n'apparaissent pas explicitement.

Dans la plupart des cas, la règle associée à un opérateur consiste à typer les sous-termes de cet opérateurs, avec éventuellement des conditions annexes, mais qui ne peuvent invoquer le typage. Les schémas de type capturent bien ces cas. La règle de l'abstraction ne vérifie pas cette condition puisqu'une prémisse demande que le produit soit bien formé. Ceci fait que l'abstraction est particulièrement délicate à typer, et nous oblige à considérer des sous-classes de PTS comme les PTS *full* ou *semi-full*, qui nous dispensent de typer effectivement le produit.

Le fait que les constantes soient obligatoirement totalement appliquées n'est pas vraiment un problème, puisqu'il suffit en général d'η-allonger les termes (cela peut échouer lorsque le type de l'opérateur ne peut être exprimé dans le système de type, mais c'est ce que nous voulions pour ne pas avoir trop d'arguments).

5.1.1 Méthodologie

On reprend la même méthodologie que pour les PTS. Ce développement sera encore plus court que celui des PTS, car il est encore plus général, ce qui laissera plus de travail au moment de l'instantiation.

Notre but est de produire un noyau de vérification de type pour notre classe de systèmes, et cela aura pour effet de dégager des conditions sur les paramètres (notamment le sous-typage, mais ce qui est nouveau, c'est la signature).

Ce développement sera moins modulaire que les autres, mais c'est uniquement par manque de temps (il pourrait l'être autant). Seule la partie métathéorie simple, correspondant à la section 4.3, sera totalement générique.

Un certain nombre de définitions n'ont pratiquement pas changé par rapport au PTS. Dans ce cas, on ne redonnera pas la définition. Beaucoup de noms de lemmes et de types ont été réutilisés.

5.2 Syntaxe des termes

Comme d'habitude, la formalisation débute avec la définition des termes. Nous avons remarqué que l'absence de noms dans les lieurs était particulièrement gênante lorsque l'on avait à afficher des termes. Pour que l'utilisateur ne soit pas complètement désorienté, il est important de réutiliser les noms donnés par l'utilisateur, ou au moins des noms suffisamment proches[3]. Ainsi, nous continuons de raisonner avec des indices de de Bruijn, mais chaque lieur porte un nom qui sera utilisé comme préfixe du nom utilisé au moment de l'affichage. C'est uniquement à l'affichage que l'on résoudra les risques de capture.

Cette fois, les termes sont paramétrés par les types suivants :

- comme pour les PTS, un ensemble des sortes SORT, sur lequel l'égalité est décidable ;
- un ensemble non vide (nous noterons _ un élément particulier de cet ensemble) de noms de variables NAME, dont l'égalité est décidable ;
- un ensemble d'opérateurs OPER, dont l'égalité est décidable.

La première idée pour représenter le type des termes serait de remplacer la constructeur d'application par celui des opérateurs qui prendraient en argument une liste de termes. Comme pour les PTS, nous retrouverions les sortes, les variables, l'abstraction et le produit.

[3] Des renommages peuvent être nécessaires pour éviter des ambiguïtés. Par exemple, il serait incorrect d'afficher $\lambda x{:}T_1.\,\lambda x{:}T_2.\,(\natural 0\ \natural 1)$ de manière naïve : $\lambda x{:}T_1.\,\lambda x{:}T_2.\,(x\ x)$. Il faut renommer l'un des x.

```
Inductive term : Set :=
    Srt: sort -> term
  | Rel: nat -> term
  | Cst: oper -> (list term) -> term
  | Prd: name -> term -> term -> term
  | Lam: name -> term -> term -> term.
```

Hélas, les types inductifs de Coq ne permettent pas d'écrire des fonctions récursives dont les appels récursifs se font sur les éléments de la liste argument de `Cst`. L'astuce classique pour contourner ce problème est d'expanser la définition des listes, ce qui donnerait lieu au type inductif suivant :

```
Mutual Inductive term : Set :=
    Srt: sort -> term
  | Rel: nat -> term
  | Cst: oper -> lterm -> term
  | Prd: name -> term -> term -> term
  | Lam: name -> term -> term -> term
with lterm : Set :=
    Lnil: lterm
  | Lcons: term -> lterm -> lterm.
```

L'inconvénient de ce choix est qu'il complique les raisonnements par récurrence car il faut fournir un énoncé équivalent sur les listes de termes. Il faut aussi dupliquer les jugements de typage de manière à typer les listes de termes par des listes de types. Il nous a semblé plus simple de confondre termes et listes de termes, simplement en introduisant la paire (qui correspond au *cons* des listes) et les opérateurs d'arité zéro (qui sera utilisé pour le *nil* des listes). Le produit cartésien sert de *cons* pour les listes de types. Ce qui fait que le schéma de type $(M_1 : T_1; \ldots; M_n : T_n) \to U$ sera noté $((M_1, \ldots, M_n), T_1 * \ldots * T_n) \to U$. La déclaration en Coq du type des termes est donc :

```
Inductive term: Set :=
    Srt: sort -> term
  | Rel: nat -> term
  | Cst0: oper -> term
  | Cst1: oper -> term -> term
  | Prd: name -> term -> term -> term
  | Lam: name -> term -> term -> term
  | Sum: term -> term -> term
  | Pair: term -> term -> term.
```

Définition 5.1 (type `term`) *Le type des termes est défini selon la grammaire suivante :*

$$T_1, T_2 : \textsc{Term} \quad := \quad s \mid \natural n \mid c \mid c(T_1) \mid \Pi x{:}T_1.\, T_2 \mid \lambda x{:}T_1.\, T_2 \mid T_1 * T_2 \mid (T_1,\, T_2)$$

où $s \in \textsc{Sort}$, $x \in \textsc{Name}$, $c \in \textsc{Oper}$, *et* $n \in \mathbb{N}$.

Notre système ne comporte de manière primitive qu'un lieur et les paires. Tous les autres opérateurs sont construits en utilisant ce lieur. Ce lieur sert à typer des termes dans un contexte différent.

L'utilité des paires sera essentiellement de permettre de représenter les listes d'arguments. Grâce aux paires, il n'est pas nécessaire d'avoir des opérateurs d'arité quelconque : il suffit d'avoir des d'opérateurs d'arité zéro et un. Un opérateur d'arité deux (comme l'application) prendra comme argument une paire. Le type des paires formées d'un objet de

type A et d'un autre de type B sera noté $A * B$, et la paire formée à partir de M et N sera notée (M, N). Cette manière de procéder est totalement symétrique au cas du produit et de l'abstraction : on introduit une opération sur les types, ainsi qu'un constructeur pour habiter les types de cette espèce. Le couple formé par le produit et l'abstraction introduit la notion de fonction, tout comme le couple produit cartésien et paire introduit une notion rudimentaire de structure, en donnant la possibilité d'agréger des objets. Aussi, les projections, qui permettent d'accéder aux composantes de la paire, ne sont pas primitives dans le système, mais définies par l'intermédiaire des opérateurs, comme pour l'application.

Définition 5.2 (type `decl,env`) *Les déclarations sont soit des variables, soit des définitions. Les déclarations étant des lieurs, elles comportent un nom.*

$$\begin{array}{rcl} \textsc{Decl} & := & [x : T] \mid [x{=}M : T] \\ \textsc{Ctx} & := & [] \mid \Gamma\delta \end{array}$$

avec $\Gamma \in \textsc{Ctx}$, $\delta \in \textsc{Decl}$, $x \in \textsc{Name}$ *et* $M, T \in \textsc{Term}$.

Pour toute déclaration δ, on note $\delta.\textsc{x}$ le nom qu'elle contient, et $\delta.\textsc{b}$ son éventuel corps, la notation $\delta.\textsc{b} = \bot$ signifiant l'absence de corps.

Les fonctions de relocation et de substitution sont définies comme pour les PTS. Il faut simplement préciser les liaisons de variable pour les nouvelles constructions : un opérateur ne lie aucune variable dans son argument ; le produit cartésien $A * B$ et la paire (M, N) ne lient pas de variables. Autrement dit, on étend les définition de la figure 4.1 avec les équations suivantes :

Relocation	Substitution
$\uparrow_k^n c = c$	$c\{k\backslash N\} = c$
$\uparrow_k^n c(M) = c(\uparrow_k^n M)$	$c(M)\{k\backslash N\} = c(M\{k\backslash N\})$
$\uparrow_k^n (A * B) = \uparrow_k^n A * \uparrow_k^n B$	$(A * B)\{k\backslash N\} = A\{k\backslash N\} * B\{k\backslash N\}$
$\uparrow_k^n (A,\, B) = (\uparrow_k^n A,\, \uparrow_k^n B)$	$(A,\, B)\{k\backslash N\} = (A\{k\backslash N\},\, B\{k\backslash N\})$

On peut démontrer exactement les mêmes propriétés algébriques concernant les relocations et substitutions. Nous aurons besoin d'un résultat supplémentaire, qui n'est pas une conséquence directe de ces propriétés.

Lemme 5.3 (`replace0`) *Si dans un terme M, on crée une variable $k{+}1$, et que l'on substitue la variable k par $\natural 0$ (qui dénote donc la variable nouvellement créée), on obtient M.*

$$(\uparrow_{k+1}^1 M)\{k\backslash\natural 0\} = M$$

Définition 5.4 (prédicat `free_db`) *L'ensemble des variables qui ne sont pas libres dans un terme M est noté $\mathsf{V}(M)$. Il s'agit de l'ensemble des entiers qui n'apparaissent pas dans le terme, et qui ne sont liés par aucun produit ou abstraction. Par abus de langage, on dit que la variable n* apparaît *dans M, si $n{+}m$ apparaît syntaxiquement sous m lieurs. Formellement, ce prédicat est défini à la Prolog :*

$$\frac{}{n \in \mathsf{V}(s)}(s \in \textsc{Sort}) \qquad \frac{n \neq k}{n \in \mathsf{V}(\natural k)} \qquad \frac{}{n \in \mathsf{V}(c)}(c \in \textsc{Oper}) \qquad \frac{n \in \mathsf{V}(M)}{n \in \mathsf{V}(c(M))}$$

$$\frac{n \in \mathsf{V}(T) \qquad n{+}1 \in \mathsf{V}(U)}{n \in \mathsf{V}(\Pi x{:}T.\, U)} \qquad \frac{n \in \mathsf{V}(T) \qquad n{+}1 \in \mathsf{V}(M)}{n \in \mathsf{V}(\lambda x{:}T.\, M)}$$

$$\frac{n \in \mathsf{V}(A) \qquad n \in \mathsf{V}(B)}{n \in \mathsf{V}(A * B)} \qquad \frac{n \in \mathsf{V}(M) \qquad n \in \mathsf{V}(N)}{n \in \mathsf{V}((M,\, N))}$$

Définition 5.5 (prédicat `closed`) *La "taille" d'un terme M est l'ensemble des indices de de Bruijn pour lesquelles aucune des indices supérieurs n'apparaît dans M :*

$$|M| \stackrel{\text{def}}{\equiv} \{n \mid \forall m \geq n.\, m \in \mathsf{V}(M)\}$$

La propriété essentielle de $|M|$ est que c'est l'ensemble des longueurs de contextes qui lient toutes les variables de M, ou bien le nombre d'abstraction qu'il faut rajouter pour obtenir un terme clos.

Formellement, on définit $|M|$ de manière similaire à $\mathsf{V}(M)$, à la Prolog, en remplaçant la prémisse $n \neq k$ par $n > k$. L'équivalence avec la définition ci-dessus est très facile à prouver. La raison est qu'avec un prédicat défini inductivement, on bénéficie des tactiques d'inversions qui permettent de prouver automatiquement que $n \in |\lambda x : T.\, M|$ implique $n \in |T|$ et $n+1 \in |M|$, propriété qui devrait être énoncée explicitement avec la définition informelle.

Intuitivement, on verrait plutôt $|M|$ comme l'élément minimal de l'ensemble défini ci-dessus, mais à aucun moment dans le développement il n'a été nécessaire d'introduire explicitement cette borne inférieure.

Lemme 5.6 (`closed_lift_rec`, `closed_subst_rec`) *Les relocations et substitutions à un rang au dela de la plus grande variable libre sont sans effet :*

$$k \in |M| \;\Rightarrow \uparrow_k^n M = M$$
$$k \in |M| \;\Rightarrow\; M\{k\backslash N\} = M$$

On définit quelques sous-types de l'ensemble des termes, en fonction du nombre de variables libres :

Définition 5.7 (types `cterm`, `cdecl`) *L'ensemble des termes M tels que $n \in |M|$ sera noté* TERM_n. *En particulier,* TERM_0 *est l'ensemble des clos. Enfin,* DECL_n *sera l'ensemble des déclarations qui ne comportent que des termes dans* TERM_n.

Les opérations de relocation et de substitution dans les contextes (InsInEnv et SubInEnv) se définissent comme pour les PTS, et possèdent les mêmes propriétés, car leur définition est indépendante de la structure des termes.

5.3 Règles de sous-typage

Comme pour les PTS, nous allons paramétrer le système par une relation de sous-typage, qui est une relation ternaire vérifiant quelques propriétés vis-à-vis de la relocation et de la substitution. Nous retrouvons Rlift sans aucun changement (voir page 96). La substitutivité est définie sous deux formes. L'une, faible, notée Rsubstwk est identique à celle des PTS.

Définition 5.8 (prédicat `R_subst_weak`) *La définition de la stabilité (faible) d'une relation ternaire est définie comme dans le chapitre 4 :*

$$\mathsf{Rsubstwk} \stackrel{\text{def}}{\equiv} \left\{ R \;\middle|\; \begin{array}{l} M \in \mathsf{ValOk}(\delta) \wedge R(\Gamma\delta(\Delta^+), A, B) \\ \quad\Rightarrow\; R^*(\Gamma(\Delta\,\{M\}), A\{|\Delta|\backslash M\}, B\{|\Delta|\backslash M\}) \end{array} \right\}$$

Nous introduirons aussi la substitutivité forte, Rsubst, qui a pour conséquence la version faible de substitutivité.

Définition 5.9 (prédicat `R_subst`) *La substitutivité forte ne fait pas appel à la fermeture réflexive transitive :*

$$\mathsf{Rsubst} \stackrel{\text{def}}{\equiv} \left\{ R \;\middle|\; \begin{array}{l} M \in \mathsf{ValOk}(\delta) \wedge R(\Gamma\delta(\Delta^+), A, B) \\ \quad \Rightarrow\ R(\Gamma(\Delta\,\{M\}), A\{|\Delta|\backslash M\}, B\{|\Delta|\backslash M\}) \end{array} \right\}$$

Ces deux notions de substitutivité se confondent sur les préordres. En revanche, lorsque l'on considère les règles de réductions élémentaires, ces deux définitions diffèrent. Notamment, la δ-réduction ne vérifie pas la substitutivité forte.

Il existait une troisième propriété requise pour former une règle de sous-typage : l'invariance du sous-typage par modifications des types figurant dans le contexte (Rstable). Nous raffinons cette condition en la paramétrant par une relation sur les déclarations (plus exactement, un sous-ensemble de $\textsc{Ctx} \times \textsc{Decl} \times \textsc{Decl}$) permettant de contrôler les modifications apportées au contexte.

Définition 5.10 (prédicat `R_stable`) *Soit S une relation d'inclusion entre déclarations. L'ensemble* $\mathsf{Rstable}_S$ *des relations ternaires invariantes par modification du contexte suivant S est défini par :*

$$\mathsf{Rstable}_S \stackrel{\text{def}}{\equiv} \{R \mid R(\Gamma, A, B) \ \wedge\ (\Gamma')\ S\ (\Gamma) \ \Rightarrow\ R(\Gamma', A, B)\}$$

où $(\Gamma')\ S\ (\Gamma)$ signifie que Γ et Γ' ne diffèrent que pour un certain certain rang n pour lequel on a $S(\Gamma_0, \Gamma(n), \Gamma'(n))$, avec Γ_0 le contexte commun de $\Gamma(n)$ et $\Gamma'(n)$.

Pour toute relation de sous-typage R, on appelle $\mathrm{ty}(R)$ la relation d'inclusion entre les déclarations qui n'affecte pas les valeurs.

Définition 5.11 (prédicat `rel_typ`) *La relation entre déclarations $\mathrm{ty}(R)$ permet d'appliquer R aux types des déclarations, sans affecter les valeurs. Les noms peuvent être modifiés.*

$$\mathrm{ty}(R)(\Gamma, \delta_1, \delta_2) \stackrel{\text{def}}{\equiv} R(\Gamma, \delta_{1.\mathrm{T}}, \delta_{2.\mathrm{T}}) \ \wedge\ \delta_{1.\mathrm{B}} = \delta_{2.\mathrm{B}}$$

Cela permet au sous-typage de dépendre des types présents dans le contexte, en tout cas, en ce qui concerne la métathéorie simple. En revanche, cela risque fortement de poser des problème de décision du sous-typage car l'algorithme qui consiste à comparer les formes normales ne convient plus, comme le montre l'exemple suivant.

Exemple 5.12 *Supposons que les deux termes à comparer ont comme formes normales $\Pi x{:}T.\,U$ et $\Pi y{:}T'.\,U'$ dans Γ, ce qui signifie que U est une forme normale dans $\Gamma\,[x : T]$ et U' dans $\Gamma\,[y : T']$. Avec la contravariance, on vérifie que T' est un sous-type de T, et que U est un sous-type de U' dans $\Gamma\,[y : T']$. Le problème est que U se retrouve dans un contexte plus petit que celui dans lequel il a été normalisé. Cela peut autoriser plus de réductions. L'algorithme de sous-typage consistant à comparer les formes normales ne fonctionne plus.*

Définition 5.13 (prédicat `decl_rel`) *Nous définissons* $\mathrm{dclsub}(R_1, R_2)$, *une autre relation d'inclusion entre déclarations : R_1 est une relation indiquant quelles valeurs on identifie, alors que R_2 est une relation d'inclusion entre types, par exemple le sous-typage.*

$$\mathrm{dclsub}(R_1, R_2)(\Gamma, \delta_1, \delta_2) \stackrel{\text{def}}{\equiv} R_2(\Gamma, \delta_{1.\mathrm{T}}, \delta_{2.\mathrm{T}}) \ \wedge\ (\delta_{2.\mathrm{B}} = \bot \vee R_1(\Gamma, \delta_{1.\mathrm{B}}, \delta_{2.\mathrm{B}}))$$

Les règles d'inférences suivantes définissent le même prédicat de manière plus lisible :

$$\frac{R_2(\Gamma, T_1, T_2)}{\mathrm{dclsub}(R_1, R_2)(\Gamma, [x{=}M_1 : T_1], [y : T_2])} \qquad \frac{R_1(\Gamma, M_1, M_2) \qquad R_2(\Gamma, T_1, T_2)}{\mathrm{dclsub}(R_1, R_2)(\Gamma, [x{=}M_1 : T_1], [y{=}M_2 : T_2])}$$

Il est évident que $\mathrm{ty}(R) \subseteq \mathrm{dclsub}(=, R)$ pour tout R. Cette propriété (lorsque instanciée par le sous-typage) assure que le sous-typage dans le contexte est contravariant, i.e. plus les types dans le contexte sont petits, plus on a d'informations sur la valeur que peut prendre la variable, plus on peut faire de réductions. Cela justifie le fait que `decl_rel` inclut $[x{=}M:T]$ dans $[y:T]$.

Définition 5.14 (prédicat `Subtyping_rule`) *Une règle de sous-typage est un préordre vérifiant la conjonction des trois propriétés ci-dessus :*

$$\textsc{SubRule} \stackrel{\text{def}}{\equiv} \left\{ R \in \mathsf{Rlift} \cap \mathsf{Rsubst} \;\middle|\; R \in \mathsf{Rstable}_{\mathrm{ty}(R)} \;\wedge\; R^* \subseteq R \right\}$$

5.4 Signature d'opérateurs

Le dernier paramètre, qui caractérise notre classe de systèmes de types, est la signature décrivant comment se typent les opérateurs d'arité zéro ou un. Elle est composée de deux relations Σ_0 et Σ_1 :

- $\Sigma_0 \in \mathcal{P}(\textsc{Oper} \times \textsc{Term})$ pour les opérateurs d'arité zéro. Si l'on a $(c, U) \in \Sigma_0$, cela signifie que l'opérateur constant c a pour type U.
- $\Sigma_1 \in \mathcal{P}(\textsc{Oper} \times \textsc{Ctx} \times \textsc{Term}^3)$ pour les opérateurs d'arité un. Si $(c, \Gamma, M, T, U) \in \Sigma_1$, cela signifie que dans le contexte Γ, on peut associer à l'opérateur c le schéma de type $(M, T) \to U$. La signature Σ_1 dépend du contexte.

En d'autres termes, on pourrait considérer typer les opérateurs à l'aide des règles de typages suivantes :

$$\frac{\Gamma \vdash \qquad (c, U) \in \Sigma_0}{\Gamma \vdash c : U} \qquad \frac{\Gamma \vdash M : T \qquad (c, \Gamma, M, T, U) \in \Sigma_1}{\Gamma \vdash c(M) : U}$$

Pour avoir de bonnes propriétés métathéoriques indépendamment de la signature, nous renforcerons ces règles en nous assurant que les types issus de la signature sont bien formés (tout comme nous vérifions la bonne formation des types dans la régle de conversion) :

$$(\textsc{Op}0)\ \frac{\Gamma \vdash \qquad [\_ : U] \vdash \qquad (c, U) \in \Sigma_0}{\Gamma \vdash c : U}$$

$$(\textsc{Op}1)\ \frac{\Gamma\,[\_ : T] \vdash \qquad \Gamma\,[\_ : U] \vdash \qquad \Gamma \vdash M : T \qquad (c, \Gamma, M, T, U) \in \Sigma_1}{\Gamma \vdash c(M) : U}$$

Cette hypothèse de confort nous permet de définir plus facilement la signature, i.e. sans trop se soucier du typage. L'inconvénient est que la décidabilité du typage sera moins directe. En effet, nous ne pourrons pas faire d'appel récursif pour vérifier le type de U (la fonction de typage fait une récurrence structurelle sur le terme à typer). Pour nous dispenser de cette vérification, nous prouverons un résultat métathéorique assurant que si T est bien formé (ce que nous pourrons vérifier grâce au résultat de correction des types), alors U le sera aussi. En d'autres termes, si les types donnés en "entrée" de la signature sont corrects, alors les types en "sortie" de la signature seront corrects.

Bien que la vérification ci-dessus nous donne une plus grande liberté, la signature ne peut pas dépendre arbitrairement de la valeur et du type de ses arguments. Comme pour la relation de sous-typage, il faut quelques conditions de régularité vis-à-vis de la relocation, de la substitution et de la réduction dans le contexte.

Définition 5.15 (prédicat `signature`) *Soit $\leq$ une relation ternaire. Une signature Σ_1 est dite invariante par rapport à $\leq$ si elle vérifie les conditions suivantes :*

$$
\begin{array}{ll}
\langle\ \texttt{sig\_member\_lift}: & \forall \Gamma_0, \Gamma, \Gamma', n, c, M, T, U. \\
& \quad \mathsf{InsInEnv}(\Gamma_0, \delta, n, \Gamma, \Gamma') \ \wedge\ (c, \Gamma, M, T, U) \in \Sigma_1 \\
& \quad\quad \Rightarrow\ (c, \Gamma', \uparrow^1_n M, \uparrow^1_n T, \uparrow^1_n U) \in \Sigma_1 \\
\texttt{sig\_member\_subst}: & \forall \Gamma_0, \delta, \Gamma, \Gamma', n, c, N, M, T, U. \\
& \quad \mathsf{SubInEnv}(\Gamma_0, \delta, N, n, \Gamma, \Gamma') \ \wedge\ (c, \Gamma, M, T, U) \in \Sigma_1 \\
& \quad\quad \Rightarrow\ (c, \Gamma', M\{n\backslash N\}, T\{n\backslash N\}, U\{n\backslash N\}) \in \Sigma_1 \\
\texttt{sig\_member\_stable}: & \forall \Gamma, \Gamma', c, M, T, U. \\
& \quad (\Gamma')\ \mathrm{ty}(\leq)\ (\Gamma) \ \wedge\ (c, \Gamma, M, T, U) \in \Sigma_1 \\
& \quad\quad \Rightarrow\ (c, \Gamma', M, T, U) \in \Sigma_1 \\
\rangle &
\end{array}
$$

On notera $\mathsf{SignInv}_{\leq}$ l'ensemble des signatures Σ_1 vérifiant les trois propriétés ci-dessus.

5.5 Définition des jugements de typage

5.5.1 Paramètres

Définition 5.16 (type `PTSO_spec`) *La spécification d'un PTS avec sous-typage et opérateurs est une structure regroupant six paramètres :*

$$
\begin{array}{lll}
\langle\ \textsc{Axiom}: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort}); & \text{(* Typage des sortes *)} \\
\textsc{Rule}: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort} \times \textsc{Sort}); & \text{(* Formation des produits *)} \\
\textsc{Pair}: & \mathcal{P}(\textsc{Sort} \times \textsc{Sort} \times \textsc{Sort}); & \text{(* Formation des paires *)} \\
\leq: & \textsc{SubRule}; & \text{(* Relation de sous-typage *)} \\
\Sigma_0: & \mathcal{P}(\textsc{Oper} \times \textsc{Term}); & \text{(* Typage des opérateurs constants *)} \\
\Sigma_1: & \mathsf{SignInv}_{\leq} & \text{(* Typage des opérateurs d'arité un *)} \\
\rangle &&
\end{array}
$$

Cet ensemble de spécifications sera noté $\mathcal{PTSO}$.

Nous retrouvons les paramètres Axiom, Rule et $\leq$ des PTS. Leur signification est inchangée. Parmi les nouveaux paramètres, il y a Pair, qui régit la formation des paires exactement de la même manière que Rule régissait celle des produits. Enfin, Σ_0 et Σ_1 sont les relations définissant le typage des opérateurs d'arité zéro ou un introduits dans la section 5.4.

5.5.2 Règles de typage

Par rapport aux PTS, on note peu de changements essentiels : la règle d'application est supprimée. Elle est avantageusement remplacée par celle des opérateurs. Nous avons en outre la règle d'introduction des paires.

Définition 5.17 (prédicat `typ,wf`) *Les règles de typage définissant les jugements de typage des termes et de bonne formation des contextes des PTSO sont présentées figures 5.2 et 5.1.*

$$(\text{WF-}[\,]) \ \frac{}{[\,] \vdash} \qquad (\text{WF-VAR}) \ \frac{\Gamma \vdash \qquad \Gamma \vdash T : s}{\Gamma\,[x : T] \vdash} (s \in \text{SORT})$$

$$(\text{WF-DEF}) \ \frac{\Gamma \vdash \qquad \Gamma \vdash M : T \qquad \Gamma \vdash T : s}{\Gamma\,[x = M : T] \vdash} (s \in \text{SORT})$$

FIG. 5.1: Règles de typage des contextes

$$(\text{SRT}) \ \frac{\Gamma \vdash \qquad (s_1, s_2) \in \text{AXIOM}}{\Gamma \vdash s_1 : s_2} \qquad (\text{REL}) \ \frac{\Gamma \vdash \qquad \Gamma(n) = \delta}{\Gamma \vdash \natural n : \uparrow^{n+1} \delta.\text{T}}$$

$$(\text{OP0}) \ \frac{\Gamma \vdash \qquad [\_ : U] \vdash \qquad (c, U) \in \Sigma_0}{\Gamma \vdash c : U}$$

$$(\text{OP1}) \ \frac{\Gamma\,[\_ : T] \vdash \qquad \Gamma\,[\_ : U] \vdash \qquad \Gamma \vdash M : T \qquad (c, \Gamma, M, T, U) \in \Sigma_1}{\Gamma \vdash c(M) : U}$$

$$(\text{PROD}) \ \frac{\Gamma \vdash T : s_1 \qquad \Gamma\,[x : T] \vdash U : s_2}{\Gamma \vdash \Pi x{:}T.\, U : s_3} \ (s_1, s_2, s_3 \in \text{RULE})$$

$$(\text{LAM}) \ \frac{\Gamma\,[x : T] \vdash M : U \qquad \Gamma \vdash \Pi x{:}T.\, U : s}{\Gamma \vdash \lambda x{:}T.\, M : \Pi x{:}T.\, U} (s \in \text{SORT})$$

$$(\text{SUM}) \ \frac{\Gamma \vdash A : s_1 \qquad \Gamma \vdash B : s_2}{\Gamma \vdash A * B : s_3} \ (s_1, s_2, s_3 \in \text{PAIR})$$

$$(\text{PAIR}) \ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B \qquad \Gamma \vdash A * B : s}{\Gamma \vdash (M,\, N) : A * B} (s \in \text{SORT})$$

$$(\text{CONV}) \ \frac{\Gamma \vdash M : U \qquad \Gamma \vdash V : s \qquad U \leq_\Gamma V}{\Gamma \vdash M : V} \ (s \in \text{SORT})$$

$$(\text{CONV-SRT}) \ \frac{\Gamma \vdash M : U \qquad U \leq_\Gamma s}{\Gamma \vdash M : s} \ (s \in \text{SORT})$$

FIG. 5.2: Règles de typage

Dans le système proposé, certaines prémisses s'avèrent superflues : dans chacune des règles WF-VAR et WF-DEF, la première prémisse est inutile puisque nous pourrons prouver que le contexte de n'importe quel jugement de typage est bien formé. La raison d'une telle redondance est de bénéficier d'un schéma de récurrence (engendré automatiquement par `Coq` suivant la définition de `typ` et `wf`) offrant plus d'hypothèses de récurrence. Il pourrait être intéressant de s'assurer de l'équivalence proposé avec celui n'ayant aucune prémisse superflue.

Les noms apparaissant dans les contextes n'ont pas à être distincts puisque nous utilisons les indices de de Bruijn. Nous rappelons que les noms ne sont considérés que comme des indications sur le nom à donner à la variable, mais ne servent pas à y faire référence (si ce n'est dans la syntaxe concrète).

5.5.3 Règles dérivées

La relation RULE*, obtenue en complétant RULE avec l'inclusion entre sortes, se définit comme pour les PTS. Symétriquement, nous définissons PAIR* et montrons sa règle dérivée associée.

Définition 5.18 (prédicat `pr_sat`) *Les règles de formation des paires peuvent être complétées par le sous-typage, comme les règles de formation des produits :*

$$(s_1, s_2, s_3) \in \text{PAIR}^* \stackrel{\text{def}}{\equiv} \exists s_1', s_2'.\, s_1 \leq s_1' \ \wedge\ s_2 \leq s_2' \ \wedge\ (s_1', s_2', s_3) \in \text{PAIR}$$

Lemme 5.19 (`typ_sum`) *La règle suivante est dérivable :*

$$\frac{\Gamma \vdash T : s_1 \qquad \Gamma \vdash U : s_2}{\Gamma \vdash T * U : s_3}\ (s_1, s_2, s_3 \in \text{PAIR}^*)$$

Comme pour les règles de formation des produits pour les PTS, étant donné une signature Σ_1, nous pouvons définir une signature qui rend inutile l'application de la règle de sous-typage en fin de dérivation du jugement de typage de l'argument de l'opérateur.

Définition 5.20 (prédicat `in_sign_sat`) *Pour toute signature Σ_1, on définit $\Sigma_1^{\leq}$, la signature complétée par la relation de sous-typage :*

$$(c, \Gamma, M, T, U) \in \Sigma_1^{\leq} \stackrel{\text{def}}{\equiv} \exists T'.\ \begin{cases} \Gamma \vdash T \leq T' \\ (c, \Gamma, M, T', U) \in \Sigma_1 \\ \Gamma\,[\_ : T'] \vdash \\ \Gamma\,[\_ : U] \vdash \end{cases}$$

Ici encore, cette définition donne lieu à une règle dérivée permettant de l'utiliser :

Lemme 5.21 (`typ_cst1_sat`) *La règle suivante est dérivable :*

$$\frac{\Gamma \vdash M : T \qquad (c, \Gamma, M, T, U) \in \Sigma_1^{\leq}}{\Gamma \vdash c(M) : U}$$

5.6 Métathéorie du système avec opérateurs

Les règles de typage ont été renforcées de manière à ne pas avoir à faire de récurrence sur la taille des dérivations, mais uniquement des récurrences structurelles.

Un bon nombre des résultats ci-dessous sont formulés de manière identique au cas des PTS. Parfois, les preuves diffèrent légèrement. Nous nous contenterons de relever les différences.

Lemmes d'inversion

Lemme 5.22 (`typ_wf`) *Le contexte de tout jugement de typage dérivable est bien formé. La règle suivante est admissible :*

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash}$$

Lemme 5.23 (`inversion_lemma`) *Les lemmes d'inversion du jugement de typage se déduisent systématiquement de la définition du jugment de typage. La figure 5.3 récapitule les énoncés.*

Lemme des variables libres

Lemme 5.24 (`typ_clos`) *Toutes les variables libres d'un terme bien typé sont liées dans le contexte :*

$$\Gamma \vdash M : T \;\Rightarrow\; |\Gamma| \in |M|$$

Preuve Par récurrence sur $\Gamma \vdash M : T$, sans aucune difficulté. Les cas de base sont élémentaires. Les autres cas se resolvent simplement grâce aux hypothèses de récurrence. ■

Lemme d'affaiblissement

Le lemme d'affaiblissement (ainsi que le lemme de substitution) diffère légèrement du cas des PTS.

Lemme 5.25 (`typ_weak`) *Le typage est préservé par ajout d'une déclaration dans le contexte, dans la mesure où celui-ci reste bien formé. La règle*

$$\frac{\Gamma\Delta \vdash M : T \qquad \Gamma\delta \vdash}{\Gamma\delta(\Delta^+) \vdash \uparrow^1_{|\Delta|} M : \uparrow^1_{|\Delta|} T}(s \in \textsc{Sort})$$

est admissible.

Preuve Nous ne pouvons plus utiliser la même astuce que dans le cas des PTS, à cause du cas des opérateurs d'arité un. Pour avoir les hypothèses de récurrence manquantes, nous prouvons le lemme d'affaiblissement par récurrence mutuelle. Nous introduisons d'abord l'hypothèse $\Gamma\delta \vdash$, puis nous prouvons simultanément :

$$\frac{\Gamma\Delta \vdash M : T}{\Gamma\delta(\Delta^+) \vdash \uparrow^1_{|\Delta|} M : \uparrow^1_{|\Delta|} T} \qquad \frac{\Gamma\Delta \vdash}{\Gamma\delta(\Delta^+) \vdash}$$

La preuve ne présente alors aucune difficulté notable. ■

Lemme de substitution

Lemme 5.26 (`typ_sub`) *Le typage est préservé par toute substitution d'une variable par un terme dont le type est celui déclaré pour la variable. La règle*

$$\frac{\Gamma\delta\Delta \vdash M : T \qquad \Gamma \vdash N : \delta_{\mathrm{T}} \qquad N \in \mathsf{ValOk}(\delta)}{\Gamma(\Delta\,\{N\}) \vdash M\{|\Delta|\backslash N\} : T\{|\Delta|\backslash N\}}$$

est admissible.

Preuve Comme pour le lemme d'affaiblissement, après avoir introduit les hypothèses annexes (ici $\Gamma \vdash N : \delta_{\mathrm{T}}$ et $N \in \mathsf{ValOk}(\delta)$), nous prouvons par récurrence mutuelle les énoncés :

$$\frac{\Gamma\delta\Delta \vdash M : T}{\Gamma(\Delta\,\{N\}) \vdash M\{|\Delta|\backslash N\} : T\{|\Delta|\backslash N\}} \qquad \frac{\Gamma\delta\Delta \vdash}{\Gamma(\Delta\,\{N\}) \vdash}$$

■

$$\Gamma \vdash s_1 : T \quad \Rightarrow \quad \exists s_2 \in \textsc{Sort}. \begin{cases} (s_1, s_2) \in \textsc{Axiom} \\ \Gamma \vdash s_2 \leq T \end{cases}$$

$$\Gamma \vdash \natural n : T \quad \Rightarrow \quad \exists \delta \in \textsc{Decl}. \begin{cases} \Gamma(n) = \delta \\ \Gamma \vdash \uparrow^{n+1} \delta_{\mathrm{T}} \leq T \end{cases}$$

$$\Gamma \vdash c : T \quad \Rightarrow \quad \exists U \in \textsc{Term}. \begin{cases} (c, U) \in \Sigma_0 \\ [\_ : U] \vdash \\ \Gamma \vdash U \leq T \end{cases}$$

$$\Gamma \vdash c(M) : T \quad \Rightarrow \quad \exists A, B \in \textsc{Term}. \begin{cases} (c, \Gamma, M, A, B) \in \Sigma_1 \\ \Gamma\,[\_ : A] \vdash \\ \Gamma\,[\_ : B] \vdash \\ \Gamma \vdash M : A \\ \Gamma \vdash B \leq T \end{cases}$$

$$\Gamma \vdash \lambda x{:}A.\, M : T \quad \Rightarrow \quad \exists B \in \textsc{Term}.\, \exists s \in \textsc{Sort}. \begin{cases} \Gamma\,[x : A] \vdash M : B \\ \Gamma \vdash \Pi x{:}A.\, B : s \\ \Gamma \vdash \Pi x{:}A.\, B \leq T \end{cases}$$

$$\Gamma \vdash \Pi x{:}A.\, B : T \quad \Rightarrow \quad \exists (s_1, s_2, s_3) \in \textsc{Rule}. \begin{cases} \Gamma \vdash A : s_1 \\ \Gamma\,[x : A] \vdash B : s_2 \\ \Gamma \vdash s_3 \leq T \end{cases}$$

$$\Gamma \vdash (M,\, N) : T \quad \Rightarrow \quad \exists A, B \in \textsc{Term}.\, \exists s \in \textsc{Sort}. \begin{cases} \Gamma \vdash M : A \\ \Gamma \vdash N : B \\ \Gamma \vdash A * B : s \\ \Gamma \vdash A * B \leq T \end{cases}$$

$$\Gamma \vdash A * B : T \quad \Rightarrow \quad \exists (s_1, s_2, s_3) \in \textsc{Pair}. \begin{cases} \Gamma \vdash A : s_1 \\ \Gamma \vdash B : s_2 \\ \Gamma \vdash s_3 \leq T \end{cases}$$

FIG. 5.3: Lemmes d'inversion du jugement de typage

Correction des types

Lemme 5.27 (`type_correctness`) *Seuls les types bien formés et les sortes sont habités : pour tout environnement* Γ *et tout terme* M, T*,*

$$\Gamma \vdash M : T \;\Rightarrow\; T \in \mathcal{WT}_\Gamma$$

Renforcement du contexte

Les résultats de réduction dans l'environnement sont plus fins :

Lemme 5.28 (`reduction_env_rec`) *Soit* S *une relation entre déclarations vérifiant les trois propriétés suivantes (pour tout* $\Gamma, \Gamma', \delta, \delta', c, M, T, U$*) :*

- $S(\Gamma, \delta, \delta') \;\Rightarrow\; \Gamma \vdash \delta_{\cdot \mathrm{T}} \leq \delta'_{\cdot \mathrm{T}}$
- $\leq \in \mathsf{Rstable}_S$
- $(\Gamma')\ S\ (\Gamma) \;\wedge\; (c, \Gamma, M, T, U) \in \Sigma_1 \;\Rightarrow\; (c, \Gamma', M, T, U) \in \Sigma_1$

Alors, la règle suivante est admissible :

$$\frac{\Gamma \vdash M : T \qquad (\Gamma')\ S\ (\Gamma) \qquad \Gamma' \vdash}{\Gamma' \vdash M : T}$$

L'énoncé de ce lemme est relativement complexe, et souvent nous n'utiliserons qu'un cas particulier de ce lemme. Le corollaire suivant sera par exemple utilisé pour prouver l'auto-réduction de β.

Corollaire 5.29 (`subtype_env`) *Si l'on remplace le contexte* Γ *d'un jugement de typage par un contexte* Γ' *bien formé et plus fort (i.e. comportant des types plus petits au sens de* $\leq$*), alors on obtient encore un jugement dérivable.*

$$\frac{\Gamma \vdash M : T \qquad \Gamma' \vdash \qquad (\Gamma')\ \mathrm{ty}(\leq)\ (\Gamma)}{\Gamma' \vdash M : T}$$

Preuve Il suffit d'instancier la relation S du lemme précédent avec $\mathrm{ty}(\leq)$. ■

5.7 Décidabilité du typage

Comme nous avons un certain nombre de constructeurs de termes, la preuve de décidabilité du typage devient assez longue. Ce qui n'est pas très pratique puisque cela rend les scripts de preuves Coq fragiles, i.e. une modifications anodine peut avoir d'importantes conséquences sur le script. Noter que la même chose se produit avec l'implantation : si nos termes comportaient beaucoup de constructions, l'algorithme risquerait d'être compliqué, et l'on écrirait plutôt une fonction auxiliaire pour chaque construction.

Nous allons donc rendre la preuve plus modulaire en la découpant en morceaux, chaque morceau étant responsable de faire les vérifications nécessaires pour une des constructions du langage. Ces fonctions jouent alors un rôle équivalent aux règles élémentaires de LCF. Nous définirons concrètement la notion de jugement. Jusqu'ici, la dérivabilité d'un jugement n'était qu'une propriété logique. Nous allons introduire le type des jugement dérivables. Après extraction, ce type sera isomorphe au type des jugements, c'est-à-dire un couple de termes (le contexte restant implicite).

Tout d'abord, nous décrivons l'ensemble des hypothèses qui permettront de décider les jugements de typage.

(* Nom *)	(* Spécification *)
`infer_axiom:`	$\forall s_1.\, \mathsf{InfPpal}(s_2 \mid (s_1, s_2) \in \textsc{Axiom}, \leq);$
`infer_sign0:`	$\forall c.\, \mathsf{InfPpal}(U \mid (c, U) \in \Sigma_0 \ \wedge\ [\_ : U] \vdash, =);$
`infer_sign1:`	$\forall c, \Gamma, M, T.$ $\Gamma \vdash M : T \ \Rightarrow\ \mathsf{InfPpal}(U \mid (c, \Gamma, M, T, U) \in \Sigma_1^{\leq}, \leq);$
`least_sort:`	$\forall \Gamma.\, \forall M \in \mathcal{WT}_\Gamma.\, \mathsf{InfPpal}(s \in \textsc{Sort} \mid \Gamma \vdash M \leq s, \leq);$
`infer_rule:`	$\forall s_1, s_2.\, \mathsf{InfPpal}(s_3 \mid (s_1, s_2, s_3) \in \textsc{Rule}^*, \leq);$
`weak_rule:`	$\forall \Gamma, s_1, B.$ $\Gamma\,[\_ : B] \vdash \Rightarrow\ \mathsf{InfPpal}(B' \mid B' \in \mathrm{LeastRange}(\textsc{Rule}^*, \Gamma, s_1, B), \leq);$
`sub_prod:`	$\forall \Gamma, x, A, B, B'.$ $\Gamma\,[x : A] \vdash B \leq B' \ \Rightarrow\ \Gamma \vdash \Pi x{:}A.\, B \leq \Pi x{:}A.\, B';$
`infer_prod:`	$\forall s_1, s_2.\, \exists^{\mathsf{Ppal}}(s_3 \mid (s_1, s_2, s_3) \in \textsc{Pair}^*, \leq);$
`sub_sum:`	$\forall \Gamma, A, A', B, B'.$ $\Gamma \vdash A \leq A' \ \wedge\ \Gamma \vdash B \leq B' \ \Rightarrow\ \Gamma \vdash A * B \leq A' * B';$
`topsort:`	$\forall \Gamma, T, s_1, s.$ $\Gamma \vdash s_1 \leq T \ \wedge\ \Gamma \vdash T : s \ \Rightarrow\ \exists s_2.\, (s_1, s_2) \in \textsc{Axiom};$
`sub_dec:`	$\forall \Gamma.\, \forall A, B \in \mathcal{WT}_\Gamma.\, \mathsf{Dec}\,(\Gamma \vdash A \leq B)$

FIG. 5.4: Conditions assurant la décidabilité du typage (PTSOPALGOS)

5.7.1 Propriétés requises

Parmi les algorithmes que nous devrons fournir, il faudra être capable de calculer le sur-type appartenant à la plus petite sorte. Il sera utilisé pour le typage de l'abstraction.

Définition 5.30 (`weak_to_least_sort`) *La spécification de la fonction (pouvant échouer) calculant le plus petit sur-type T de B tel que T habite une sorte avec lequel on puisse former un produit avec un type de la sorte s_1.*

$$\text{LeastRange}(R, \Gamma, s_1, B) \stackrel{\text{def}}{\equiv} \{T \mid \Gamma \vdash B \leq T \ \wedge \ \exists s_2, s_3 \in \text{SORT}.\, (\Gamma \vdash T : \ s_2 \ \wedge \ R(s_1, s_2, s_3))\}$$

Définition 5.31 (type `PTSO_algos`) *La structure de donnée regroupant les algorithmes et propriétés assurant la décidabilité du typage est définie figure 5.4.*

Cette structure est la contrepartie de PTSALGOS pour nos systèmes de types. Elle comporte plus de champs car d'une part nous avons plus de constructions élémentaires (le produit cartésien), et d'autre part car nous sommes plus généraux en ne nous limitant pas aux PTS *full*.

Comme d'habitude, la première condition sert à inférer le type des sortes. Les deux suivantes pour le typage des opérateurs. Notamment, Σ_0 doit être injective. Le champ `least_sort` sert à vérifier qu'un terme est un type bien formé (typable par une sorte).

Ensuite, `infer_rule` et `weak_rule` sont les propriétés de décidabilité requises sur RULE. L'énoncé de `weak_rule` est un peu complexe, mais il généralise facilement les hypothèses simplificatrices que l'on fait généralement pour avoir un typage décidable (absence de sorte non typée, PTS *full* ou *semi-full*). Nous pouvons même prouver qu'il s'agit d'une condition nécessaire pour l'existence d'un programme calculant le type principal d'un terme.

Dans le développement, nous supposerons que toutes les paires peuvent être formées, au sens où pour toutes les sortes s_1 et s_2, il esiste au moins une sorte s_3, telle que $(s_1, s_2, s_3) \in$ PAIR. Pour supprimer cette restriction, il faudrait que l'on puisse reconstruire la sorte de n'importe quel type que l'on sait bien formé, ou alors annoter les paires avec les types des deux composantes.

Le champ `topsort` assure qu'une sorte non typée n'a pas de sur-type qui soit typable par une sorte. Ainsi, si le corps d'une abstraction est typé par une sorte non typée, le produit ne peut être formé même en utilisant le sous-typage. Cela suit l'idée que le typage des sortes et l'inclusion entre sortes doivent aller dans le même sens, sous peine d'introduire le paradoxe de Girard.

5.7.2 Jugements, hypothèses et définitions

Nous n'allons pas construire directement la fonction de typage comme pour les PTS. On va procéder un peu à la manière de LCF, avec un type de donnée pour les jugements. Mais il n'est pas abstrait. En revanche, on fait les preuves qui assurent que les jugement formés sont corrects.

Type principal

On va chercher à inférer le type principal. Il est important de pouvoir calculer le type principal, car il permet de montrer la non-typabilité d'un terme.

Par exemple, Type(0) a le type Type(2), et $\lambda x\colon \text{Type}(1).\, x$ a le type $\text{Type}(1) \to \text{Type}(1)$. Bien que Type(2) ne puisse se coercer vers Type(1), le terme $(\lambda x\colon \text{Type}(1).\, x\ \text{Type}(0))$ est bien typé.

Définition 5.32 (type `ppal_judge`) *Un jugement principal dans le contexte* Γ *est un couple de termes* (v, t) *tel que* t *est le type principal de* v*, i.e.* t *est un type de* v*, et tout autre type* U *de* v *est plus grand que* t *:*

$$\textsc{Judge}_\Gamma \stackrel{\text{def}}{\equiv} \{\langle v : \textsc{Term}; t : \textsc{Term}\rangle \mid \Gamma \vdash v : t \ \wedge\ \forall U.\ \Gamma \vdash v : U \ \Rightarrow\ \Gamma \vdash t \leq U\}$$

Lorsque l'on compose les jugements, on ne veut pas avoir à comparer les environnements dans lesquels ils sont exprimés, car ce serait une opération très coûteuse. On se sert des types dépendants pour différencier les jugements en fonction de l'environnement. Les types dépendants permettent d'assurer que deux jugements sont exprimés dans le même contexte sans avoir à explicitement comparer les deux contextes.

Définition 5.33 (type `infer_typ_ppal`) *La spécification de l'inférence de type est l'existence (constructive) d'un jugement principal dont le terme est* M *dans le contexte* Γ *ou la non-typabilité de* M *:*

$$\mathsf{InfJdge}(\Gamma,\, M) \stackrel{\text{def}}{\equiv} (\exists^\star J \in \textsc{Judge}_\Gamma.\, J.\textsc{v} = M) \ +\ (\forall T.\, \neg\Gamma \vdash M : T)$$

Dans le chapitre sur les PTS, les fonctions de typage ne retournaient que le type. Ici, nous retournons un jugement, c'est-à-dire un couple terme-type, avec la contrainte que le terme est M.

Sorte principale

Par ailleurs, plusieurs règles demandent qu'un sous-terme soit typé par une sorte. On factorise ces vérifications en introduisant une autre structure de donnée similaire aux jugements, mais pour les jugements dont le type est une sorte. De la même manière, on va toujours chercher à calculer la plus petite sorte, pour pouvoir montrer la non-typabilité, par exemple lorsque l'on forme un produit.

Définition 5.34 (type `assum`) *Une hypothèse est une structure regroupant un nom, un type et la sorte principale de ce type.*

$$\textsc{Hyp}_\Gamma \stackrel{\text{def}}{\equiv} \{\langle x : \textsc{Name}; t : \textsc{Term}; k : \textsc{Sort}\rangle \mid \Gamma \vdash t : k \ \wedge\ \forall s \in \textsc{Sort}.\ \Gamma \vdash t : s \ \Rightarrow\ k \leq s\}$$

Attention : il ne faut pas confondre sorte principal et type principal qui est une sorte. Il est vrai que si le type principal d'un terme est une sorte, alors c'est aussi la sorte principale de ce terme. En revanche, un terme peut avoir un type principal et une sorte principale qui soient différents. Cela arrive lorsque l'on a du sous-typage qui permet de coercer des types qui ne sont pas des sortes vers une sorte.

Par exemple, on peut imaginer avoir une coercion des booléens vers les propositions. Dans ce cas, le type principal de `true` serait `bool`, mais il aurait comme sorte principale Prop.

Définition 5.35 (`push_assum`) *Ajout d'une hypothèse* $H \in \textsc{Hyp}_\Gamma$ *dans son contexte* Γ *:*

$$\Gamma H \stackrel{\text{def}}{\equiv} \Gamma\,[H.\textsc{x} : H.\textsc{t}]$$

La cohérence entre le contexte dans lequel est exprimé l'hypothèse et le contexte dans lequel elle est ajoutée se fait grâce aux types dépendants.

Nous introduisons d'autres notations pour les lieurs (abstraction et produit). pour $A \in \textsc{Hyp}_\Gamma$:

$$\lambda H.\, M \stackrel{\text{def}}{\equiv} \lambda H.\textsc{x} : H.\textsc{t}.\, B$$

$$\Pi H.\, B \stackrel{\text{def}}{\equiv} \Pi H.\textsc{x} : H.\textsc{t}.\, B$$

Lemme 5.36 (`push_wf`) *Une hypothèse est une structure qui contient suffisamment d'informations pour pouvoir être ajoutée à l'environnement sans autre vérification :*

$$\forall H \in \textsc{Hyp}_\Gamma.\ \Gamma H \vdash$$

Définition 5.37 (type `infer_sort_ppal`) *La spécification de la construction (pouvant échouer) d'une hypothèse de type T et de nom x dans le contexte Γ est la suivante :*

$$\mathsf{InfHyp}(\Gamma,\, x,\, T) \stackrel{\text{def}}{\equiv} (\exists^\star H \in \textsc{Hyp}_\Gamma.\, H.\textsc{x} = x \wedge H.\textsc{t} = T) \ +\ (\forall s \in \textsc{Sort}.\, \neg \Gamma\,[x : T] \vdash)$$

Algorithme 5.38 (`infer_assum`) *Avec les hypothèse faites, il décidable de savoir si un jugement peut être transformé en une hypothèse.*

$$\forall x.\, \forall J \in \textsc{Jdge}_\Gamma.\ \mathsf{InfHyp}(\Gamma,\, x,\, J.\textsc{v})$$

Preuve Il suffit de vérifier que le type du jugement peut s'affaiblir vers une sorte avec `least_sort`. ■

Définitions

On fait de même avec les termes dont le type est typé par une sorte. Cette dernière structure assure que le terme pourra être introduit dans le contexte sous forme de définition.

Définition 5.39 (type `const`) *Une définition est une strucutre regroupant un nom, un jugement principal, et une preuve que le type de jugement est typable par une sorte.*

$$\textsc{Cste}_\Gamma \stackrel{\text{def}}{\equiv} \{\langle x : \textsc{Name};\, j : \textsc{Jdge}_\Gamma\rangle \mid \exists s \in \textsc{Sort}.\, \Gamma \vdash j.\textsc{t} :\ s\}$$

L'ajout d'une constante dans le contexte se note ainsi :

$$\Gamma K = \Gamma\,[K.\textsc{x} = K.\textsc{v} : K.\textsc{t}]$$

Une constante ne contient pas explicitement la sorte, mais simplement la preuve (non calculatoire) qu'il existe une sorte qui type le type de la définition.

Lemme 5.40 (`push_const_wf`) *Une constante peut être poussée dans le contexte (en ajoutant une définition) sans autre condition :*

$$\forall K \in \textsc{Cste}_\Gamma.\ \Gamma K \vdash$$

Lemme 5.41 (`infer_def`) *Étant donné un jugement, il est décidable de savoir s'il peut être transformé en une constante.*

$$\forall x.\, \forall J \in \textsc{Jdge}_\Gamma.\, (\exists^\star K \in \textsc{Cste}_\Gamma.\, K.\textsc{j} = J\ \wedge\ K.\textsc{x} = x) + \neg(\Gamma\,[x = J.\textsc{v} : J.\textsc{t}] \vdash)$$

Preuve D'après le lemme de correction des types, soit le type du jugement est typé par une sorte, et c'est gagné, soit c'est une sorte. Il suffit alors de vérifier que ce n'est pas une sorte non typée. ■

Nous avons fini la définition des fonctions auxiliaires de typage. Nous allons maintenant construire les fonctions élémentaires de typage qui composent les jugements.

5.7.3 Opérations élémentaires de typage

Pour chaque construction, on écrit deux fonctions de typage élémentaires qui composent des jugements passés en argument. D'une part, une fonction qui construit le jugement à partir des jugements des sous-termes, avec comme précondition que le jugement résultat est licite. Ensuite, on écrit une fonction qui prend en argument les jugements des sous-termes, fait les vérifications nécessaires pour que la construction du jugement soit correcte. Dans tous les algorithmes de cette section, nous supposerons que le contexte Γ est bien formé.

Sortes

Pour les sortes et les variables, c'est un peu particulier : nous montrons d'abord la fonction qui peut échouer dans la construction du jugement, puis celle qui ne peut échouer.

Algorithme 5.42 (`infer_sort`) *La construction d'un jugement dont le sujet est une sorte est décidable :*

$$\forall s \in \textsc{Sort}.\ \mathsf{InfJdge}(\Gamma,\, s)$$

Preuve Conséquence directe de `infer_axiom`. ■

Algorithme 5.43 (`mk_sort`) *S'il existe une sorte s' telle que $(s, s') \in \textsc{Axiom}$, alors la construction du jugement n'échoue pas :*

$$\forall s \in \textsc{Sort}.\, (\exists s'.\, (s, s') \in \textsc{Axiom}) \ \Rightarrow\ \exists^\star J \in \textsc{Jdge}_\Gamma.\, J.\textsc{v} = s$$

Remarque : l'existentielle n'étant pas calculatoire, il faut quand même faire appel à `infer_axiom` pour retrouver la sorte principale de s.

Variables

Comme pour les sortes, il est plus simple de montrer en premier la fonction de construction pouvant échouer.

Algorithme 5.44 (`infer_rel`) *Le typage des indices de de Bruijn est décidable :*

$$\forall n \in \mathbb{N}.\ \mathsf{InfJdge}(\Gamma,\, \natural n)$$

Algorithme 5.45 (`mk_rel`) *Pour que $\natural n$ soit typable, il suffit que n soit inférieur à la longueur du contexte :*

$$\forall n \in \mathbb{N}.\, n < |\Gamma| \ \Rightarrow\ \exists^\star J \in \textsc{Jdge}_\Gamma.\, J.\textsc{v} = \natural n$$

Opérateurs

Algorithme 5.46 (`mk_cst0`)

$$\forall c, U.\ [\_ : U] \vdash\ \wedge\ (c, U) \in \Sigma_0 \ \Rightarrow\ \exists^\star J \in \textsc{Jdge}_\Gamma.\, J.\textsc{v} = c$$

Algorithme 5.47 (`infer_cst0`) *L'inférence dy type des opérateurs d'arité zéro est décidable :*

$$\forall c \in \textsc{Oper}.\ \mathsf{InfJdge}(\Gamma,\, c)$$

Algorithme 5.48 (`mk_cst1`)

$$\forall c.\, \forall K \in \textsc{Cste}_\Gamma.\, (\exists^{\mathsf{Ppal}}(U \mid (c, \Gamma, K.\textsc{v}, K.\textsc{t}, U) \in \Sigma_1^{\leq}, \leq)) \ \Rightarrow\ \exists^\star J \in \textsc{Jdge}_\Gamma.\, J.\textsc{v} = c(K.\textsc{v})$$

Algorithme 5.49 (`infer_cst1`) *L'inférence du type des opérateurs d'arité un est décidable :*

$$\forall c \in \textsc{Oper}.\, \forall J_1 \in \textsc{Jdge}_\Gamma.\ \mathsf{InfJdge}(\Gamma,\, c(J_1.\textsc{v}))$$

Produit et abstraction

Algorithme 5.50 (`mk_prd`)

$$\forall H \in \text{HYP}_\Gamma.\, \forall B \in \text{HYP}_{\Gamma H}.\, (\exists^\star s_3.\, (H.\text{K}, B.\text{K}, s_3) \in \text{RULE}^*) \\ \Rightarrow\ \exists^\star J \in \text{JDGE}_\Gamma.\, J.\text{V} = \Pi H.(B.\text{T})$$

Algorithme 5.51 (`infer_prd`) *L'inférence du type d'une abstraction est décidable lorsque son sous-terme gauche est une hypothèse et son sous-terme droit un terme bien typé :*

$$\forall H \in \text{HYP}_\Gamma.\, \forall J \in \text{JDGE}_{\Gamma H}.\ \mathsf{InfJdge}(\Gamma,\, \Pi H.(J.\text{V}))$$

Noter que dans le cas présent, il faut que le sous-terme gauche soit une hypothèse, et non pas simplement un jugement. En effet, l'algorithme de typage ne pourra être appelé qu'avec un contexte bien formé. Donc, pour former le jugement J, il faut avoir prouvé que le contexte étendu est bien formé, cela revient à dire que l'on a déjà formé l'hypothèse pour le sous-terme gauche.

Algorithme 5.52 (`mk_lam`)

$$\forall H \in \text{HYP}_\Gamma.\, \forall J \in \text{JDGE}_{\Gamma H}.\, (\exists^{\mathsf{Ppal}}(U \mid \text{LeastRange}(\text{RULE}^*, \Gamma H, H.\text{K}, J.\text{T}), \leq)) \\ \Rightarrow\ \exists^\star J \in \text{JDGE}_\Gamma.\, J.\text{V} = \lambda H.(J.\text{V})$$

Algorithme 5.53 (`infer_lam`) *L'inférence du type d'une abstraction est décidable :*

$$\forall H \in \text{HYP}_\Gamma.\, \forall J_1 \in \text{JDGE}_{\Gamma H}.\ \mathsf{InfJdge}(\Gamma,\, \lambda H.(J_1.\text{V}))$$

Type somme et paire

Nous avons supposé que l'on pouvait former toutes les paires, c'est-à-dire qu'étant donné deux sortes s_1 et s_2, il existe une sorte s_3 telle que $(s_1, s_2, s_3) \in \text{PAIR}$. La formation du type somme ne requiert pas de condition particulière.

Algorithme 5.54 (`mk_sum`) *Pour toute paire d'hypothèses (H_1, H_2), on peut former le type somme $H_1.\text{T} * H_2.\text{T}$:*

$$\forall H_1, H_2 \in \text{HYP}_\Gamma.\, \exists^\star J \in \text{JDGE}_\Gamma.\, J.\text{V} = H_1.\text{T} * H_2.\text{T}$$

Algorithme 5.55 (`infer_sum`) *Étant donné deux jugements J_1 et J_2, la construction du jugement somme est décidable :*

$$\forall J_1, J_2 \in \text{JDGE}_\Gamma.\ \mathsf{InfJdge}(\Gamma,\, J_1.\text{V} * J_2.\text{V})$$

Preuve À partir du moment où les jugements peuvent être convertis en hypothèses (avec `infer_assum`), le jugement somme peut être construit avec `mk_sum`. ■

Algorithme 5.56 (`mk_pair`)

$$\forall K_1, K_2 \in \text{CSTE}_\Gamma.\, \exists^{\mathsf{Ppal}}(T \mid \Gamma \vdash (K_1.\text{V},\, K_2.\text{V}) : T, \leq)$$

Algorithme 5.57 (`infer_pair`) *L'inférence du type d'une paire est décidable :*

$$\forall J_1, J_2 \in \text{JDGE}_\Gamma.\ \mathsf{InfJdge}(\Gamma,\, (J_1.\text{V},\, J_2.\text{V}))$$

5.7.4 Construction du noyau

Il est facile de construire l'algorithme qui type entièrement un terme : il suffit d'appeler la règle de typage correspondant à chacun des constructeurs apparaissant dans le terme.

Algorithme 5.58 (`typecheck`) *L'inférence du type principal dans un contexte bien formé est décidable :*

$$\forall \Gamma, M.\ \Gamma \vdash \Rightarrow \mathsf{InfJdge}(\Gamma,\ M)$$

Preuve Pour la plupart des cas, il suffit de typer les sous-termes récursivement, puis appeler la règle appropriée de la deuxième série (celle sans préconditions). Il y a un peu de travail avec l'abstraction et le produit car il faut former une hypothèse avec le sous-terme gauche avant de pouvoir typer le sous-terme droit.

Il est possible de définir des macro-tactiques qui font une bonne partie du travail. Ainsi, pour le typage de l'abstraction le script de tactique reflète parfaitement l'algorithme employé :

1. on type le sous-terme gauche, ce qui produit un jugement ;
2. on transforme ce jugement en hypothèse, grâce à `infer_assum` ;
3. on type le sous-terme droit, ce qui produit un deuxième jugment ;
4. on appelle la fonction élémentaire de typage de l'abstraction (`infer_lam`).

Après ces quatre tactiques, il ne reste plus qu'à prouver que si le premier jugement ne se transforme pas en hypothèse, alors l'abstraction est mal typée (en fait c'est la contraposée qui est automatiquement engendrée, et le lemme d'inversion est appliqué). Deux tactiques achèvent le typage de l'abstraction. ■

La spécification ci-dessus permet de construire un jugement à partir d'un terme M. Nous pourrions prendre en argument un autre genre d'objet. Une relation entre ce type d'objet et les termes permettrait de spécifier le terme du jugement à construire en fonction de l'entrée. En particulier, l'entrée pourrait être un AST (arbre de syntaxe abstraite). Ces AST pourraient comporter des "macros" qui seraient expansées par la fonction de typage. Par exemple, supposons que le système considéré possède l'application. Nous définirions une construction syntaxique `let x = M : T in N`, équivalente au radical $(\lambda x : T.\, N\ M)$. Pour typer notre construction, il suffirait, après avoir typé récursivement les sous-termes, d'appeler successivement `infer_lam` et `infer_cst1` (avec comme opérateur @).

Cela n'est pas très avantageux en soi, puisque l'on pourrait faire un première passe consistant à expanser les macros, puis une deuxième passe ferait le typage. Mais nous pouvons faire mieux, sachant que le type T peut être inféré. Ainsi, la macro s'écrirait `let x = M in N`, s'expansant en $(\lambda x : T.\, N\ M)$ pour un certain type T (non unique). L'avantage est double : d'une part l'utilisateur n'a pas a donner le type T ; le type serait synthétisé par la fonction de typage. D'autre part, la fonction de typage n'a pas a vérifier, comme ce qui serait fait dans le paragraphe précédent, que le type principal de M est bien un sous-type de T. La séquence d'appels pour notre construction serait :

1. typer les sous-termes, ce qui permet de déduire T,
2. appeler `infer_lam` pour s'assurer que l'abstraction est correcte,
3. enfin, appeler `mk_cst1` (avec @) pour construire l'application, sachant que celle-ci sera bien typée.

Cet exemple met en évidence l'utilité de la première série de fonctions de typage élémentaires. La construction d'un jugement à partir d'un AST peut être plus efficace que de partir d'un terme.

Ce genre de techniques peut se généraliser à la résolution d'arguments implicites : avec le produit dépendant, le type d'un argument peut parfois déterminer la valeur d'un des arguments précédents. Prenons l'exemple des listes, dont le constructeur *cons* est de type $\Pi A{:}\mathrm{Set}.\ A \to (\mathrm{List}\ A) \to (\mathrm{List}\ A)$. Le type A peut être inféré à partir du type des deuxième et troisième arguments.

Définition 5.59 (prédicat `PTSO_TC`) *L'interface* PtsOpTc *est à peu près la même que* PtsTc*, sauf que l'on construit des jugements au lieu de simplement inférer des types.*

$$
\begin{array}{lllll}
\langle\ \texttt{tc\_inf\_ppal\_type}: & \forall \Gamma, M. & \Gamma \vdash & \Rightarrow & \mathsf{InfJdge}(\Gamma,\ M); \\
\quad \texttt{tc\_inf\_ppal\_sort}: & \forall \Gamma, x, T. & \Gamma \vdash & \Rightarrow & \mathsf{InfHyp}(\Gamma,\ x,\ T); \\
\quad \texttt{tc\_chk\_typ}: & \forall \Gamma, M, T. & \Gamma \vdash & \Rightarrow & \mathsf{Dec}\,(\Gamma \vdash M\ :\ T)\,; \\
\quad \texttt{tc\_chk\_decl}: & \forall \Gamma, \delta. & \Gamma \vdash & \Rightarrow & \mathsf{Dec}\,(\Gamma\delta \vdash)\,; \\
\quad \texttt{tc\_chk\_wk}: & \forall \Gamma, T. & \Gamma \vdash & \Rightarrow & \mathsf{Dec}\,(T \in \mathcal{WT}_\Gamma)\,; \\
\quad \texttt{tc\_chk\_wft}: & \forall \Gamma, M, T. & \Gamma \vdash\ \wedge\ T \in \mathcal{WT}_\Gamma & \Rightarrow & \mathsf{Dec}\,(\Gamma \vdash M\ :\ T) \\
\rangle &&&&
\end{array}
$$

Théorème 8 (`type_checker`) *Ensuite, on dérive les autres fonctions du noyau comme pour les PTS.*

$$\textsc{PtsOpAlgos} \subseteq \textsc{PtsOpTc}$$

Corollaire 5.60 (`decide_wf`, `decide_type`) *Les jugements de typage sont décidables.*

5.8 Règles de conversion

Cette section correspond à la section 4.4 du développement sur les PTS. Nous définissons les mêmes opérateurs de règles de réduction : fermetures réflexives, symétriques, transitives ou contextuelles. Nous montrons que ces opérateurs préservent les propriétés des règles de sous-typage. La seule différence notable est le changement dans la définition de Rstable, mais cela n'affecte pas les résultats.

De même, nous définissons l'ensemble des termes en forme normale ainsi que les formes normales de tête. Les propriétés de confluence et de Church-Rosser sont inchangées et bénéficient des mêmes propriétés.

Pour montrer la préservation du résultat d'auto-réduction par fermeture contextuelle, nous avons défini un nouvel opérateur : la fermeture contextuelle parallèle, qui permet de faire des réductions dans plusieurs sous termes en même temps.

Définition 5.61 (prédicat `par_ctxt`) *L'opérateur de fermeture contextuelle parallèle est défini figure 5.5.*

Définition 5.62 (`ctxt_sound`) *Soit R une relation ternaire remplissant les conditions suivantes :*

- $R \ \subseteq \leq \cap \leq^{-1}$
- $(c, \Gamma, M, T, U) \in \Sigma_1 \ \wedge\ \Gamma \vdash M \to_{R_{/\!/}} M' \ \Rightarrow\ \exists T', U'.$ $\begin{cases} \Gamma \vdash T \to_{R_{/\!/}} T' \\ \Gamma \vdash U \to_{R_{/\!/}} U' \\ (c, \Gamma, M', T', U') \in \Sigma_1 \end{cases}$

Alors $R \in \mathcal{SR} \ \Rightarrow\ [R] \in \mathcal{SR}$.

$$(\textsc{Step})\ \frac{R(\Gamma, M, N)}{\Gamma \vdash M \to_{R_{/\!/}} N} \qquad (\textsc{Refl})\ \frac{}{\Gamma \vdash M \to_{R_{/\!/}} M}$$

$$(\textsc{Op1})\ \frac{\Gamma \vdash M \to_{R_{/\!/}} M'}{\Gamma \vdash c(M) \to_{R_{/\!/}} c(M')}$$

$$(\textsc{Abs})\ \frac{\Gamma \vdash M \to_{R_{/\!/}} M' \qquad x\,[\Gamma : M] \vdash N \to_{R_{/\!/}} N'}{\Gamma \vdash \lambda x{:}M.\, N \to_{R_{/\!/}} \lambda x{:}M'.\, N'}$$

$$(\textsc{Prd})\ \frac{\Gamma \vdash M \to_{R_{/\!/}} M' \qquad x\,[\Gamma : M] \vdash N \to_{R_{/\!/}} N'}{\Gamma \vdash \Pi x{:}M.\, N \to_{R_{/\!/}} \Pi x{:}M'.\, N'}$$

$$(\textsc{Sum})\ \frac{\Gamma \vdash M \to_{R_{/\!/}} M' \qquad \Gamma \vdash N \to_{R_{/\!/}} N'}{\Gamma \vdash M * N \to_{R_{/\!/}} M' * N'}$$

$$(\textsc{Pair})\ \frac{\Gamma \vdash M \to_{R_{/\!/}} M' \qquad \Gamma \vdash N \to_{R_{/\!/}} N'}{\Gamma \vdash (M,\, N) \to_{R_{/\!/}} (M',\, N')}$$

FIG. 5.5: Fermeture contextuelle parallèle d'une règle de réduction

On montre d'abord le résultat pour $[R]_{/\!/}$.

Enfin, nous utiliserons le même ordre de normalisation. La suite logique serait de définir l'ordre de normalisation, que nous devons adapter du fait du changement dans la formulation de Rstable. Ceci nous permettrait de montrer la correction de l'algorithme de conversion générique dans le cas des relations confluentes et ayant un algorithme de mise en forme normale de tête. Toutefois, nous ne présenterons pas la preuve car elle ne sera pas utilisée dans la suite.

5.9 Conclusion

La formalisation des hypothèses et des jugements, ainsi que le fait d'avoir une fonction de typage par constructeur, que l'on assemble en une seule fonction est une idée que l'on retrouve dans beaucoup d'implantations de systèmes de preuves (LCF, Coq).

Cela rend le code et les preuves très modulaires. L'ajout d'un opérateur primitif dans le système en est grandement facilité. Même s'il reste nécessaire de reparcourir tout le développement, cela rend les scripts assez robustes.

Dans la prochaine partie nous allons étudier le Calcul des Constructions Inductives (le système de types de Coq) dans ce cadre.

Troisième partie

Une présentation du Calcul des Constructions Inductives

Chapitre 6

Le Calcul des Constructions Inductives

Dans cette partie, nous commençons la formalisation du Calcul des Constructions Inductives. Nous allons évidemment utiliser les résultats du chapitre précédent en instanciant les paramètres de nos PTSO (la structure $\mathcal{PTSO}$ pour définir le système, et PtsOpAlgos pour assurer la décidabilité du typage). Donc on se retrouve à peu près au niveau de la section 4.5 dans le développement des PTS.

Ce chapitre est consacré à la définition des différentes composantes de CCI en tant que PTSO, ce qui nécessitera de faire quelques preuves. Les théorèmes plus conséquents et les résultats métathéoriques spécifiques à CCI, feront l'objet du chapitre suivant.

Nous devons tout d'abord instancier les types paramètrant les termes : les variables les sortes et les opérateurs. Les variables seront implantées par les chaînes de caractères de ML. Pour cela nous axiomatisons en Coq un type `ml_string` et nous indiquons à l'extracteur qu'il doit correspondre au type `string` d'Objective Caml. Il convient d'être prudent lors de cette étape car Coq n'a aucun moyen de vérifier la correction de cette axiomatisation.

6.1 Hiérarchie de sortes

Nous reprenons la même hiérarchie de sortes que dans la section 4.7. Rappelons simplement que nous disposons de deux sortes imprédicatives Prop et Set, et une hiérarchie d'univers prédicatifs Type(i). Nous ajouterons quand même une coercion entre Prop et Set. La raison est technique, mais on pourrait faire autrement.

Définition 6.1 (prédicat `univ_v6`) *La relation de cumulativité de la version 6.2 de* Coq *est :*

$$\mathrm{Prop} \prec^{\mathrm{V6}} s \qquad \mathrm{Prop}^{\epsilon} \prec^{\mathrm{V6}} \mathrm{Type}(i) \qquad \mathrm{Type}(i) \prec^{\mathrm{V6}} \mathrm{Type}(j)$$

pour i et j tels que $i \leq j$.

Les règles de typage des sortes $\textsc{Axiom}^{\mathrm{V6}}$ et de formation des produits $\textsc{Rule}^{\mathrm{V6}}$ sont inchangées. Nous avons simplement à donner les règles de formation des paires.

Définition 6.2 (prédicat `pairs_v6`) *Les règles de formation du produit cartésien sont les*

suivantes ($i \leq k$ et $j \leq k$) :

$$(\text{Set}, \text{Prop}^\epsilon, \text{Set}) \in \text{PAIR}^{\text{V6}} \qquad (\text{Prop}^\epsilon, \text{Set}, \text{Set}) \in \text{PAIR}^{\text{V6}}$$
$$(\text{Prop}, \text{Prop}, \text{Prop}) \in \text{PAIR}^{\text{V6}} \qquad (\text{Type}(i), \text{Prop}^\epsilon, \text{Type}(k)) \in \text{PAIR}^{\text{V6}}$$
$$(\text{Prop}^\epsilon, \text{Type}(i), \text{Type}(k)) \in \text{PAIR}^{\text{V6}} \qquad (\text{Type}(i), \text{Type}(j), \text{Type}(k)) \in \text{PAIR}^{\text{V6}}$$

Nous voyons qu'à chaque fois, le produit cartésien vit dans la sorte de ses composante la plus "haute". Le produit cartésien est donc toujours prédicatif. Cela nous est imposé car nous pouvons toujours former les projections. Si nous avions un produit cartésien dans une sorte plus basse qu'une de ses composantes, nous pourrions définir une injection de la sorte la plus haute vers la plus basse, ce qui permettrait d'encoder la paradoxe de Girard.

D'une certaine manière, nous considérons que Set se situe plus haut que Prop dans la hiérarchie des sortes, bien que jusqu'ici, elles paraissaient tout à fait semblables. Cela est dû au fait que nous souhaitons utiliser ces sortes de manière différente : dans Set, nous mettrons les types calculatoires, comme les entiers, pour lesquels nous souhaitons pouvoir prouver que zéro est différent de un. En revanche, dans Prop, nous aimerions pouvoir poser le tiers exclu $\Pi P : \text{Prop}.\, P \vee \neg P$ en axiome. Or, Barbanera et Berardi [3] ont prouvé que dans le Calcul des Constructions, le tiers exclu et l'axiome du choix impliquaient la non-pertinence des preuves, i.e. toutes les preuves d'une même proposition sont égales. Ceci nous empêche de confondre Prop et Set. Il est cependant possible de considérer une injection de Prop vers Set.

6.2 Les opérateurs de CCI

Nous commençons la définition formelle de notre système par l'ensemble des opérateurs du Calcul des Constructions Inductives.

Définition 6.3 (type `cci_op`) *L'ensemble des opérateurs du Calcul des Constructions Inductivesest le suivant :*

$$\begin{aligned} c : \text{OPER} \quad := \quad & \pi_1 \mid \pi_2 \mid @ \mid \text{Cst}\{C\} \\ & \mid \text{Ind}\{I, n\} \mid \text{Constr}\{C\} \mid \text{Case}\{\vec{p}\} \mid \text{Fix} \\ & \mid \epsilon \mid \varnothing \mid ::\mid \mathcal{M} \mid \mathcal{M}^* \\ & \mid \text{Record}\{\vec{x}\} \mid \text{Struct}\{\vec{x}\} \mid \text{Field}\{x\} \end{aligned}$$

où $n \in \mathbb{N}$, $I, C, x \in \text{NAME}$ et $\vec{p}, \vec{x} \in \text{NAME}^*$.

6.2.1 Opérateurs simples

Les trois premiers opérateurs sont les destructeurs des constructeurs de types primitifs : π_1 et π_2 sont respectivement les première et deuxième projection, et @ est l'application. Les projections servent à extraire l'une des composantes d'un couple donné en argument comme le montreront les règles de réduction associées à ces opérateurs.

Pour améliorer la lisibilité, nous emploierons les notations suivantes :

$$\begin{aligned} @((M, N)) &\rightarrow (M\ N) \\ \Pi\_ : A.\, B &\rightarrow A \to B \end{aligned}$$

Le développement formel utillise certaines de ces notations, grâce au système de grammaires extensibles.

Le quatrième opérateur Cst$\{C\}$ sert à référencer les définitions ou axiomes globaux de nom C, par opposition aux indices de de Bruijn qui servent à réprésenter les variables liées.

6.2.2 Types inductifs

Ensuite, nous trouvons les opérateurs implantant les types inductifs : type inductif, constructeurs, filtrage et point fixe. L'expression Ind$\{I, n\}(P, A, L)$ représente le type inductif de nom I, étant appliqué au paramètre P et argument A. Les arguments n et L servent à caractériser les marques associées au types inductif. Lorsque le type inductif n'est pas marqué (i.e. de la forme Ind$\{I, 0\}(p, a, \varnothing)$), nous écrirons Ind$(I, p, a)$. Le terme Constr$\{C\}\,(P, A)$ représente le constructeur de nom C appliqué au paramètre P et argument A.

Les types inductifs et les constructeurs sont toujours totalement appliqués. Le coût d'une telle restriction est de parfois avoir à η-allonger les termes, mais les avantages sont importants. Les règles de réduction (surtout celle du filtrage) seront plus simples car il n'y a pas un nombre arbitraire de constructeurs d'application qui peuvent s'intercaler entre le constructeur et le Case. Ensuite, cela permet de mieux marquer la distinction entre paramètres et arguments.

Comme dans le système Coq actuel, l'opérateur de récursion primitive (comme dans le système T de Gödel) est séparé en un opérateur de filtrage et un opérateur de point fixe. Cela permet des appels récursifs avec des sous-termes indirects (ce qui permet par exemple d'écrire la division par deux sans calculer le reste). Les sections 6.5 et 6.6.1 donneront plus de détails concernant les opérateurs de point fixe et de filtrage.

Le Case est assez proche du Cases de Coq : il y a des motifs de filtrage. Même si les motifs introduits dans cette thèse sont assez peu évolués (ils sont réduits à des noms de constructeurs) et ne permettent pas des définitions aussi élégantes que Cases (thèse de Cornes [14]), il s'agit d'un premier pas. Notamment, le motif "attrappe-tout" devrait pouvoir s'ajouter facilement. L'expression Case$\{\vec{p}\}\,(M, (Q, B))$, ce que nous préfererons noter $\langle Q\rangle$Cases M of $\vec{p} \Rightarrow B$ end représente le filtrage du terme M. Les branches sont codées par la liste de motifs $\vec{p}$ et la liste des corps de branches B (sous forme de n-uplet). Le codage sera expliqué plus loin par les règles de typage. Q est le prédicat indiquant le type des objets construits par filtrage.

La terminaison des expressions de point fixe sera assurée par un système de marques, que nous décrirons dans ces mêmes sections. Les cinq opérateurs suivants servent à représenter les marques et les listes de marques. Tous ces opérateurs sont d'arité zéro à l'exception de :: qui est binaire.

6.2.3 Enregistrements

Enfin, les trois derniers opérateurs servent à implanter les Σ-types, ou enregistrements. Ils permettent de faire des vecteurs dépendants de termes, i.e. le type d'un champ peut dépendre des valeurs des champs précédents. Ainsi on pourra obliger les types inductifs et les constructeurs à n'avoir qu'un seul argument. Nous avons encore le même schéma que pour les produits (fonctionnel et cartésien) et les types inductifs : un opérateur sert à représenter les expressions de type (Record$\{\vec{x}\}$), un opérateur pour introduire des objets de ce type (Struct$\{\vec{x}\}$), et un opérateur pour déstructurer ces objets (Field$\{x\}$).

L'accès aux champs de ces enregistrements se fait par nom. Les noms figurant dans ces trois opérateurs ne sont évidemment pas susceptibles d'être renommés. L'opérateur Record$\{\vec{x}\}$ représente le type des enregistrements dont les noms de champs sont $\vec{x}$. Le type de ces champs est caractérisé par l'argument de l'opérateur. Nous verrons cela plus tard. L'opérateur de construction Struct$\{\vec{x}\}$ possède lui aussi les noms des champs qu'il possède.

Enfin, l'opérateur Field$\{x\}$ est la projection qui permet d'extraire le champ dont le nom est x de l'enregistrement argument.

Un avantage d'avoir des Σ-types de premier ordre (i.e. pas comme un type inductif particulier) est que du point de vue logique, la quantification existentielle se trouve aussi primitive que la quantification universelle. En Coq, la manipulation de formules ayant beacoup de connecteurs existentiels est rendue pénible par le fait qu'il faut destructurer ces connecteurs un par un. La quantification universelle (le produit dépendant) n'a pas cet inconvénient, ce qui fait qu'il est plus commode d'utiliser le codage imprédicatif : une formule $\exists x_1 : T_1, \ldots, x_n : T_n.\, P(x_1, \ldots, x_n)$ sera codée par $\forall Q : \mathrm{Prop}.\, (\forall x_1 : T_1, \ldots, x_n : T_n.\, P(x_1, \ldots, x_n) \to Q) \to Q$. Cet encodage est moins lisible, mais il permet de déstructurer d'un seul coup toutes les existentielles.

6.3 Signature de CCI

6.3.1 Définition de la signature de CCI

On a vu que la signature abstraite était caractérisée par ses relations d'appartenance Σ_0 et Σ_1 (section 5.4). Dans CCI, il y a des opérateurs qui sont définis de manière permanente, comme l'application, les projections, etc. Mais il y a aussi des opérateurs qui peuvent être introduits par l'utilisateur, comme par exemple de nouvelles définitions, eventuellement inductives.

Concrètement, la signature est en quelque sorte l'historique des déclarations d'axiomes, de constantes et de types faites par l'utilisateur. À partir d'une signature concrète (ou *environnement*, on définit la signature abstraite, caractérisée par ses deux relations d'appartenance. À titre d'exemple, les opérateurs application ou projection sont définis indépendemment de l'environnement, alors qu'un constructeur C n'est typable que s'il existe dans l'environnement une déclaration inductive comprenant un constructeur nommé C.

Définition 6.4 (types `one_ind`, `ind_pack`, `constant_obj`, `sig_item`) *Une déclaration globale σ est soit une déclaration de constante ξ (axiome ou définition), soit une déclaration de types mutuellement inductifs Ψ, qui sont des listes de déclarations inductives.*

$$
\begin{array}{lcl}
\xi : \textsc{CstObj} & := & \langle p : \textsc{Term}_0;\ d : \textsc{Decl}_1 \rangle \\
\kappa : \textsc{Constr} & := & \langle x : \textsc{Name};\ a : \textsc{Term}_1;\ i : \textsc{Term}_2 \rangle \\
\Phi : \textsc{OneInd} & := & \langle x : \textsc{Name};\ p : \textsc{Term}_0;\ a : \textsc{Term}_1;\ s : \textsc{Sort};\ ck : \textsc{Sort}\ [c : \textsc{Constr}^*] \rangle \\
\Psi : \textsc{MutInd} & := & \textsc{OneInd}^* \\
\sigma : \textsc{SigItem} & := & \xi \mid \Psi
\end{array}
$$

Nous rappelons que $\textsc{Term}_n$ *désigne l'ensemble des termes n'ayant pas de variables libres au delà de n (voir page 134).*

Les déclarations de constantes sont formées du type des paramètres, et d'une déclaration indiquant le nom, le type et éventuellement la valeur de la constante.

Une déclaration de types mutuellement inductifs est une liste de descripteurs de types inductifs.

Pour chaque type inductif, X est le nom du type, P le type du paramètre, A le type des arguments, S la sorte du type inductif, CK la sorte dans laquelle vivent les arguments des constructeurs et eventuellement C la liste des descripteurs de constructeurs associés. Ainsi, chaque type inductif a ses propres paramètres.

Le champ C (la liste des constructeurs) des déclarations inductives est optionnel. Lorsqu'il n'est pas présent, on a ce qu'on appelle un type inductif abstrait. Sa principale utilité est de permettre de typer les types des constructeurs (qui peuvent lui faire référence dans le cas des types effectivement récursifs). Ainsi, la vérification d'une déclaration inductive se fait en deux temps : d'abord, on vérifie que la déclaration est correcte sans tenir compte des constructeurs. Ensuite, dans le contexte étendu avec les déclarations abstraites que nous venons de vérifier, on type les constructeurs. Une fois ces deux étapes faites, la déclaration inductive peut être ajoutée dans la signature.

Définition 6.5 (`abstract_ind`) *On note $\overline{\Psi}$ la déclaration Ψ dans laquelle on a enlevé les constructeurs.*

Si le type inductif est abstrait (i.e. le champ C n'est pas présent), on ne s'autorise pas à écrire $\Phi.\mathrm{C}$. De même, le fait d'écrire $\Phi.\mathrm{C}$ suppose que le champ C est présent.

Il ne faut surtout pas confondre un type inductif abstrait avec un type inductif avec zéro constructeurs : dans le premier cas, on ne peut pas faire d'élimination car on ne connait pas encore les constructeurs (le Case ne se type que lorsque le champ constructeur est présent). Dans le deuxième, on peut faire une élimination (Case avec zéro branches).

Le champ CK des déclarations inductives indique la sorte des arguments des constructeurs. Cela est nécessaire pour savoir si le type inductif est prédicatif ou non, c'est-à-dire si les arguments des constructeurs sont dans des univers "plus petits" que celui du type inductif. Si ce n'est pas le cas, il faudra restreindre les éliminations, sous peine d'introduire un paradoxe. Nous en reparlerons lorsque nous décrirons l'opérateur de filtrage.

Pour chaque constructeur κ, le champ X désigne le nom du constructeur, A le type des arguments, et I l'instance (la valeur des arguments du type inductif) introduite par le constructeur.

Exemple 6.6 *Le prédicat "être pair" que l'on définirait par deux clauses :*

$$\frac{}{0 \in 2\mathbb{N}} \qquad \frac{n \in 2\mathbb{N}}{n+2 \in 2\mathbb{N}}$$

qui se coderait tout naturellement en Coq *avec un prédicat inductif de la manière suivante :*

```
Inductive pair: nat->Prop :=
  pair_0:  (pair 0)
| pair_SS: (n:nat)(pair n)->(pair (S (S n))).
```

sera représenté dans le système objet (CCI) par la déclaration :

$$\begin{array}{l}\Phi = \langle\ x = \texttt{pair};\ p = 1;\ a = \texttt{nat};\ s = \mathrm{Prop};\ ck = \mathrm{Set} \\ \qquad c = [\ \langle x = \texttt{pair0};\quad a = 1;\qquad\qquad\qquad\qquad i = O\qquad\ \rangle; \\ \qquad\qquad \langle x = \texttt{pairSS};\quad a = \Sigma n{:}\texttt{nat}.\,\mathrm{Ind}(\texttt{pair}, (), n)\,;\quad i = (S\ (S\ n))\ \rangle\ ]\ \rangle\end{array}$$

où 1 est le type ne contenant que l'élément $()$, et $\Sigma n : \texttt{nat}.\,\mathrm{Ind}(\texttt{pair}, (), n)$, le type des couples formé d'un entier n et d'une preuve de $\mathrm{Ind}(\texttt{pair}, (), n)$. Il s'agit d'un Σ-type, que nous coderons à l'aide d'un enregistrement à deux champs.

Définition 6.7 (type `sigma`) *La signature est une liste de déclarations globales.*

$$\Sigma : \textsc{Sign} \quad := \quad \textsc{SigItem}^*$$

6.3.2 Recherche dans la signature

Définition 6.8 (type `global_spec`) *Le type des résultats de recherche dans l'environnement est défini par :*

$$G : \textsc{Glob} \quad := \quad \mathsf{Cst}(\xi) \mid \mathsf{Ind}(\rho, \Phi) \mid \mathsf{Constr}(\Phi, \kappa)$$

avec $\xi \in \textsc{CstObj}$, $\rho \in \vec{\textsc{Name}}$, $\Phi \in \textsc{OneInd}$ *et* $\kappa \in \textsc{Constr}$.

Définition 6.9 (prédicat `sign_glob_spec`) *On définit un prédicat qui à chaque nom défini par l'environnement, associe un descripteur.*

$$\frac{\xi \in \Sigma}{\xi.\mathrm{X} \rhd_\Sigma \mathsf{Cst}(\xi)} \qquad \frac{\Psi \in \Sigma \qquad \Phi \in \Psi}{\Phi.\mathrm{X} \rhd_\Sigma \mathsf{Ind}(FV(\overline{\Psi}), \Phi)}$$

$$\frac{\Psi \in \Sigma \qquad \Phi \in \Psi \qquad \kappa \in \Phi.\mathrm{C}}{\kappa.\mathrm{X} \rhd_\Sigma \mathsf{Constr}(\Phi, \kappa)}$$

Définition 6.10 *On appelle domaine de Σ l'ensemble des noms auxquels on peut associer un descripteur :* $\mathrm{Dom}(\Sigma) \stackrel{\text{def}}{\equiv} \{x \mid \exists G.\, x \rhd_\Sigma G\}$.

Lemme 6.11 (`search_global`) *La recherche d'un nom global dans la signature est décidable.*

$$\forall \Sigma, x.\, \mathsf{DecFind}\,(G \mid x \rhd_\Sigma G)$$

Cette spécification ne peut être utile que si l'on utilise des noms différents pour les inductifs, les constructeurs et les constantes. Sinon, plusieurs résultats sont possibles. Cela vient du fait que l'on voudrait avoir un espace de nom unique pour tous les objets globaux, pour ne pas avoir de syntaxe particulière suivant que le genre d'objet global que l'on désigne. Il est surprenant de constater qu'une considération totalement implémentatoire influe sur la formalisation.

Dans la suite, on raisonnera dans un environnement Σ quelconque fixé, et nous omettrons parfois de mentionner Σ comme paramètre.

6.4 Enregistrements

Comme nous l'avons vu dans l'exemple du prédicat "pair", le produit cartésien n'est pas suffisant pour exprimer certaines définitions inductives, lorsque nous sommes en présence de types dépendants. L'idée est d'introduire un nouvel opérateur de type qui transforme un n-uplet de valeurs de façon à abstraire la valeur de certains champs dans le type des champs suivants.

6.4.1 Definition des arités d'enregistrement

Définition 6.12 (type `rspec`) *Le type d'un enregistrement (ou arité) est caractérisé par une liste de couples nom-type, associant un type à chaque champ. Comme nous avons des Σ-types, le type de chaque champ se comprend dans le contexte dans lequel on a rajouté les déclarations correspondants aux champs précédents.*

$$\Theta : \mathrm{Rspec} \stackrel{\text{def}}{\equiv} (\textsc{Name} * \textsc{Term})^*$$

On notera $\Theta.1$ la liste des noms d'une arité.

Les opérations de relocation et de substitution s'étendent à cette structure, mettant en évidence le fait que chaque type de champ est dans un contexte comprenant les champs précédents :

Définition 6.13 (`lift_rsign, subst_rsign`) *Pour Θ une arité d'enregistrement, nous définissons :*

$$\begin{array}{rcl} \uparrow_k^n [] & = & [] \\ \uparrow_k^n ((x{:}T)\Theta) & = & (x{:}\uparrow_k^n T)\ \uparrow_{k+1}^n \Theta \end{array}$$

$$\begin{array}{rcl} []\{k\backslash M\} & = & [] \\ ((x{:}T)\Theta)\{k\backslash M\} & = & (x{:}T\{k\backslash M\})\ (\Theta\{k+1\backslash M\}) \end{array}$$

Comme nous l'avions fait pour les termes, nous prouvons des résultats de commutation des relocations et substitutions. Nous ne les détaillerons pas.

Définition 6.14 (type `mspec`) *Un enregistrement est une arité pour laquelle nous associons aussi une valeur à chaque champ*

$$\theta : \text{Mspec} \stackrel{\text{def}}{\equiv} (\text{NAME} * \text{TERM} * \text{TERM})^*$$

Contrairement aux types, les valeurs de chacun des champs se trouvent dans le même contexte.

La liste des valeurs d'un enregistrement se notera $\theta.3$. Nous considérons Mspec comme un sous-type de Rspec : il suffit d'oublier le champ correspondant aux valeurs.

La relocation et la substitution de cette structure est plus délicate puisque nous avons d'une part des types dont le contexte change d'un champ à l'autre et d'autre part des valeurs, toutes dans le même contexte. Ces opérations possèdent donc plus de paramètres

Définition 6.15 (`lift_mspec, subst_mspec`)

$$\begin{array}{rcl} \uparrow_{k,k'}^n [] & = & [] \\ \uparrow_{k,k'}^n ((x{:}T{:=}M)\theta) & = & (x{:}\uparrow_k^n T{:=}\uparrow_{k'}^n M)\ \uparrow_{k+1,k'}^n \Theta \end{array}$$

$$\begin{array}{rcl} []\{k,k'\backslash N\} & = & [] \\ ((x{:}T{:=}M)\theta)\{k,k'\backslash N\} & = & (x{:}T\{k\backslash N\}{:=}M\{k'\backslash N\})\ (\theta\{k{+}1,k'\backslash N\}) \end{array}$$

6.4.2 Représentation des n-uplets

Comme les enregistrements sont un moyen de transformer des n-uplets en Σ-types, nous introduisons maintenant des notations pour représenter ces n-uplets, ainsi que les produits cartésiens de n types.

Par souci d'uniformité, le produit cartésien de n types ne sera pas $T_1 * \ldots * T_n$, mais $T_1 * \ldots * T_n * \mathcal{M}^*$. Cela nous évite d'avoir à traiter de manière particulière le cas où le n-uplet est de taille un. Accessoirement, cela permet d'avoir des n-uplets de taille zéro.

Définition 6.16 (`mkvect`) *Soit $\vec{T}$, une liste de termes (on note $|\vec{T}|$ le nombre d'élément qu'elle comporte). Le produit des n premiers éléments de $\vec{T}$ est noté*

$$\mathop{*}_{0\le i}^{i<n} T_i \stackrel{\text{def}}{\equiv} T_0 * \ldots * T_{n-1} * \mathcal{M}^*$$

Pour former le produit de tous les type de $\vec{T}$, on écrira $*\vec{T}$, dont la définition est :

$$*\vec{T} \stackrel{\text{def}}{\equiv} \mathop{*}_{0\le i}^{i<|\vec{T}|} \vec{T}$$

On utilise $\mathcal{M}^*$ pour terminer le vecteur afin d'éviter d'introduire trop d'opérateurs.

Symétriquement, nous définissons le n-uplet formé à partir d'une liste de termes.

Définition 6.17 (`mktuple`) *Le n-uplet associé à la liste de termes $M_1 \dots M_n$ est*

$$(M_1, (M_2, \dots, (M_n, \varnothing) \dots),$$

ce que nous écrirons $(M_1, \dots, M_n)$.

6.4.3 Typage des enregistrements

Le typage des enregistrements est assez complexe. La raison est qu'à un même n-uplet de valeurs, nous pouvons associer différents types. Par exemple, si nous avons $0 : \mathbb{N}$ et $\pi : P(0)$, nous pouvons considérer l'enregistrement ayant un champ x valant 0 et un champ p valant π

- soit comme un couple non dépendant, en lui associant l'arité $\Theta_1 = (x{:}\mathbb{N})(p{:}P(0))$,
- ou bien comme une paire dépendante dont la première composante est un entier x et la deuxième une preuve de $P(x)$ (cet enregistrement code la proposition $\exists x{:}\mathbb{N}.\, P(x)$), en lui associant l'arité $\Theta_2 = (x{:}\mathbb{N})(p{:}P(x))$.

Ces deux types ne sont pas comparables. Si nous voulons que notre système de type admette un type principal, il nous faut lever l'ambiguïté ci-dessus au moment où l'on forme un enregistrement. C'est pourquoi l'argument de l'opérateur $\mathrm{Struct}\{\vec{x}\}$ n'est pas simplement les n valeurs de champs, mais y figure aussi l'arité de l'enregistrement.

Définition 6.18 (`prod_arity, struct_of`) *L'arité des types enregistrement est encodée à l'aide du type suivant :*

$$\begin{aligned} \mathrm{ProdArity}([]) &= \mathcal{M}^* \\ \mathrm{ProdArity}((x{:}T)\Theta) &= \Pi x{:}T.\, \mathrm{ProdArity}(\Theta) \end{aligned}$$

Cette définition permet de définir le terme associé à un enregistrement θ :

$$\mathrm{Struct}\{\theta\} \stackrel{\mathrm{def}}{\equiv} \mathrm{Struct}\{\theta.1\}\,(\mathrm{ProdArity}(\theta), \theta.3)$$

Ceci explique pourquoi nous avons considéré des enregistements à n-champs plutôt que simplement les Σ-types binaires. Les Σ-types binaires doivent eux aussi être annotés avec un type indiquant comment abstraire la valeur de la première composante. Si l'on code un n-uplet à l'aide de n Σ-types emboîtés, la représentation devient excessivement redondante : un enregistrement de longueur trois $(x_1 : T_1 := M_1)(x_2 : T_2 := M_2)(x_3 : T_3 := M_3)$ devrait être représenté par le Σ-type suivant :

$$\langle \Pi x_1 : T_1.\, \Pi x_2 : T_2.\, T_3, M_1, \langle \Pi x_2 : (T_2\{0 \backslash M_1\}).\, (T_3\{1 \backslash M_1\}), M_2, M_3 \rangle \rangle$$

On voit qu'à chaque niveau, on rappelle le type de tous les champs suivants instanciés avec les valeurs des champs précédents. La taille de la structure croit de manière quadratique avec le nombre de champs, ce qui est rapidement intolérable.

L'argument de l'opérateur $\mathrm{Record}\{\vec{x}\}$ est lui aussi un terme caractérisant l'arité de l'enregistrement. Mais nous avons un problème pour déterminer dans quelle sorte vit l'enregistrement. Pour préserver la cohérence, et puisque nous voulons toujours pouvoir définir les projections, il est nécessaire que l'enregistrement soit dans un univers plus élevé que chacune de ses composantes. Or, le codage $\mathrm{ProdArity}(\Theta)$ est toujours dans la sorte Prop car $\mathcal{M}^*$ est typé par Prop, qui est imprédicative. Nous introduisons une autre façon de représenter une arité Θ.

Définition 6.19 (`record_arity, record_of`) *Le deuxième codage d'une arité est le suivant :*

$$\begin{aligned} \mathrm{RecordArity}([]) &= \mathcal{M}^* \\ \mathrm{RecordArity}((x{:}T)\Theta) &= T * \Pi x{:}T.\,\mathrm{RecordArity}(\Theta) \end{aligned}$$

Ainsi, pour une arité Θ, nous associons tout naturellement le type enregistrement suivant :

$$\mathrm{Record}\{\Theta\} \stackrel{\text{def}}{\equiv} \mathrm{Record}\{\Theta.1\}\,(\mathrm{RecordArity}(\Theta))$$

Nous utilisons le fait que $\Pi x{:}T.\,U$ se trouve toujours dans un univers plus grand que celui de U pour montrer que $\mathrm{RecordArity}(\Theta)$ est dans un univers plus grand que chacun des types de champ. Ainsi, la règle de typage des enregistrements sera la suivante :

$$(\textsc{Record})\ \frac{\Gamma \vdash \mathrm{RecordArity}(\Theta) : s \qquad \mathrm{AllDiff}(\Theta.1)}{\Gamma \vdash \mathrm{Record}\{\Theta\} : s}$$

Nous nous assurons aussi que tous les champs ont des noms différents afin que la projection ne soit pas ambiguë. Il n'est pas possible d'imposer la convention que $\mathrm{Field}\{\vec{x}\}\,(M)$ représente le premier champ de nom x. En effet, il ne serait plus possible de représenter la projection correspondant aux éventuels autres champs nommés x, ce qui empêche d'exprimer le type des composantes dépendant de ce champ caché.

Définition 6.20 (`fields_type`) *Pour un enregistrement (un élément de* Mspec*), nous définissons le type de ses composantes (cette fois les types sont tous dans le même contexte, puisque nous substituons les valeurs des champs précédents).*

$$\begin{aligned} \mathrm{FldTyp}_k([]) &= \mathcal{M}^* \\ \mathrm{FldTyp}_k((x{:}T{:=}M)\theta) &= T * ((\mathrm{FldTyp}_{k+1}(\theta))\{0 \setminus \uparrow^k M\}) \end{aligned}$$

Il s'agit en fait du type formé du n-uplet formé à partir des valeurs de chaque champ. Cette fonction sera utile pour définir le type associé à un champ. Mais elle nous permet déjà d'écrire la règle de typage des enregistrements :

$$(\textsc{Struct})\ \frac{\Gamma \vdash \mathrm{ProdArity}(\theta) : s \qquad \Gamma \vdash \theta.3 : \mathrm{FldTyp}_0(\theta)}{\Gamma \vdash \mathrm{Struct}\{\theta\} : \mathrm{Record}\{\theta\}}$$

Définition 6.21 (`expand_sign`) *L'instantiation d'une arité Θ par un terme M (dénotant un enregistrement dont l'arité est Θ) consiste à définir chaque champ x de Θ par* $\mathrm{Field}\{x\}\,(M)$*. Nous noterons cet enregistrement $\Theta + M$.*

$$\overrightarrow{(x_i{:}T_i)} + M \stackrel{\text{def}}{\equiv} \overrightarrow{(x_i{:}T_i{:=}\mathrm{Field}\{x_i\}\,(M))}$$

Nous utilisons la fonction mettant "à plat" les types d'une arité d'enregistrement en subsituant dans les type les valeurs des champs précédents. Comme nous ne disposons que d'une arité Θ (les valeurs des champs sont gardées abstraites), il nous faut l'instancier à l'aide des projections de M. Ainsi, $\mathrm{FldTyp}_0(\Theta + M)$ est le vecteur de types des champs de M. Le règle de typage des projections est donc :

$$(\textsc{Field})\ \frac{\Gamma \vdash M : \mathrm{Record}\{\Theta\} \qquad \Theta.1 = \vec{x_i} \qquad \mathrm{FldTyp}_0(\Theta + M) = *\vec{T_i}}{\Gamma \vdash \mathrm{Field}\{x_k\}\,(M) : T_k}$$

$$(\textsc{Step})\ \frac{<(\Gamma, M, N)}{\Gamma \vdash M\ [<]^T\ N} \qquad (\textsc{Refl})\ \frac{}{\Gamma \vdash T\ [<]^T\ T}$$

$$(\textsc{Prod})\ \frac{\Gamma \vdash A'\ [<]^T\ A \qquad \Gamma\,[x : A'] \vdash B\ [<]^T\ B'}{\Gamma \vdash \Pi x{:}A.\, B\ [<]^T\ \Pi x{:}A'.\, B'}$$

$$(\textsc{Sum})\ \frac{\Gamma \vdash A\ [<]^T\ A' \qquad \Gamma \vdash B\ [<]^T\ B'}{\Gamma \vdash A * B\ [<]^T\ A' * B'}$$

$$(\textsc{Record})\ \frac{\Gamma \vdash \Theta\ [<]^T\ \Theta'}{\Gamma \vdash \mathrm{Record}\{\Theta\}\ [<]^T\ \mathrm{Record}\{\Theta'\}}$$

$$(\textsc{RecArNil})\ \frac{}{\Gamma \vdash [\,]\ [<]^T\ [\,]}$$

$$(\textsc{RecArCons})\ \frac{\Gamma \vdash A\ [<]^T\ A' \qquad \Gamma\,[x : A] \vdash \Theta\ [<]^T\ \Theta'}{\Gamma \vdash (x{:}A)\Theta\ [<]^T\ (x{:}A')\Theta'}$$

FIG. 6.1: Fermeture contextuelle d'une règle de sous-typage $<$

6.4.4 Fermeture contextuelle de types

Nous nous trouvons dans un système avec les opérateurs de types suivants : produit cartésien, produit fonctionnel, enregistrements et types inductifs. Nous allons établir les opérateurs qui sont monotones. Le sous-terme gauche du produit est contravariant (comme pour les PTS). Les deux arguments du produit cartésien sont covariants. L'application n'a pas de variance fixée : sachant que $A \leq A'$, on ne peut a priori rien dire entre $(T\ A)$ et $(T\ A')$ car cela dépend de T. La variance change si T vaut, par exemple $\lambda A : \mathrm{Prop}.\, A$ ou $\lambda A{:}\mathrm{Prop}.\, A \to \bot$.

Définition 6.22 (prédicat `clos_le, clos_record`) *Soit $<$ une relation ternaire. Nous définissons $[<]^T$, la fermeture contextuelle de types, figure 6.1. Noter que les deux dernières règles portent sur des arités d'enregistrement. Formellement, nous définissons deux prédicats mutuellement inductifs.*

Intuitivement, cet opérateur joue le même rôle que l'opérateur de fermeture contextuelle $[R]$, sauf qu'ici, on ne peut appliquer le sous-typage à tous les sous-termes. La réduction servait à identifier des objets, alors que le sous-typage décrit le fait qu'un type est inclus dans un autre, ce qui n'est pas une relation congruente.

6.5 Définitions récursives

6.5.1 Explications informelles

Suivant la discussion section 2.2.4, nous n'autorisons que les expressions de points définis par récurrence structurelle sur un objet inductif.

Il y a plusieurs façons de vérifier la bonne formation d'un point fixe : on peut avoir une condition syntaxique qui permet de vérifier que les appels récursifs sont corrects, ou bien faire cette vérification par typage. C'est la première solution qui est implantée en Coq, et la

règle de typage du point fixe ressemble à :

$$\frac{\Gamma\,[f:T] \vdash M : T \qquad \textsc{Guarded}_f(M)}{\Gamma \vdash \mathrm{Fix}(f.M) : T}$$

La définition de GUARDED est assez compliquée, et ne sera pas présentée ici (voir [23]).

Cette méthode a montré plusieurs inconvénients, dont la plupart ont déjà été remarqués par Gimenez :

- La définition est complexe et pourtant, beaucoup de cas intéressants échappent à cette condition. La raison est qu'il est très difficile de suivre le flot de données syntaxiquement.
- La vérification est peu efficace. Une règle de la condition de garde est qu'on accepte les corps de point fixes qui se réduisent vers un terme bien gardé. Mais pour pouvoir réduire le corps du point fixe, il faut l'avoir typé, pour être sûr qu'il admet une forme normale. Si le terme comporte un grand nombre de points fixes emboîtés, il faut alternativement typer et vérifier la condition de garde.
- La règle mentionnée ci-dessus donne lieu à des termes non fortement normalisables, comme remarqué par Luther et Strecker. En effet, il se peut qu'un corps de point fixe soit bien typé, se réduise vers un terme bien gardé, mais comportant un appel récursif incorrect. Nous donnons un exemple en ML :

```
let rec f x =
  let _ = f x in
    if x=0 then 0 else f (x-1)
```

- Cela s'accorde mal avec la construction interactive de termes, car celle-ci construit les termes par la racine. Ainsi, un raisonnement par récurrence consiste à un introduire un point fixe dont le corps est une métavariable (représentant la preuve du sous-but à résoudre). Le problème est de déterminer si un terme comportant des métavariables est bien gardé, puisque cela dépend des instantiations ultérieures.
- Un problème lié à celui ci-dessus, est que les tactiques automatiques tendent à construire des termes mal gardés : lorsque l'on fait un raisonnement par récurrence, on introduit dans le contexte une hypothèse ayant exactement le même type que le but. Les tactiques automatiques, qui par simplicité ne tiennent pas compte de la condition de garde, résolvent alors le but de façon triviale, en faisant un appel récursif incorrect. Une solution serait de vérifier la condition de garde après chaque application de tactique, mais cela est clairement inefficace. Cette remarque, bien qu'uniquement d'ordre implémentatoire, est probablement l'aspect le plus gênant de la condition de garde.

Une solution évitant d'avoir une condition annexe dans la règle de typage a été proposée par Gimenez [25]. Il semble que cela répare la plupart des problèmes évoqués ci-dessus. Le coût est que l'on aura à compliquer un peu le typage des types inductifs, en introduisant du sous-typage. Ce qui est formalisé ici est en fait une variante de [25].

Le principe du marquage est que pour chaque type inductif I, on introduit deux nouveaux types I^+ et I^-. Lorsque l'on type un point fixe, au lieu d'associer au paramètre récursif le type I, on lui donne le type I^+, et on introduit la fonction permettant de faire les appels récursifs avec un type dont le domaine est I^-. Le typage du filtrage serait tel que les sous-expressions d'un terme de type I^+ auraient le type I^-. Évidemment, à tout moment, nous souhaitons pouvoir utiliser l'argument récursif ou ses sous-termes comme des objets de type I, c'est pourquoi nous introduisons les règles de sous-typage $I^+ \leq I$ et $I^- \leq I$. La règle de

typage serait :

$$\frac{\Gamma\,[f : I^- \to T] \vdash M : I^+ \to T}{\Gamma \vdash \mathrm{Fix}(f.M) : I \to T}$$

Nous rajoutons la règle de sous-typage $I^- \leq I^+$ pour permettre de faire des appels récursifs avec une profondeur supérieure à un, comme dans l'exemple de la division par deux ci-dessous :

```
Fix(div2. [n:nat+]
          Case m of
            O => O
          | S => [k:nat-]
                 Case k of
                   O => O
                 | S => [k':nat-](S (div2 k')))
```

Le terme `k` de type nat^- et d'abord coercé vers le type nat^+, pour pouvoir être destructuré en `k'`.

Mais il y a un problème si on emboîte les points fixes :

```
Fix(f. [n:nat+]
       (Fix (g. [m:nat+]
                Case m of
                  O => O
                | S => [k:nat-](f k (S k)))))
```

Ce programme correspond au programme ML suivant :

```
let rec f n =
  let rec g m =
    match m with
      O -> O
    | (S k) -> f k (S k)
  in g;;
```

La fonction `f` a deux arguments n et m dont il ne faut pas confondre les tailles. Ce terme serait bien typé puisque k a le type nat^-. Mais `(f O (S O))` boucle. Il faut donc une marque qui différencie les différents points fixes.

À chaque point fixe est introduit une variable d'un type particulier (le type des marques, que nous notons $\mathcal{M}$), qui sert à identifier par rapport à quel point fixe la marque $+$ ou $-$ s'applique. Le type des marques est un type comme les autres, ce qui nous évitera, au moment de décrire les règles de réduction du point fixe, d'introduire un nouveau mécanisme pour reloger et substituer les marques.

Pour une marque m, on utilisera les notations I^{+m} et I^{-m}. Dans le corps du point fixe on a non seulement introduit une variable pour faire les appels récursifs, mais aussi une variable pour identifier le point fixe. Lors de l'expansion la variable f sera remplacée par le point fixe, et la marque sera remplacée par la marque vide, puisque les marques n'ont de sens qu'à l'intérieur du point fixe.

De plus, si un argument récursif est appliqué à un point fixe, il faut que dans ce deuxième point fixe, on puisse toujours appeler le premier, puisque l'on sait que cette variable dénotera toujours un sous-terme (au sens large) de notre argument récursif. L'exemple suivant illustre le fait qu'un type inductif peut avoir à être annoté avec plusieurs marques simultanément.

```
Fix(f. [x:nat+f]Case x of
                  O => true
```

```
  | S => [y:nat-f]
       (Fix(g. [y:nat+g-f](andb (Case y of
                                   0 => false
                                 | S => [m:nat-g-f](g m))
                                (f y)))
          k))
```

Dans cet exemple, on utilise le nom du point fixe pour représenter la marque associée à ce point fixe. La variable `y` donne lieu à un appel récursif avec `f`. Il faut donc qu'elle porte la marque `-f`. Quant à `m`, elle porte les marques `-f` et `-g`, car nous savons que c'est un sous-terme strict des arguments récursifs des deux points fixes.

Pour récapituler les remarques faites jusqu'ici, les types inductifs doivent être annotés par des listes de marques, chaque marque pouvant être étiquetée avec `+` ou `-`.

Ensuite, nous devons tenir compte des types dépendants : le type sur lequel on fait la récurrence peut dépendre d'arguments qui varient au cours des appels récursifs. Ainsi, l'argument récursif d'un point fixe ne sera pas le premier, mais le deuxième argument (dans le cas où l'on a plusieurs arguments, il est toujours possible d'utiliser un enregistrement). Nous aboutissons à la règle suivante :

$$\frac{\Gamma\,[m:\mathcal{M}]\,\big[f:\Pi x{:}P.\,\Pi y{:}I^{-m;-L}.\,T\big]\vdash M:\ \Pi x{:}P.\,\Pi y{:}I^{+m;L}.\,T}{\Gamma\vdash \mathrm{Fix}(f.M):\ \Pi x{:}P.\,\Pi y{:}I^{L}.\,T}$$

Enfin, comme nous pouvons définir des types mutuellement inductifs, nous souhaitons pouvoir définir des points fixes mutuels. Nous employons l'astuce habituelle consistant à définir une liste de fonctions récursivement. Toutes ces définitions utiliseront la même marque.

Pour ne pas alourdir cette présentation informelle, nous donnons simplement l'exemple de deux fonctions mutuellement récursives, sans type dépendants. La définition formelle complète se trouve dans la section suivante.

$$\frac{\Gamma\,[m:\mathcal{M}]\,\Big[f:(I_1^{-m;-L_1}\to T_1)*(I_2^{-m;-L_2}\to T_2)\Big]\vdash M:\ (I_1^{+m;L_1}\to T_1)*(I_2^{+m;L_2}\to T_2)}{\Gamma\vdash \mathrm{Fix}(m.f.M):\ (I_1^{L_1}\to T_1)*(I_2^{L_2}\to T_2)}$$

Sémantiquement, on peut considérer qu'une marque est une indication sur la hauteur de l'objet inductif. Si m est une marque, I^{+m} représente l'ensemble des objets de type I de hauteur inférieure ou égale à m, alors que I^{-m} représente ceux de taille strictement inférieure à m. On pourrait avoir un entier plutôt qu'un simple signe pour indiquer la taille relativement à celle d'une marque.

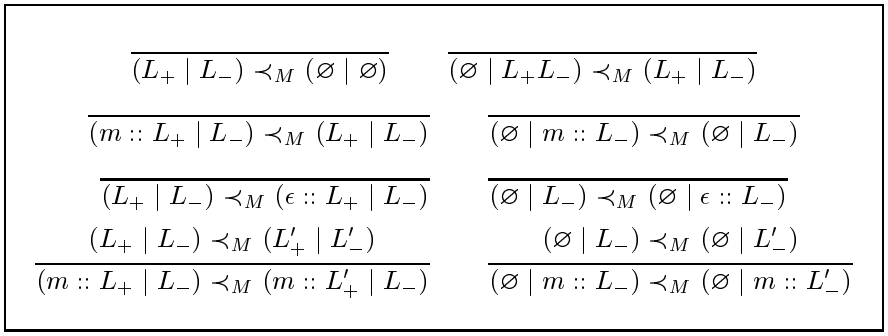
6.5.2 Sous-typage des marques

Pour résumer ce que nous avons vu dans la section précédente, le sous-typage des marques a principalement trois règles : on peut oublier une marque, transférer des marques de L_- vers L_+ (affaiblissement permettant les appels récursifs à une profondeur supérieure à un), ou rajouter des marques vides (ϵ). Les autres règles servent à faire la fermeture réflexive transitive, et à faire ces opérations à n'importe quelle position dans la liste.

Nous emploierons la notation $M :: N$ pour désigner $::((M,N))$.

Définition 6.23 (prédicat `incl_mark`, `ind_marks`) *La règle de sous-typage des marques est la suivante :*

$$\frac{(L_+\mid L_-)\prec_M^*(L'_+\mid L'_-)}{\Gamma\vdash \mathrm{Ind}\{I,|L_+|\}(P,A,L_+L_-)\prec_M \mathrm{Ind}\{I,|L'_+|\}(P,A,L'_+L'_-)}$$



$$\frac{}{(L_+ \mid L_-) \prec_M (\varnothing \mid \varnothing)} \qquad \frac{}{(\varnothing \mid L_+ L_-) \prec_M (L_+ \mid L_-)}$$

$$\frac{}{(m :: L_+ \mid L_-) \prec_M (L_+ \mid L_-)} \qquad \frac{}{(\varnothing \mid m :: L_-) \prec_M (\varnothing \mid L_-)}$$

$$\frac{}{(L_+ \mid L_-) \prec_M (\epsilon :: L_+ \mid L_-)} \qquad \frac{}{(\varnothing \mid L_-) \prec_M (\varnothing \mid \epsilon :: L_-)}$$

$$\frac{(L_+ \mid L_-) \prec_M (L'_+ \mid L'_-)}{(m :: L_+ \mid L_-) \prec_M (m :: L'_+ \mid L_-)} \qquad \frac{(\varnothing \mid L_-) \prec_M (\varnothing \mid L'_-)}{(\varnothing \mid m :: L_-) \prec_M (\varnothing \mid m :: L'_-)}$$

FIG. 6.2: Sous-typage des marques

Au lieu de munir la type inductif de deux listes de marques, nous avons préféré n'avoir qu'une seule liste, ainsi qu'un entier (placé sur l'opérateur), indiquant à quelle profondeur se situe la séparation entre L_+ et L_-. Ce codage, qui n'est pas particulièrement élégant, permet de décrire simplement le passage d'une marque de L_+ vers L_- par simple décrémentation de cet entier.

Nous prouvons quelques résultats élémentaires sur les marques. Ces résultats servent d'interface, ce qui permettra d'avoir des preuves plus robustes face aux changements de représentation des marques.

Lemme 6.24 (`incl_str_marks`)

$$\Gamma \vdash \mathrm{Ind}\{I, 0\}(P, A, m :: L) \prec_M \mathrm{Ind}\{I, n\}(P, A, L)$$

Lemme 6.25 (`incl_add_mt_mark`)

$$\Gamma \vdash \mathrm{Ind}\{I, n\}(P, A, L) \prec_M \mathrm{Ind}\{I, n{+}1\}(P, A, \epsilon :: L)$$

Lemme 6.26 (`incl_mt_highest`) *Les marques plus grandes que $(\varnothing, \varnothing)$ ne contiennent que des marques vides. Nous avons donc le résultat suivant :*

$$\Gamma \vdash \mathrm{Ind}\{I, 0\}(P, A, \varnothing) \prec_M \mathrm{Ind}\{I, n\}(P, A, L)$$

$$\Rightarrow \Gamma \vdash \mathrm{Ind}\{I', 0\}(P', A', \varnothing) \prec_M \mathrm{Ind}\{I', 0\}(P', A', L)$$

6.5.3 Typage des points fixes

Dans cette section, nous définissons formellement les notions caractéristiques des points fixes : comment ils se typent, comment ils se réduisent.

Comme nous l'avons vu dans la règle de typage informelle, les points fixes ont une forme particulière (le domaine est un type inductif, etc.), et se retrouve sous trois formes voisines, suivant le marquage. Nous définissons un enregistrement regroupant toutes les composantes d'un type de point fixe, comme nous l'avons fait pour les enregistrements :

Définition 6.27 (type `fix_info`) *Un descripteur de type de point fixe est une structure*

dont les champs ont les types et les descriptions suivantes :

$$\begin{array}{ll}
\langle\ x : \textsc{Name}; & \textit{(* nom du paramètre *)} \\
\quad P : \textsc{Term}; & \textit{(* type du paramètre *)} \\
\quad y : \textsc{Name}; & \textit{(* nom de l'argument récursif *)} \\
\quad I : \textsc{Name}; & \textit{(* nom du type inductif *)} \\
\quad n : \mathbb{N}; & \textit{(* nombre de marques positives *)} \\
\quad p : \textsc{Term}; & \textit{(* paramètre du type inductif *)} \\
\quad a : \textsc{Term}; & \textit{(* argument du type inductif *)} \\
\quad l : \textsc{Term}; & \textit{(* marques du type inductif *)} \\
\quad T : \textsc{Term} & \textit{(* type du résultat *)} \\
\rangle &
\end{array}$$

Cette structure se comprend mieux grâce à la définition de $\mathcal{FTY}$ ci-dessous.

Définition 6.28 (`lift_finfo, subst_finfo`) *Les opérations de relocation et de substitution s'étendent aux descripteurs de point fixes :*

$$\uparrow_k^n \langle x; P; y; I; n; p; a; l; T\rangle \stackrel{\text{def}}{\equiv} \langle x; \uparrow_k^n P; y; I; n; \uparrow_{k+1}^n p; \uparrow_{k+1}^n a; \uparrow_{k+1}^n l; \uparrow_{k+2}^n T\rangle$$

$$\langle x; P; y; I; n; p; a; l; T\rangle\{k\backslash N\} \stackrel{\text{def}}{\equiv}$$

$$\langle x; P\{k\backslash N\}; y; I; n; p\{k{+}1\backslash N\}; a\{k{+}1\backslash N\}; l\{k{+}1\backslash N\}; T\{k{+}2\backslash N\}\rangle$$

Par rapport au champ P, il y a une variable liée supplémentaire dans p, a et l, et deux dans T.

Les trois définitions suivantes construisent les différentes variations des types de point fixes.

Définition 6.29 (`fix_ty, fix_scheme`) *Soit $F = \langle x; P; y; I; n; p; a; l; T\rangle$ un descripteur de point fixe. Le type du point fixe vu de l'extérieur est défini par :*

$$\mathcal{FTY}(F) \stackrel{\text{def}}{\equiv} \Pi x{:}P.\,\Pi y{:}\mathrm{Ind}\{I, n\}(p, a, l).\,T$$

Pour ce même point fixe, les appels récursifs se font à l'aide d'une fonction dont le type est :

$$\mathcal{FTY}^-(F) \stackrel{\text{def}}{\equiv} \Pi x{:}\uparrow^1 P.\,\Pi y{:}\mathrm{Ind}\{I, 0\}(\uparrow_1^1 p, \uparrow_1^1 a, \natural 1 ::\uparrow_1^1 l).\,\uparrow_2^1 T$$

Enfin, le corps de ce point fixe doit avoir le type :

$$\mathcal{FTY}^+(F) \stackrel{\text{def}}{\equiv} \Pi x{:}\uparrow^1 P.\,\Pi y{:}\mathrm{Ind}\{I, n{+}1\}(\uparrow_1^1 p, \uparrow_1^1 a, \natural 1 ::\uparrow_1^1 l).\,\uparrow_2^1 T$$

Noter que $\mathcal{FTY}^-(F)$ et $\mathcal{FTY}^+(F)$ comportent des relocations d'amplitude 1 car ces termes sont placés dans un contexte où l'on a rajouté la marque caratéristique du point fixe (qui apparaît sous la forme $\natural 1$ dans les listes de marques).

Dans le cas où il n'y a qu'une seule fonction définie récursivement, si le corps du point fixe a le type $\Pi m{:}\mathcal{M}.\,\mathcal{FTY}^-(F) \to \uparrow^1 \mathcal{FTY}^+(F)$ alors le type du point fixe sera $\mathcal{FTY}(F)$.

Si maintenant on considère plusieurs fonctions définies simultanément par récurrence, le corps du point fixe sera une fonction qui prend en argument une marque et un vecteur de fonctions avec lesquelles se feront les appels récursifs, et retourne le vecteur des corps de points fixes.

Définition 6.30 (`fix_bodies`, `fix_types`) *Soit $(F_i)_{0\leq i<n}$ une liste de descripteurs de point fixe. Le type de l'ensemble des corps de points fixes mutuels est :*

$$\mathcal{FBD}(m,\vec{F}) \stackrel{\text{def}}{\equiv} \Pi m{:}\mathcal{M}.\ \overset{i<n}{\underset{0\leq i}{*}} \mathcal{FTY}^{-}(F_i) \to \uparrow^1 \overset{i<n}{\underset{0\leq i}{*}} \mathcal{FTY}^{+}(F_i)$$

Et le type de l'ensemble des points fixes est :

$$\mathcal{FTYS}(\vec{F}) \stackrel{\text{def}}{\equiv} \overset{i<n}{\underset{0\leq i}{*}} \mathcal{FTY}(F_i)$$

Ainsi, le corps d'un ensemble de points fixes mutuels doit être une fonction prenant en argument une marque et un vecteur de fonctions avec lesquels on fera les appels récursifs, et retournant le vecteur de fonctions correspondant à une itération supplémentaire.

Lemme 6.31 *Toutes ces définitions sont stables vis-à-vis des indices de de Bruijn (stabilité par relocation et substitution).*

Avec ces définitions, on peut dores et déjà exprimer la règle de typage du point fixe (elle sera rappelée figure 6.4).

$$(\textsc{Fix})\ \frac{\Gamma \vdash \mathcal{FTYS}(\vec{F}) :\ s \qquad \Gamma \vdash B :\ \mathcal{FBD}(\vec{F})}{\Gamma \vdash \mathrm{Fix}(\mathcal{FTYS}(\vec{F}), B) :\ \mathcal{FTYS}(\vec{F})}$$

Pour permettre la décidabilité du typage dans de bonnes conditions, le point fixe doit être annoté par les types des fonctions définies.

Pour en terminer avec les points fixes, nous montrons deux propriétés des définitions ci-dessus qui garantissent que la règle de réduction des points fixes est correcte. Nous définissons la réduction de point fixe non gardée car elle est plus simple, et possède la plupart des bonnes propriétés attendues, et c'est un sur-ensemble de la réduction gardée. Cette règle de réduction n'est pas intégrée dans le système telle quelle, puisqu'elle briserait la normalisation forte du calcul.

Définition 6.32 (prédicat `fix_free`) *L'expansion de point fixe non gardée par un constructeur est définie par :*

$$\frac{}{\Gamma \vdash \mathrm{Fix}(T,f) \to (f\ \epsilon\ \mathrm{Fix}(T,f))}\ (\iota\text{-}\textsc{FixFree})$$

Pour assurer la correction de cette règle, il faut vérifier deux choses. D'une part, le point fixe, une fois expansé, doit avoir un type inclus dans celui du point fixe initial :

Lemme 6.33 (`fix_body_incl_types`) *Le type du corps du point fixe (dans lequel on a effacé la marque) est plus petit que le type du point fixe vu de l'extérieur.*

$$\Gamma \vdash \overset{i<n}{\underset{0\leq i}{*}} \mathcal{FTY}^{+}(F_i)\{0\backslash\epsilon\}\ [\prec_M]^T\ \mathcal{FTYS}(\vec{F})$$

D'autre part, le type du deuxième argument du corps de point fixe (la marque étant une fois encore instanciée par la marque vide) doit être plus grand que le type du point fixe :

Lemme 6.34 (`fix_exp_incl`) *Le type du point fixe est plus petit que le type de l'argument du corps de point fixe.*

$$\Gamma \vdash \mathcal{FTYS}(\vec{F})\ [\prec_M]^T \overset{i<n}{\underset{0\leq i}{*}} \mathcal{FTY}^{-}(F_i)\{0\backslash\epsilon\}$$

Il est donc correct d'appliquer le point fixe à son corps dans lequel la marque est instanciée par la marque vide.

Enfin, pour tout terme M, il est décidable d'inférer s'il existe un F tel que $M = \mathcal{FTY}(F)$, ce qui permet de décider de la condition annexe (i.e. la première) de la règle de typage du point fixe.

Lemme 6.35 (`is_fix_types_dec`)

$$\forall M.\, \mathsf{DecFind}\left(\vec{F} \;\middle|\; M = \mathcal{FTYS}(\vec{F})\right)$$

6.6 Filtrage

Le typage du filtrage est fortement lié a celui du point fixe, car c'est le filtrage qui assure que l'on fait des appels récursifs avec un terme plus petit, en annotant les sous-termes d'un objet inductif de type I^{+m} avec la marque $-m$.

Le point fixe introduit une marque sur la variable représentant l'argument récursif. Ensuite, le filtrage doit propager les marques de façon à ce que les sous-termes avec lesquels on a le droit de faire des appels récursifs soient marqués. Enfin, la fonction avec laquelle on fait l'appel récursif est marquée de façon à n'accepter que les termes qui sont des sous-termes stricts de l'argument récursif initial.

On annote le case avec le prédicat indiquant le type du résultat. Cette information ne peut évidemment pas être inférée à partir du type des branches puisque dans le cas où l'on destructure un type inductif avec zéro constructeurs, il y aura zéro branches. Même dans les cas plus courants où il y a des branches, il faudrait faire du filtrage d'ordre supérieur pour la retrouver (avec l'élimination dépendante, les différentes branches peuvent ne pas avoir le même type). Ce n'est pas décidable, et la solution pourrait ne pas être unique, ce qui nous empêche d'inférer un type principal.

Dans le système Coq, l'opérateur de filtrage est assez rudimentaire : il faut fournir une branche pour chaque constructeur, et il faut écrire les branches en respectant l'ordre des constructeurs dans la déclaration du type inductif. Les sous-termes de chaque constructeur étant introduits par abstraction. Il est beaucoup plus lisible d'avoir, comme en ML, un filtrage avec des motifs, en ayant toute liberté quand à l'ordre dans lequel on présente les branches.

6.6.1 Typage de l'opérateur de filtrage

Typage du prédicat

Nous avons qu'en présence de types dépendants, les branches d'un filtrage pouvaient avoir des types différents. En effet, le fait qu'une certain branche a été choisie nous donne une information sur la forme de l'objet destructuré.

Le type des objets produits par filtrage n'est pas simplement un type, mais plutôt un prédicat dont le type a une forme particulière :

Définition 6.36 (`case_predicate_form`) *Pour toute spécification de type inductif Φ, le type du prédicat de filtrage vers la sorte s_e est :*

$$\mathsf{Cpf}(\Phi, s_e) \stackrel{\text{def}}{\equiv} \Pi\_ : (\Phi.\mathrm{A}).\, \mathrm{Ind}(\Phi.\mathrm{X}, \natural 1, \natural 0) \to s_e$$

Remarque : on ne considère que des éliminations dépendantes. Pour une élimination non dépendante, il suffit que le prédicat ignore son deuxième argument :

$$\mathsf{Cpf}(\Phi, s_e) \stackrel{\text{def}}{\equiv} \Phi.\mathrm{A} \to s_e$$

Nous devons restreindre les classes de types des objets construits par filtrage (caractérisée par la sorte s_e). Cela peut être utile pour éviter des paradoxes. Par exemple, dans Coq, dans le cas des inductifs imprédicatifs (lorsque l'univers des arguments des constructeurs est plus haut que celui de l'inductif), il faut restreindre l'élimination à Prop. Cela peut servir aussi à garantir la possibilité d'extraire le contenu calculatoire (distinction Set/Prop de Coq).

Définition 6.37 (prédicat `elim_sort`) *Les règles d'élimination autorisées sont ($i \leq j$) :*

$$(s, \mathrm{Prop}, \mathrm{Prop}) \in \mathrm{ElimSort} \qquad (\mathrm{Prop}^\epsilon, \mathrm{Set}, s) \in \mathrm{ElimSort}$$
$$(\mathrm{Type}(i), \mathrm{Set}, \mathrm{Prop}^\epsilon) \in \mathrm{ElimSort}$$
$$(\mathrm{Prop}^\epsilon, \mathrm{Type}(i), s) \in \mathrm{ElimSort} \qquad (\mathrm{Type}(i), \mathrm{Type}(j), s) \in \mathrm{ElimSort}$$

Le premier argument est la sorte dans laquelle vivent les types de constructeurs, le deuxième est la sorte du type inductif, et le troisième la sorte du type des objets que l'on souhaite construire par filtrage. Ainsi, pour une définition inductive Φ, les éliminations autorisées seront les sortes s_e telles que $(\Phi.\mathrm{CK}, \Phi.\mathrm{S}, s_e) \in \mathrm{ElimSort}$.

Noter que Prop permet moins d'éliminations que Set, car on souhaiterait que le principe de non-pertinence des preuves (*proof-irrelevance*) soit cohérent (ce qui permet d'avoir le tiers-exclu dans Prop). Aussi, les types inductifs dans Prop ne peuvent pas être éliminés vers Set car on voudrait l'élimination forte dans Set, et une élimimnation vers Set ferait que le tiers exclu dans Prop se propagerait dans Set. Or la non-pertinence des preuves et l'élimination forte se contredisent de manière évidente.

Typage des branches

On considère une liste de noms ρ : il s'agit de la liste des noms de types inductifs définis mutuellement. En effet, la propagation des marques s'étend à tous les objets inductifs définis simultanément.

Les prochaines définitions vont permettre de typer les branches associées à chacun des motifs.

Pour simplifier, pour un prédicat d'élimination Q, et une branche correspondant à un constructeur C dont le type des arguments est A, le type de la branche sera $\Pi x\!:\!A.\ (Q\ C(x))$. Il faut tenir compte du fait que le prédicat prend comme premier argument la valeur des arguments de type inductif correspondant à son deuxième argument, et le fait que l'on marque les sous-termes structurellement plus petits que l'objet destructuré (pour permettre les appels récursifs). Cela donnerait le type $\Pi x : A^-.\ (Q\ \kappa_{\mathrm{I}}\ C(x))$. C'est ce que fera la définition de $\mathcal{BTY}$ ci-dessous.

Nous allons décomposer cette définition, en commençant par l'opération de marquage du type des arguments de constructeurs.

Définition 6.38 (prédicat `str_substp`, `mark_record`) *Le marquage des types inductifs ρ avec la liste de marques L dans le terme M est M', ce qui sera noté $M \overset{\rho,L}{\gg} M'$, est défini*

par les règles suivantes :

$$\frac{}{M \overset{\rho,L}{\gg} M} \qquad \frac{I \in \rho}{\mathrm{Ind}(I,P,A) \overset{\rho,L}{\gg} \mathrm{Ind}\{I,0\}(P,A,L)}$$

$$\frac{U \overset{\rho,\uparrow^1 L}{\gg} U'}{\Pi x{:}T.\,U \overset{\rho,L}{\gg} \Pi x{:}T.\,U'} \qquad \frac{T \overset{\rho,L}{\gg} T' \qquad U \overset{\rho,L}{\gg} U'}{T * U \overset{\rho,L}{\gg} T' * U'}$$

$$\frac{\Theta \overset{\rho,L}{\gg} \Theta'}{\mathrm{Record}\{\Theta\} \overset{\rho,L}{\gg} \mathrm{Record}\{\Theta'\}} \qquad \frac{}{[] \overset{\rho,L}{\gg} []} \qquad \frac{A \overset{\rho,L}{\gg} A' \qquad \Theta \overset{\rho,\uparrow^1 L}{\gg} \Theta'}{(x,A)\Theta \overset{\rho,L}{\gg} (x,A')\Theta'}$$

Comme pour la définition de le fermeture contextuelle de type, nous définissons deux prédicats mutuellements inductifs pour tenir compte des types enregistrement.

Cette définition donne le type des arguments des branches du case. C'est ici qu'on choisit à quels arguments se propagent les marques. On ne marque que les occurrences strictement positives.

Afin d'éviter que le point fixe ne boucle, n'importe quel argument de constructeur ne peut hériter de la marque. Ceci est dû à l'imprédicativité (voir l'exemple de Coquand dans [12]). Il y a une notion d'*argument de constructeur récursif*. Dans la présentation actuelle, il s'agit de ne marquer que les arguments appartenant à l'un des types définis simultanément avec le type inductif déstructuré.

Par comodité, le marquage n'est pas fonctionnel (un même terme peut se marquer de différentes manières), pour ne pas avoir à énumérer tous les cas où la marque ne se propage pas. Mais le seul marquage qui nous intéresse est celui qui rajoute les marques dans le plus grand nombre d'arguments, puisqu'il sera toujours possible d'effacer les marques par sous-typage, comme le montre le résultat suivant.

Lemme 6.39 (`str_sub_smaller`) *Un type de branche marqué est un sous-type du type de branche non marqué :*

$$A \overset{\rho,L}{\gg} B \;\Rightarrow\; \Gamma \vdash B\ [\prec_M]^T\ A$$

Ce résultat tient car nous ne propageons les marques que dans des sous-termes covariants (voir la définition 6.22).

Lemme 6.40 (`incl_arg_str_branch`) *Le résultat-clé qui permettra de prouver l'auto-réduction du filtrage est :*

$$A \overset{\rho,\uparrow^1 L}{\gg} B \;\wedge\; \Gamma \vdash \mathrm{Ind}(I,p,a)\ [\prec_M]^T\ \mathrm{Ind}\{I',n\}(p',a',L) \;\Rightarrow\; \Gamma\,[x:T] \vdash A\ [\prec_M]^T\ B$$

Preuve Remarquons que L ne peut être qu'une liste de marques vides. Il s'ensuit très facilement que A est inclus dans B, car il suffit de rajouter dans A des marques vides. ■

Le type proprement dit d'une branche de filtrage est défini par la fonction suivante (en supposant que A est le type des arguments du constructeur dans lequel on a déjà fait le marquage des arguments récursifs).

Définition 6.41 (`branch_ty`) *Type des branches pour un constructeur κ dont les arguments récursifs sont marqués dans A et pour un prédicat Q :*

$$\mathcal{BTY}(\kappa,A,Q) \overset{\mathrm{def}}{\equiv} \Pi\_ : A.\,(\uparrow^2 Q\ \kappa.\mathrm{I}\ \mathrm{Constr}\{\kappa.\mathrm{X}\}\ (\natural 1, \natural 0))$$

Si κ est un constructeur de Φ, Q doit être typé par $\mathsf{Cpf}(\Phi, s_e)$ pour une sorte s_e. Et comme A doit être un sous-type de $\kappa.\text{A}$, le terme ci-dessus sera bien typé.

On généralise cela au vecteur de motifs. Pour chaque motif, on va chercher dans la liste $\vec{\kappa}$ des constructeurs le constructeur correspondant au motif, puis on marque le type des arguments avec L, et enfin on applique $\mathcal{BTY}$:

Définition 6.42 (prédicat `case_branch`) *Correspondance entre motif et type de branche : à chaque motif m_i, il doit correspondre un constructeur κ ($\kappa.\text{X} = m_i$), dont le type des arguments est marqué avec L en A. B_i doit être égal au type de branche correspondant au prédicat Q.*

$$\text{CaseBranch}(\rho, \Phi, Q, L, \vec{m}, \vec{B}) \stackrel{\text{def}}{\equiv}$$

$$|\vec{m}| = |\vec{B}| \ \wedge \ \forall i. \exists \kappa \in (\Phi.\text{C}). \exists A. \, \kappa.\text{X} = m_i \ \wedge \ \kappa.\text{A} \overset{\rho,L}{\gg} A \ \wedge \ B_i = \mathcal{BTY}(\kappa, A, Q)$$

Cette relation garantit que chaque branche correspond bien à un constructeur du type inductif.

Exaustivité des motifs

Il reste à vérifier qu'il y a un motif pour chacun des constructeurs, i.e. le motif de filtrage est exaustif. Cette condition assure que pour tout terme canonique du type inductif (donc avec un constructeur en tête du terme), il est possible de réduire le filtrage.

Un filtrage non exaustif reviendrait à avoir une fonction partielle (définie uniquement pour les valeurs correspondant aux motifs présents), et nous avons montré comment construire un paradoxe à partir d'une fonction partielle.

Définition 6.43 (prédicat `full_patt`) *Soit Φ un descripteur de type inductif et $\vec{p}$ une liste de motifs de filtrage. Celle-ci est dite exaustive pour le type inductif Φ si pour tout constructeur de Φ, il existe un motif de $\vec{p}$ qui le filtre :*

$$\text{Full}(\Phi, \vec{p}) \stackrel{\text{def}}{\equiv} \exists \vec{\kappa} = \Phi.\text{C}. \forall i. \, \kappa_i.\text{X} \in \vec{p}$$

Typage du filtrage

Habituellement, dans les compilateurs ML, il y a une vérification pour s'assurer que toutes les branches d'un filtrage sont utiles, ce qui signifie qu'il existe une valeur pour laquelle la réduction du filtrage se fera en utilisant cette branche. L'existence d'une branche inutilisée est la plupart du temps le signe que l'utilisateur a commis une erreur.

Avec les types dépendants, il est complexe de déterminer si une branche est utile ou pas. Considérons le type des vecteurs (listes dont le type est annoté avec la longueur) :

```
Coq < Inductive vect [A :Set] : nat->Set :=
Coq <   Vnil : (vect A 0)
Coq < | Vcons : (n :nat)A->(vect A n)->(vect A (S n)).
```

Seul le vecteur nul `Vnil` est de longueur 0. Il ne devrait pas être nécessaire de fournir une branche pour le cas `Vcons` lorsque l'on filtre un objet de type `(Vect A 0)`.

Nous ne ferons donc aucune vérification. En particulier, deux branches peuvent correspondre au même motif. Dans ce cas, c'est la règle du premier motif qui s'applique.

On peut maintenant écrire la règle de typage du filtrage :

$$(\textsc{Case})\ \frac{\begin{array}{c}\Gamma \vdash M : \mathrm{Ind}\{I,n\}(P,A,L) \qquad I \rhd_\Sigma \mathsf{Ind}(\rho,\Phi) \\ \Gamma \vdash Q : \mathsf{Cpf}(\Phi,s_e)\{0\backslash P\} \qquad \mathrm{ElimSort}(\Phi.\textsc{ck},\Phi.\textsc{s},s_e) \qquad \mathrm{Full}(\Phi,\vec{m}) \\ \Gamma \vdash B : (*\vec{V})\{0\backslash P\} \qquad \mathrm{CaseBranch}(\rho,\Phi,Q,L,\vec{m},\vec{V})\end{array}}{\Gamma \vdash \langle Q\rangle \mathrm{Cases}\ M\ \mathrm{of}\ \vec{m} \Rightarrow B\ \mathrm{end} : (Q\ A\ M)}$$

Important : le case ne peut se typer que si le type inductif n'est pas abstrait car on fait appel à $\Phi.\textsc{c}$ (La condition Full nous l'assure).

6.6.2 Extensions possibles des motifs de filtrage

On peut imaginer un bon nombre d'améliorations : le motif attrape-tout "_" qui permettra de factoriser des cas. C'est un vrai progrès : non seulement c'est plus commode à écrire, mais la représentation interne aussi est plus concise. Dans ce cas, l'élimination ne peut pas être dépendante. Ce qui ne veut pas dire que le autres branches ne seront pas dépendantes. En reprenant les notations de la règle de typage ci-dessus, la branche associée au motif attrape-tout devrait avoir le type $(Q\ A\ M)$.

Comme autre extension, on peut continuer les factorisations en autorisant les disjonctions de motifs (comme le $m_1 \mid m_2$ de Objective Caml). Ici encore, on ne peut plus faire d'élimination dépendante, puisque la forme exacte du terme déstructuré M n'est pas connue.

Nous pourrions autoriser plus de souplesse dans la conversion des branches de filtrage. Deux filtrages dont les branches ne sont pas dans le même ordre ne sont pas convertibles. Par exemple, Les deux termes suivants ne sont pas des termes convertibles :

$$\langle Q\rangle \mathrm{Cases}\ M\ \mathrm{of}\ \mathrm{O} \Rightarrow f_O \mid \mathrm{S} \Rightarrow \lambda k{:}\mathrm{nat}.\ f_S\ \mathrm{end}$$

$$\langle Q\rangle \mathrm{Cases}\ M\ \mathrm{of}\ \mathrm{S} \Rightarrow \lambda k{:}\mathrm{nat}.\ f_S \mid \mathrm{O} \Rightarrow f_O\ \mathrm{end}$$

Nous relativiserons cet inconvénient en disant qu'il est toujours possible de faire comme avec le Case de Coq en respectant toujours le même ordre pour les constructeurs. D'autre part, dans la grande majorité des cas, lorsque deux filtrages sont convertibles, c'est qu'ils ont une origine commune, et les branches seront dans le même ordre.

En dernier ressort, il est possible de prouver que les deux termes ci-dessus sont égaux en faisant un raisonnement par cas sur M.

6.7 Définition du sous-typage

6.7.1 Règles de réduction

Les réductions considérées sont les suivantes : projections, β-réduction, δ-réduction et ι-réduction ; cette dernière se compose de la réduction du filtrage, ainsi que l'expansion du point fixe.

Définition 6.44 (prédicat `redn_term`) *La règle de réduction de CCI est la réunion des huit règles de la figure 6.3. La règle d'expansion des points fixes est gardée par un constructeur.*

Bien que les règles de typages des enregistrements que nous avons présenté n'autorisaient pas d'enregistrements ayant plusieurs champs avec le même nom, nous vérifions tout de même que l'on choisi le premier champ portant le nom de la projection pour assurer le résultat de confluence sur les termes non typés.

$$\frac{}{\Gamma \vdash (\lambda x{:}T.\ M\ \ N) \to_{\mathrm{CCI}} M\{0\backslash N\}}\ (\beta)$$

$$\frac{}{\Gamma \vdash \pi_1(M,\ N) \to_{\mathrm{CCI}} M} \qquad \frac{}{\Gamma \vdash \pi_2(M,\ N) \to_{\mathrm{CCI}} N}\ (\pi)$$

$$\frac{y = x_n \qquad \forall k{<}n.\ y \neq x_k}{\Gamma \vdash \mathrm{Field}\{y\}\ (\mathrm{Struct}\{\vec{x}\}\ (T,\ M)) \to_{\mathrm{CCI}} \pi_2(\pi_1^n(M))}\ (\pi_R)$$

$$\frac{\Gamma(n) = [x{=}M{:}T]}{\Gamma \vdash \natural n \to_{\mathrm{CCI}} \uparrow^{n+1} M}\ (\delta) \qquad \frac{x \rhd_\Sigma \mathsf{Cst}\,\langle T; [x{=}M{:}U]\rangle}{\Gamma \vdash \mathrm{Cst}\{x\}(N) \to_{\mathrm{CCI}} M\{0\backslash N\}}\ (\delta\text{-G})$$

$$\frac{m_n = C \qquad \forall i < n.\ m_i \neq C}{\Gamma \vdash \langle Q\rangle \mathrm{Cases}\ \mathrm{Constr}\{C\}\ (P, A)\ \mathrm{of}\ \vec{m} \Rightarrow B\ \mathrm{end} \to_{\mathrm{CCI}} (\pi_1(\pi_2^n(B))\ \ A)}\ (\iota\text{-Case})$$

$$\frac{c = \mathrm{Constr}\{C\}\ (p, a)}{\Gamma \vdash (\pi_1(\pi_2^n(\mathrm{Fix}(T, f)))\ \ m\ \ c) \to_{\mathrm{CCI}} (\pi_1(\pi_2^n((f\ \epsilon\ \mathrm{Fix}(T, f))))\ \ m\ \ c)}\ (\iota\text{-Fix})$$

FIG. 6.3: Règles de réduction de CCI

Nous définissons un prédicat d'inversion de la réduction. Il permet de montrer la correction de l'algorithme de conversion à partir de celle de l'algorithme de mise en forme normale de tête.

Définition 6.45 (prédicat `redn_term_hn_inv`) *Prédicat d'inversion de la réduction :*

$$\frac{}{\Gamma \vdash s \to^{\mathrm{inv}}_{\mathrm{CCI}} s}\ (s \in \mathrm{Sort}) \qquad \frac{}{\Gamma \vdash \natural n \to^{\mathrm{inv}}_{\mathrm{CCI}} \natural n}$$

$$\frac{}{\Gamma \vdash c \to^{\mathrm{inv}}_{\mathrm{CCI}} c}\ (c \in \mathrm{Oper}) \qquad \frac{\Gamma \vdash M \rhd^*_{\mathrm{CCI}} M'}{\Gamma \vdash c(M) \to^{\mathrm{inv}}_{\mathrm{CCI}} c(M')}$$

$$\frac{\Gamma \vdash A \rhd^*_{\mathrm{CCI}} A' \qquad \Gamma\,[x : A] \vdash B \rhd^*_{\mathrm{CCI}} B'}{\Gamma \vdash \Pi x{:}A.\ B \to^{\mathrm{inv}}_{\mathrm{CCI}} \Pi x{:}A'.\ B'}$$

$$\frac{\Gamma \vdash A \rhd^*_{\mathrm{CCI}} A' \qquad \Gamma\,[x : A'] \vdash M \rhd^*_{\mathrm{CCI}} M'}{\Gamma \vdash \lambda x{:}A.\ M \to^{\mathrm{inv}}_{\mathrm{CCI}} \lambda x{:}A'.\ M'}$$

$$\frac{\Gamma \vdash A \rhd^*_{\mathrm{CCI}} A' \qquad \Gamma \vdash B \rhd^*_{\mathrm{CCI}} B'}{\Gamma \vdash A * B \to^{\mathrm{inv}}_{\mathrm{CCI}} A' * B'} \qquad \frac{\Gamma \vdash M \rhd^*_{\mathrm{CCI}} M' \qquad \Gamma \vdash N \rhd^*_{\mathrm{CCI}} N'}{\Gamma \vdash (M,\ N) \to^{\mathrm{inv}}_{\mathrm{CCI}} (M',\ N')}$$

Lemme 6.46 (`confluent_redn`) *Nous admettons la confluence (faible) de le règle de réduction* $\to_{\mathrm{CCI}}$.

Ce résultat ne tient que dans le cas où la signature Σ ne définit pas plusieurs fois une constante C avec des valeurs différentes. Nous verrons plus loin la condition exacte.

Algorithme 6.47 (`whnf_cci`) *Nous admettons aussi qu'il existe un algorithme de mise en forme normale de tête des termes fortement normalisables.*

6.7.2 Conversion

La conversion n'est pas simplement la fermeture réflexive symétrique transitive et congruente de la réduction : on veut tenir compte de l'α-conversion. Celle-ci ne peut pas être

incluse dans la réduction sous sa forme habituelle, car cela donnerait lieu à une réduction qui ne serait pas normalisante. C'est le même genre de problème que l'on rencontre avec la symétrie dans les systèmes de réécriture associatifs et commutatifs (on peut se référer à [34]).

Définition 6.48 (prédicat `alpha`) *L'α-conversion s'applique aux produits et aux abstractions.*

$$\frac{}{\Gamma \vdash \Pi x{:}T.\, U \to_\alpha \Pi y{:}T.\, U} \qquad \frac{}{\Gamma \vdash \lambda x{:}T.\, M \to_\alpha \lambda y{:}T.\, M}$$

L'α-réduction est symétrique.

Définition 6.49 (prédicat `eq_term`) *La convertibilité sur les termes de CCI est la fermeture réflexive, symétrique, transitive et congruente de la règle de réduction $\to_{\mathrm{CCI}}$, et de l'α-conversion. La convertibilité entre M et N dans le contexte Γ se note $\Gamma \vdash M \equiv_{\mathrm{CCI}} N$.*

$$\underset{\mathrm{CCI}}{\equiv} \stackrel{\mathrm{def}}{\equiv} \left[\underset{\mathrm{CCI}}{\to} \cup \to_\alpha\right]^=$$

Lemme 6.50 (`eq_subtyping_rule`) *La relation de conversion est une règle de sous-typage.*

$$\underset{\mathrm{CCI}}{\equiv} \in \textsc{SubRule}$$

Nous pouvons même prouver un résultat de stabilité plus général que celui requis pour former une règle de sous-typage (notamment la stabilité par changement de noms dans le contexte).

Définition 6.51 (`R_stable_eq_special`)

$$\forall R.\ \underset{\mathrm{CCI}}{\equiv} \in \mathsf{Rstable}_{\mathrm{dclsub}(=,R)}$$

Nous définissons le prédicat d'inversion de la conversion sur le même modèle que pour la réduction, et on montre qu'il est équivalent à $\equiv_{\mathrm{CCI}}$ sur l'ensemble des formes normales de tête. La difficulté supplémentaire est qu'il faut tenir compte de l'α-conversion qui n'apparaît pas dans la réduction, mais uniquement dans la conversion.

Nous faisons l'hypothèse supplémentaire que la réduction commute avec l'α-réduction, ce qui permet de déduire le résultat de confluence, et donc la propriété de Church-Rosser :

Lemme 6.52 (`CR`) *Si M est convertible avec N, alors il existe deux réduits M' et N' égaux modulo α-conversion :*

$$\Gamma \vdash M \underset{\mathrm{CCI}}{\equiv} N \ \Rightarrow\ \exists M', N'.\ \begin{cases} \Gamma \vdash M \triangleright^*_{\mathrm{CCI}} M' \\ \Gamma \vdash N \triangleright^*_{\mathrm{CCI}} N' \\ \Gamma \vdash M' \triangleright^*_{\alpha} N' \end{cases}$$

Ce résultat possède de nombreux corollaires : propriété de Church-Rosser, inversion de la conversion des paires :

Lemme 6.53 (`inv_pair_l`, `inv_pair_r`)

$$\Gamma \vdash (M,\, N) \underset{\mathrm{CCI}}{\equiv} (M',\, N') \ \Rightarrow\ \Gamma \vdash M \underset{\mathrm{CCI}}{\equiv} M' \ \wedge\ \Gamma \vdash N \underset{\mathrm{CCI}}{\equiv} N'$$

Enfin, nous pouvons montrer la décidabilité de la conversion sur les termes fortement normalisables.

Algorithme 6.54 (`convert`) *La convertibilité de deux termes fortement normalisables est décidable :*

$$\forall M, N \in \mathcal{SN}_\Gamma^{\rightarrow_{\mathrm{CCI}}}.\ \mathsf{Dec}\left(\Gamma \vdash M \underset{\mathrm{CCI}}{\equiv} N\right)$$

Preuve Nous commençons par prouver la décidabilité du prédicat d'inversion de la conversion, disposant d'un algorithme de conversion pour les sous-termes.

Puis, sachant que ce prédicat est équivalent à la convertibilité sur l'ensemble des formes normales de têtes, nous en déduisons un algorithme de conversion, si nous disposons d'un algorithme de conversion pour les sous-termes d'un réduit.

Enfin, il suffit de construire le point fixe de l'algorithme ci-dessus. La terminaison de ce point fixe est assurée par la bonne fondation de l'ordre de normalisation (que nous définissons de manière similaire au cas des PTS). ■

6.7.3 Règles de sous-typage

Sous-typage de CCI

Le sous-typage comporte deux notions : la cumulativité et l'affaiblissement des marques des types inductifs. Le sous-typage des marques a été défini dans la section sur les points fixes.

Définition 6.55 (prédicat `cci_step`**)**

$$\frac{s_1 \prec^{\mathrm{V6}} s_2}{\Gamma \vdash s_1 <^{\mathrm{CCI}} s_2} \qquad \frac{\Gamma \vdash M \prec_M N}{\Gamma \vdash M <^{\mathrm{CCI}} N}$$

Enfin, la relation $\leq^{\mathrm{CCI}}$ est la fermeture réflexive transitive de la réunion de l'égalité sur les termes et de la fermeture aux contextes de types de la relation vue ci-dessus.

Définition 6.56 (prédicat `cci_subtype`**)** *La relation de sous-typage de CCI est définie par :*

$$\leq^{\mathrm{CCI}} \stackrel{\mathrm{def}}{\equiv} ([<^{\mathrm{CCI}}]^T)^*$$

Lemme 6.57 (`cci_subtyping_rule`) *La relation de sous-typage de CCI vérifie bien les propriétés attendues pour une règle de sous-typage.*

$$\leq^{\mathrm{CCI}} \in \textsc{SubRule}$$

Comme pour la convertibilité, nous avons un résultat de stabilité plus fort que celui des règles de sous-typage.

Lemme 6.58 (`subtype_stable_special`) *La relation de sous-typage ne dépend ni des types, ni des noms figurant dans le contexte.*

$$\forall R.\ \ \leq^{\mathrm{CCI}} \in \mathsf{Rstable}_{\mathrm{dclsub}(=,R)}$$

Définition 6.59 (prédicat `sub_hn_inv`**)** *Le prédicat d'inversion du sous-typage : facilite la démonstration de la décidabilité du sous-typage.*

Résultat d'inversion du produit (et sa contrepartie pour le produit cartésien : $\Gamma \vdash A * B \leq^{\mathrm{CCI}} A' * B'$ implique $\Gamma \vdash A \leq^{\mathrm{CCI}} A'$ et $\Gamma \vdash B \leq^{\mathrm{CCI}} B'$, etc.).

Et enfin, on prouve la décidabilité du sous-typage sur les termes fortement normalisables.

Algorithme 6.60 (`subtype_dec`)

$$\forall \Gamma.\, \forall A, B \in \mathcal{SN}_\Gamma^{\rightarrow_{\mathrm{CCI}}}.\ \mathsf{Dec}\left(\Gamma \vdash A \leq^{\mathrm{CCI}} B\right)$$

Ce résultat est admis (rien de vraiment complexe une fois établi l'équivalenve entre le sous-typage et son prédicat d'inversion sur les formes normales de têtes).

6.8 Typage des opérateurs

Définition 6.61 (prédicats `mem_sign0, mem_sign`) *Les règles de typage sont présentées figure 6.4. Formellement, on définit deux relations Σ_0^{CCI} et Σ_1^{CCI}. La figure 6.5 montre les définitions formelles correspondant aux deux premiers tableaux de la figure 6.4.*

En fait, Σ_1^{CCI} ne dépend pas du contexte Γ.

Les règles du premier cadre définissent le typage des constructions élémentaires : fonctions, paires et définition globales. Le deuxième correspond au système de marques. Quant au troisième, il introduit les types inductifs et les points fixes. Enfin, le quatrième tableau présente les règles de bonne formation, d'introduction et d'élimination des enregistrements.

Concernant le typage de la signature : dans cette section on établira quelques propriétés que doit vérifier un environnement bien formé. Puis, dans le chapitre suivant, on définira un système d'inférence qui impliquera ces propriétés.

Lemme 6.62 (`is_signature1`) *La signature vérifie toutes les conditions de stabilité requise pour pouvoir former le PTS.*

$$\Sigma_1^{\mathrm{CCI}} \in \mathsf{SignInv}_{\leq^{\mathrm{CCI}}}$$

On peut donc construire le PTS avec tous les opérateurs de CCI :

Définition 6.63 (`cci_pts`) *On peut former un PTS avec le sous-typage et la signature définis ci-dessus.*

$$\mathrm{CCI}(\Sigma) = \left\langle \textsc{Axiom}; \textsc{Rule}; \textsc{Pair}; \Sigma_0^{\mathrm{CCI}}; \Sigma_1^{\mathrm{CCI}}; \leq^{\mathrm{CCI}} \right\rangle \in \mathcal{PTSO}$$

À ce point, nous avons montré que la définition de CCI de ce chapitre rentrait bien dans le cadre des PTS avec opérateurs défini dans le chapitre 5. Nous allons maintenant montrer les résultats métathéoriques spécifiques de CCI, qui permettront de prouver que ce PTS admet des jugments de typage décidables.

$$(\textsc{Proj1})\ \frac{\Gamma \vdash M : A * B}{\Gamma \vdash \pi_1(M) : A} \qquad (\textsc{Proj2})\ \frac{\Gamma \vdash M : A * B}{\Gamma \vdash \pi_2(M) : B}$$

$$(\textsc{App})\ \frac{\Gamma \vdash M : \Pi x{:}A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B\{0\backslash N\}}$$

$$(\textsc{Cst})\ \frac{C \rhd_\Sigma \mathsf{Cst}\,\langle T;\delta\rangle \qquad \Gamma \vdash M : T}{\Gamma \vdash \mathrm{Cst}\{C\}(M) : \delta.\mathrm{T}\{0\backslash M\}}$$

$$(\textsc{Mark})\ \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{M} : \mathrm{Prop}} \qquad (\textsc{Lmark})\ \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{M}^* : \mathrm{Prop}} \qquad (\textsc{Mt})\ \frac{\Gamma \vdash}{\Gamma \vdash \epsilon : \mathcal{M}}$$

$$(\textsc{Nil})\ \frac{\Gamma \vdash}{\Gamma \vdash \varnothing : \mathcal{M}^*} \qquad (\textsc{Cons})\ \frac{\Gamma \vdash M : \mathcal{M} \qquad \Gamma \vdash N : \mathcal{M}^*}{\Gamma \vdash M :: N : \mathcal{M}^*}$$

$$(\textsc{Ind})\ \frac{I \rhd_\Sigma \mathsf{Ind}(\rho,\Phi) \qquad \Gamma \vdash P : \Phi.\mathrm{P} \qquad \Gamma \vdash A : \Phi.\mathrm{A}\{0\backslash P\} \qquad \Gamma \vdash L : \mathcal{M}^*}{\Gamma \vdash \mathrm{Ind}\{I,n\}(P,A,L) : \Phi.\mathrm{S}}$$

$$(\textsc{Constr})\ \frac{x \rhd_\Sigma \mathsf{Constr}(\Phi,\kappa) \qquad \Gamma \vdash P : \Phi.\mathrm{P} \qquad \Gamma \vdash A : \kappa.\mathrm{A}\{0\backslash P\}}{\Gamma \vdash \mathrm{Constr}\{x\}\,(P,A) : \mathrm{Ind}(\Phi.\mathrm{X}, P, \kappa.\mathrm{I}\{1\backslash P\}\{0\backslash A\})}$$

$$(\textsc{Case})\ \frac{\begin{array}{c}\Gamma \vdash M : \mathrm{Ind}\{I,n\}(P,A,L) \qquad I \rhd_\Sigma \mathsf{Ind}(\rho,\Phi) \\ \Gamma \vdash Q : \mathsf{Cpf}(\Phi,s_e)\{0\backslash P\} \qquad \mathrm{ElimSort}(\Phi.\mathrm{CK},\Phi.\mathrm{S},s_e) \qquad \mathrm{Full}(\Phi,\vec{m}) \\ \Gamma \vdash B : (*\vec{V})\{0\backslash P\} \qquad \mathrm{CaseBranch}(\rho,\Phi,Q,L,\vec{m},\vec{V})\end{array}}{\Gamma \vdash \langle Q\rangle \mathrm{Cases}\ M\ \mathrm{of}\ \vec{m} \Rightarrow B\ \mathrm{end} : (Q\ A\ M)}$$

$$(\textsc{Fix})\ \frac{\Gamma \vdash \mathcal{FTYS}(\vec{F}) : s \qquad \Gamma \vdash B : \mathcal{FBD}(\vec{F})}{\Gamma \vdash \mathrm{Fix}(\mathcal{FTYS}(\vec{F}), B) : \mathcal{FTYS}(\vec{F})}$$

$$(\textsc{Record})\ \frac{\Gamma \vdash \mathrm{RecordArity}(\Theta) : s \qquad \mathrm{AllDiff}(\Theta.1)}{\Gamma \vdash \mathrm{Record}\{\Theta\} : s}$$

$$(\textsc{Struct})\ \frac{\Gamma \vdash \mathrm{ProdArity}(\theta) : s \qquad \Gamma \vdash \theta.3 : \mathrm{FldTyp}_0(\theta)}{\Gamma \vdash \mathrm{Struct}\{\theta\} : \mathrm{Record}\{\theta\}}$$

$$(\textsc{Field})\ \frac{\Gamma \vdash M : \mathrm{Record}\{\Theta\} \qquad \Theta.1 = \vec{x_i} \qquad \mathrm{FldTyp}_0(\Theta + M) = *\vec{T_i}}{\Gamma \vdash \mathrm{Field}\{x_k\}\,(M) : T_k}$$

FIG. 6.4: Règles de typage des opérateurs de CCI

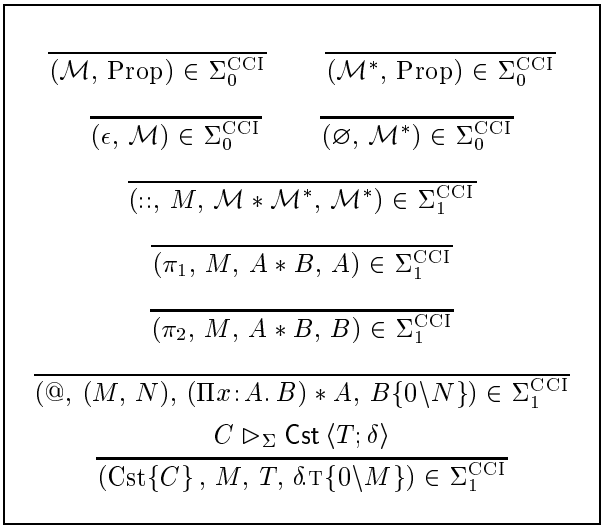


FIG. 6.5: Définition partielle de la signature de CCI

Chapitre 7

Métathéorie de CCI

Puisque nous avons montré que CCI(Σ) était un PTS avec opérateur, nous avons automatiquement les résultats métathéoriques simples (section 4.3.2) comme le lemme d'affaiblissement, de substitution, la correction des types, etc.

Pour atteindre notre objectif (un noyau pour CCI), il suffit donc d'instancier la structure regroupant les algorithmes élémentaires (voir définition 5.31). La partie la plus complexe est la décidabilité de l'appartenance à la signature. La complexité venant du fait qu'il faut garantir la bonne formation des types provenant de cette signature. Pour cela, nous aurons notamment besoin des résultats d'auto-réduction que nous montrerons dans ce chapitre.

7.1 Signature bien formée

On fait quelques hypothèses sur la signature. La raison pour cela est que la règle de typage des opérateurs exige que les types issus de la signature soient bien typés. Or, l'algorithme de typage sera simple (surtout la preuve de terminaison) et efficace si l'on ne fait des appels récursifs que pour typer les sous-termes. Nous introduisons un prédicat de bonne formation de la signature. Tout d'abord nous introduisons une version simpliste ($\models$ ci-dessous) dont la décidabilité n'est pas garantie, puis nous verrons (section 7.6) une condition plus restrictive que nous saurons décider.

Définition 7.1 (prédicat `sign_ok`) *Une signature est dite bien formée (noté $\models \Sigma$) si elle vérifie les conditions suivantes :*

$$
\begin{array}{ll}
\langle\ \texttt{ok\_gs\_inj:} & \forall x, G, G'.\, x \rhd_\Sigma G \ \wedge\ x \rhd_\Sigma G' \ \Rightarrow\ G = G'; \\
\texttt{ok\_cst:} & \forall x, T, \delta.\, x \rhd_\Sigma \mathsf{Cst}\,\langle T; \delta\rangle \ \Rightarrow\ [\_ : T]\,\delta \vdash; \\
\texttt{ok\_ind:} & \forall I, \rho, \Phi.\, I \rhd_\Sigma \mathsf{Ind}(\rho, \Phi) \ \Rightarrow\ [\_ : \Phi.\mathrm{P}]\,[\_ : \Phi.\mathrm{A}]\,[\_ : \Phi.\mathrm{S}] \vdash; \\
\texttt{ok\_constr:} & \forall c, \Phi, \kappa.\, c \rhd_\Sigma \mathsf{Constr}(\Phi, \kappa) \ \Rightarrow\ [\_ : \Phi.\mathrm{P}]\,[\_ : \kappa.\mathrm{A}] \vdash \kappa.\mathrm{I} : \uparrow^1 \Phi.\mathrm{A}; \\
\texttt{ok\_normalize:} & \forall \Gamma.\, \mathcal{WT}_\Gamma \subseteq \mathcal{SN}_\Gamma^{\rightarrow \mathrm{CCI}} \\
\rangle &
\end{array}
$$

La première de ces conditions exige que la signature soit injective, c'est-à-dire qu'elle définit de manière unique chaque nom. On n'interdit pas a priori de définir plusieurs fois le même nom si les définitions sont les mêmes, mais cela n'a pas grande importance.

Les trois conditions suivantes assurent que les types des objets définis par la signature ont des types bien formés. Enfin, nous supposons que tous les termes bien typés sont fortement normalisables. Nous verrons que cette condition n'est pas vérifiée pour toutes les déclarations de types inductifs, et nous devrons introduire la notion de stricte positivité.

7.2 Règles dérivées et lemmes d'inversion

Le lemme d'inversion est un peu insuffisant dans le cas des opérateurs car il dit simplement qu'il existe un quintuplet dans la définition de la signature qui met en relation l'opérateur avec la valeur et le type des arguments de cet opérateur. Nous souhaiterions avoir des résultats plus précis. Par exemple, si $\mathrm{Cst}\{C\}(M)$ est de type U, alors il existe une déclaration de constante ξ dans la signature concrète, telle que, entre autre, $C \rhd_\Sigma \mathsf{Cst}(\xi)$. Ces règles servent à gommer l'indirection (ou l'encodage) introduite par la notion d'opérateur.

Nous considérons une signature Σ, et nous supposons qu'elle vérifie $\models \Sigma$. Nous garderons cette hypothèses jusqu'à la section 7.6.

7.2.1 Règles dérivées

Lemme 7.2 (`projl_vect`) *La k-ième projection d'un terme M de type $B_1 * \ldots * B_n$ a le type B_k*

$$\frac{\Gamma \vdash M : \overset{i<n}{\underset{0\leq i}{*}} B_i}{\Gamma \vdash \pi_1(\pi_2^k(M)) : B_k}$$

Reamrque : le preuve de ce lemme n'utilise pas le fait que la signature est bien formée ($\models \Sigma$), puisque le typage des paires est indépendant de la signature.

Lemme 7.3 (`typ_induc0`) *La règle de typage dérivée suivante permet de montrer la bonne formation d'un type inductif en exhibant une déclaration dans Σ qui le définisse, et en montrant que ses arguments ont le type attendu :*

$$\frac{I \rhd_\Sigma \mathsf{Ind}(\rho, \Phi) \qquad \Gamma \vdash P : \Phi.\mathrm{P} \qquad \Gamma \vdash A : \Phi.\mathrm{A}\{0\backslash P\} \qquad \Gamma \vdash L : \mathcal{M}^*}{\Gamma \vdash \mathrm{Ind}\{I, n\}(P, A, L) : \Phi.\mathrm{S}}$$

Nous montrons ensuite un résultat similaire pour les constructeurs de types inductifs :

Lemme 7.4 (`typ_constructor`)

$$\frac{C \rhd_\Sigma \mathsf{Constr}(\Phi, \kappa) \qquad \Gamma \vdash P : \Phi.\mathrm{P} \qquad \Gamma \vdash A : \kappa.\mathrm{A}\{0\backslash P\}}{\Gamma \vdash \mathrm{Constr}\{C\}\,(P, A) : \mathrm{Ind}(\Phi.\mathrm{X}, P, \kappa.\mathrm{I}\{1\backslash P\}\{0\backslash A\})}$$

7.2.2 Règles d'inversion

Lemme 7.5 (`inv_projn`)

$$\Gamma \vdash \pi_2^k(M) : U \;\Rightarrow\; \exists V.\; \begin{cases} \Gamma \vdash M : V \\ \forall \vec{A}.\, \Gamma \vdash \overset{i<n}{\underset{0\leq i}{*}} A_i \leq^{\mathrm{CCI}} V \;\Rightarrow\; \Gamma \vdash \overset{i<n}{\underset{k\leq i}{*}} A_i \leq^{\mathrm{CCI}} U \end{cases}$$

La preuve de ce lemme n'utilise pas l'hypothèse $\models \Sigma$.

Les trois lemmes suivants sont utilisés pour montrer l'auto-réduction. Ils permettent d'inverser les jugements de typage pour lesquels le type est un opérateur de type canonique

(resp. produit, produit cartésien ou type inductif) et le terme sous forme du constructeur correspondant (resp. abstraction, paire ou constructeur). Les deux premiers lemmes n'utilisent pas l'hypothèse $\models \Sigma$.

Lemme 7.6 (`inv_pair_sum`)

$$\Gamma \vdash (M,\, N) :\, A * B \;\Rightarrow\; \begin{cases} \Gamma \vdash M :\, A \\ \Gamma \vdash N :\, B \end{cases}$$

Nous disposons d'un lemma similaire à `inv_pair_sum` pour le type produit :

Lemme 7.7 (`inv_typ_lam_prod`)

$$\Gamma \vdash \lambda x{:}A.\, M :\, \Pi y{:}B.\, U \;\Rightarrow\; \exists (s_1, s_2, s_3) \in \textsc{Rule}.\; \begin{cases} \Gamma \vdash B \leq^{\text{CCI}} A \\ \Gamma \vdash B :\, s_1 \\ \Gamma\,[y : B] \vdash M :\, U \\ \Gamma\,[y : B] \vdash U :\, s_2 \end{cases}$$

Et de même pour les types inductifs :

Lemme 7.8 (`inv_typ_constr_ind`)

$$\Gamma \vdash \text{Constr}\{C\}\,(P, A) : \text{Ind}\{I, n\}(P', A', L) \;\Rightarrow$$
$$\exists \rho, \Phi, \kappa, L'.\; \begin{cases} I \rhd_\Sigma \mathsf{Ind}(\rho, \Phi) \\ C \rhd_\Sigma \mathsf{Constr}(\Phi, \kappa) \\ \Gamma \vdash P :\, \Phi_{\text{P}} \\ \Gamma \vdash A :\, \kappa_{.\text{A}}\{0 \backslash P'\} \\ \Gamma \vdash P \equiv_{\text{CCI}} P' \\ \Gamma \vdash \kappa_{.\text{I}}\{1 \backslash P'\}\{0 \backslash A\} \equiv_{\text{CCI}} A' \\ \Gamma \vdash L \equiv_{\text{CCI}} L' \\ \Gamma \vdash \text{Ind}(I, P', A')\; [\prec_M]^T\; \text{Ind}\{I, n\}(P', A', L') \end{cases}$$

7.3 Correction de la signature

Ce sont des résultats qui vont nous dispenser de vérifier effectivement que le type retourné par la signature est lui-même bien typé, ce qui permettra de montrer la décidabilité de l'appartenance à la signature d'opérateurs.

Lemme 7.9 (`typ_fix_correctness`)

$$\frac{\Gamma \left[\_ : \mathcal{FTYS}(\vec{F})\right] \vdash}{\Gamma \left[\_ : \mathcal{FBD}(\vec{F})\right] \vdash}$$

Nous avons des résultats similaires pour l'opérateur de filtrage :

Lemme 7.10 (`wf_case_predicate`)

$$\frac{I \rhd_\Sigma \mathsf{Ind}(\rho, \Phi) \qquad \text{Ind}\{I, n\}(P, A, L) \in \mathcal{WT}_\Gamma \qquad (s_e, s) \in \textsc{Axiom}}{\Gamma\,[\_ : \mathsf{Cpf}(\Phi, s_e)\{0 \backslash P\}] \vdash}$$

Remarque : la condition $\mathrm{Ind}\{I,n\}(P,A,L) \in \mathcal{WT}_\Gamma$ pourrait être remplacé par $\Gamma \vdash P : \Phi_P$.

Lemme 7.11 (`str_substp_ok`) *Le marquage preserve le typage.*

$$\frac{\Gamma \vdash M : s \qquad M \overset{\rho,L}{\gg} M' \qquad \Gamma \vdash L : \mathcal{M}^*}{\Gamma \vdash M' : s}$$

Lemme 7.12 (`wf_branch`) *Typage d'une branche de case :*

$$\frac{\begin{array}{c} m \rhd_\Sigma \mathsf{Ind}(\Phi,\kappa) \qquad \kappa_A \overset{\rho,\uparrow^1 L}{\gg} A \\ \Gamma \vdash L : \mathcal{M}^* \qquad \Gamma \vdash P : \Phi_P \qquad \Gamma \vdash Q : \mathsf{Cpf}(\Phi, s_e)\{0\backslash P\} \end{array}}{\Gamma\,[\_ : \mathcal{BTY}(\kappa, A, Q)\{0\backslash P\}] \vdash}$$

Une conséquence immédiate est que le vecteur des types de branches est bien formé :

Lemme 7.13 (`wf_branch_vect`)

$$\frac{\begin{array}{c} I \rhd_\Sigma \mathsf{Ind}(\rho,\Phi) \qquad \Gamma\,[\_ : \mathrm{Ind}\{I,n\}(P,A,L)] \vdash \\ \Gamma \vdash Q : \mathsf{Cpf}(\Phi, s_e)\{0\backslash P\} \qquad \mathrm{CaseBranch}(\rho,\Phi,Q,L,\vec{m},\vec{B}) \end{array}}{\Gamma\left[\_ : (*\vec{B})\right] \vdash}$$

Nous montrons des résultats concernant les enregistrements :

Lemme 7.14 (`record_arity_ok`,`prod_arity_ok`)

$$\forall \Theta.\ (\Gamma\,[\_ : \mathrm{ProdArity}(\Theta)] \vdash) \ \Leftrightarrow\ (\Gamma\,[\_ : \mathrm{RecordArity}(\Theta)] \vdash)$$

Lemme 7.15 (`wf_fields`)

$$\frac{\Gamma \vdash M : \mathrm{Record}\{\Theta.1\}\,(\mathrm{RecordArity}(\Theta))}{\Gamma\,[\_ : \mathrm{FldTyp}_0(\Theta + M)] \vdash}$$

Preuve La complexité de ce lemme est que pour pouvoir typer le type de la n-ième projection, il faut avoir vérifier non seulement que les types des $n-1$-ièmes projections sont bien formés, mais en plus que les $n-1$-ièmes projections de M ont le bon type. ■

7.4 Résultats d'auto-réduction

On montre séparément l'auto-réduction pour chacune des composantes de la réduction (y compris α). Le schéma de preuve est toujours le même : on applique les lemmes d'inversion sur le jugement concernant le radical, puis on applique les règles de typage pour former le jugement du réduit. Il se peut que l'on ait besoin de quelques conditions annexes sur le sous-typage ou la signature.

Lemme 7.16 (`sr_beta`) *La β-réduction est correcte. La condition-clé est l'inversion du sous-typage du produit.*

Preuve La preuve est similaire à celle que nous avions fait pour les autres développemnts. ■

Lemme 7.17 (`sr_delta`) *La δ-réduction (de constantes locales ou globales) est correcte.*

Preuve La encore, en ce qui concerne la δ-réduction de variable de de Bruijn, nous reprenons la preuve faite pour les PTS. L'autre δ-réduction n'est guère plus complexe. Il suffit d'utiliser le lemme de substitution pour s'assurer que la substitution de l'argument dans le corps de la constante est bien typée. ■

Lemme 7.18 (`sr_alpha`) *L'α-réduction est correcte.*

Preuve Ce résultat est particulièrement facile étant donné que le typage est indépendant des noms de variables présents dans l'environnement de de Bruijn. ■

Lemme 7.19 (`sr_proj1, sr_proj2`) *Les règles de réduction des projections des paires (π) est correcte.*

Lemme 7.20 (`sr_fix_free, sr_iota_fix`) *La réduction de point fixe vérifie la propriéte d'auto-réduction.*

Preuve Nous commençons par prouver la correction de l'expansion de point fixe non gardée. ■

Le résultat suivant est le plus difficile :

Lemme 7.21 (`sr_iota_case`) *La réduction du filtrage vérifie l'auto-réduction.*

Lemme 7.22 (`sr_projr`) *La réduction des projections d'enregistrements vérifie la propriété d'auto-réduction.*

Nous avons donc prouvé l'auto-réduction de la réduction au sommet du terme ($\rightarrow_{\mathrm{CCI}} \in \mathcal{SR}$). Le lemme `ctxt_sound` permet d'en déduire le résultat usuel d'auto-réduction, lorsque la réduction peut se faire dans n'importe quel sous-terme. Ce lemme possède quelques prémisses qu'il nous reste à montrer :

Lemme 7.23 (`mem_sign_stable_val`) *La signature est stable par réduction de la valeur de l'argument :*

$$(c,\Gamma,M,T,U) \in \Sigma_1^{\mathrm{CCI}} \ \wedge \ \Gamma \vdash M \rightarrow_{\rightarrow_{\mathrm{CCI}}/\!/} M' \ \Rightarrow \ \exists T', U'. \begin{cases} \Gamma \vdash T \rightarrow_{\rightarrow_{\mathrm{CCI}}/\!/} T' \\ \Gamma \vdash U \rightarrow_{\rightarrow_{\mathrm{CCI}}/\!/} U' \\ (c,\Gamma,M',T',U') \in \Sigma_1^{\mathrm{CCI}} \end{cases}$$

Les sous-preuves de ce lemme concernant les enregistrements sont admises.

Lemme 7.24 (`subject_reduction`) *La relation de réduction de CCI vérifie la propriété d'auto-réduction*

$$[\underset{\mathrm{CCI}}{\rightarrow}] \in \mathcal{SR}$$

7.5 Décidabilité du typage

Ensuite, on montre que la signature est décidable, ce qui nous permettra de construire les fonctions de typage.

Algorithme 7.25 (`inf_sign0_dec`) *La signature des opérateurs constants Σ_0^{CCI} est décidable et injective.*

$$\forall c.\, \mathsf{InfPpal}(U \mid (c,U) \in \Sigma_0^{\mathrm{CCI}} \ \wedge \ [\_ : U] \vdash, =)$$

On suppose que le système est fortement normalisant, que le sous-typage est décidable et qu'il existe un algorithme de forme normale de tête.

7.5.1 Filtrage sur les types

On peut alors décider si un type peut se coercer vers l'un des quatres constructeurs de types que sont les sortes, les paires, les produits et les types inductifs.

On profite du fait que décider si un type se coerce vers un des quatre constructeurs de type est équivalent à tester s'il se réduit vers l'un de ces quatres constructeurs. Ce ne serait pas le cas si on avait du sous-typage par coercions, car une variable de type pourrait se coercer vers un constructeur de type sans qu'il ne se réduise.

Algorithme 7.26 (`red_to_sort_dec`) *Il est décidable de savoir si un type bien formé est plus petit qu'une sorte. Et on peut inférer la plus sorte sorte si elle existe.*

$$T \in \mathcal{WT}_\Gamma \;\Rightarrow\; (\exists^\star s \in \textsc{Sort}.\, \Gamma \vdash T \triangleright^*_{\text{CCI}} s) \;+\; \forall s \in \textsc{Sort}.\, \neg(\Gamma \vdash T \leq^{\text{CCI}} s)$$

Cet algorithme est utilisé pour transformer un jugement en hypothèse.

Algorithme 7.27 (`red_to_sum_dec`) *Il est décidable de savoir si un type bien formé est plus petit qu'un type somme.*

$$T \in \mathcal{WT}_\Gamma \;\Rightarrow\; (\exists^\star A, B.\, \Gamma \vdash T \triangleright^*_{\text{CCI}} A * B) \;+\; \forall A, B.\, \neg(\Gamma \vdash T \leq^{\text{CCI}} A * B)$$

Algorithme 7.28 (`red_to_prd_dec`) *Il est décidable de savoir si un type bien formé est plus petit qu'un type produit.*

$$T \in \mathcal{WT}_\Gamma \;\Rightarrow\; (\exists^\star x, A, B.\, \Gamma \vdash T \triangleright^*_{\text{CCI}} \Pi x{:}A.\, B) \;+\; \forall x, A, B.\, \neg(\Gamma \vdash T \leq^{\text{CCI}} \Pi x{:}A.\, B)$$

Algorithme 7.29 (`red_to_ind_dec`) *Il est décidable de savoir si un type bien formé est plus petit qu'un type inductif.*

$$\begin{aligned} T \in \mathcal{WT}_\Gamma \;\Rightarrow\; & (\exists^\star I, n, P, A, L.\, \Gamma \vdash T \triangleright^*_{\text{CCI}} \text{Ind}\{I, n\}(P, A, L)) \\ & + \forall I, n, P, A, L.\, \neg(\Gamma \vdash T \leq^{\text{CCI}} \text{Ind}\{I, n\}(P, A, L)) \end{aligned}$$

Ces quatre spécifications se prouvent en filtrant le type en forme normal de tête. Dans le cas du type inductif, on utilise le fait que T est bien typé pour en déduire qui si le terme est en forme normale de tête, alors l'argument de l'opérateur inductif est sous la forme d'un triplet.

7.5.2 Typage des constructions élémentaires

Algorithme 7.30 (`sign_dec_const`) *L'inférence du type principal d'une constante est décidable.*

Preuve On appelle `search_global`, et on compare le type de l'argument avec le type du paramètre de la constante. ■

Algorithme 7.31 (`sign_dec_pi1`,`sign_dec_pi2`) *L'inférence du type principal des projections est décidable.*

Preuve Il suffit de vérifier que le type de l'argument de l'opérateur se réduit vers une somme à l'aide du programme `red_to_sum_dec` ci-dessus. Le type de la première (resp. deuxième) projection est alors le sous-terme gauche (resp. droit) de cette somme. ■

Algorithme 7.32 (`sign_dec_app`) *L'inférence du type principal de l'application est décidable.*

Preuve Il faut vérifier que le type de l'argument est la somme d'un produit et d'un type qui est un sous-type du sous-terme gauche de ce produit. ■

Algorithme 7.33 (`sign_dec_cons`) *L'inférence du type principal de la concaténation de marques est décidable.*

Preuve Il suffit de vérifier que le type de l'argument est un sous-type de $\mathcal{M} * \mathcal{M}^*$. ■

Voilà pour les constructions élémentaires du calcul. Maintenant, on va prouver la même chose pour les opérateurs des types inductifs, ce qui nécessitera des lemmes intermédiaires.

7.5.3 Types inductifs

Algorithme 7.34 (`sign_dec_fixp`) *L'inférence du type principal des points fixes est décidable.*

Ce résultat est facile puisque le point fixe est annoté avec tous les types. On pourrait peut-être se passer de cette annotation.

Algorithme 7.35 (`sign_dec_indt`) *L'inférence du type principal des types inductifs est décidable.*

Pas de difficulté particulière.

Algorithme 7.36 (`sign_dec_cstr`) *L'inférence du type principal des constructeurs est décidable.*

Pas de difficulté particulière.

Algorithme 7.37 (`sign_dec_case`) *L'inférence du type principal de l'opérateur de filtrage est décidable.*

Ce dernier résultat est de loin le plus complexe.

7.5.4 Enregistrements

Algorithme 7.38 (`sign_dec_record`) *L'inférence du type principal des types enregistrement est décidable.*

Preuve Pas de difficulté particulière. Il suffit de vérifier que les noms des champs sont distincts, que l'argument est sous la forme RecordArity(Θ) pour un certain Θ, et de trouver la plus petite sorte vers laquelle se coerce le type de cette arité (textttred_to_sort_dec). ■

Algorithme 7.39 (`sign_dec_struct`) *L'inférence du type principal des enregistrement est décidable.*

Preuve Cet algorithme requiert beaucoup de soin. Les vérifications préliminaires portent sur la forme de l'argument de l'opérateur (Struct$\{\vec{x}\}$) doit être appliqué à une paire (T, M) telle que T soit une succession de produit, ce qui nous permet de reconstruire l'arité Θ permettant de typer notre eregistrement. Cela ne pose pas de problème. Il nous reste à assurer que le type enregistrement Record$\{\vec{x}\}$ (RecordArity(Θ)) est bien typé, ce qui, grâce au résultat `record_arity_ok` se fait simplement en vérifiant que les noms de $\vec{x}$ sont tous distinct.

Enfin, il faut vérifier que les types des champs peuvent se coercer vers les types des champs de notre enregistrement, à savoir $\mathrm{FldTyp}_0(\Theta + M)$. Il serait erroné de simplement appeler l'algorithme de décidabilité du sous-typage avec le type de M avec $\mathrm{FldTyp}_0(\Theta + M)$. En effet, nous devons vérifier cette inclusion de types dans un ordre bien précis pour éviter de réduire un terme dont nous ne pouvons prouver qu'il est bien typé. Or, nous ne saurons que $\mathrm{FldTyp}_0(\Theta + M)$ est bien formé que lorsque nous saurons que les projections de M sont typées correctement, ce que nous sommes justement en train de vérifier. Pour que tout se

passe bien, il faut commencer par vérifer que le type de M est un produit dont le sous-terme gauche est un sous-type du premier type de Θ. Ce type est bien formé puisque ProdArity(Θ) est bien typé. Si ce test réussit, il permet de typer la première projection ce qui assure que le type de la deuxième composante de Θ instancié par la première projection de M est bien formé, et ainsi de suite. ■

Algorithme 7.40 (`sign_dec_field`) *L'inférence du type principal des accès aux champs d'un enregistrement est décidable.*

Preuve Nous commençons par vérifier que le type de l'argument se réduit vers un type enregistrement, dont on calcule l'arité. Il suffit alors de parcourir cette arité jusqu'à trouver un champ ayant le bon nom, en accumulant les substitutions par les valeurs des champs précédents. Le lemme `wf_fields` nous assure que le type retourné est correct. ■

7.5.5 Foncteur de construction du noyau

En regroupant tous les résultats de la section précédente, nous obtenons une preuve de la décidabilité de l'appartenance à la signature, qui est la clause `infer_sign1` de la structure PtsOpAlgos.

Lemme 7.41 (`cci_algos`) *On peut instancier la structure qui assure la décidabilité du typage.*

$$\mathrm{CCI}(\Sigma) \in \textsc{PtsOpAlgos}$$

En utilisant le résultat sur les PTS avec opérateurs (théorème 8), on en déduit la décidabilité du typage, ainsi que les fonctions de typage du vérificateur de preuves. Il faut cependant se rappeler que nous avions fait quelques suppositions sur Σ.

Théorème 9 (`cci_type_checker`) *Sous l'hypothèse* $\models \Sigma$*, le système de type* CCI(Σ) *est décidable, c'est-à-dire qu'il existe une implantation de l'interface* PtsOpTc *:*

$$\mathrm{CCI}(\Sigma) \in \textsc{PtsTc}$$

Il reste maintenant à construire un algorithme qui permet d'assurer que la précondition de ce théorème est vérifiée. Notamment, il faut trouver une condition pour que le système soit fortement normalisant. Dans la prochaine section, nous définissons un jugement décidable qui impliquera la normalisation forte, ainsi que les quatre autres modalités de $\models \Sigma$.

7.6 Typage de la signature

Depuis le début du chapitre précédent, nous raisonnions à signature fixée, sur laquelle nous faisions certaines hypothèses, regroupée dans la définition de $\models \Sigma$. Dans cette section, nous définissons et étudions les vérifications à faire pour maintenir cet invariant, lorsque l'on étend la signature.

C'est ici que nous définirons ce que c'est qu'une signature bien formée, i.e. composée de déclaration correctes. Pour les types inductifs, il y a un bon nombre de vérifications à faire : positivité des constructeurs, restriction des sortes vers lesquelles on peut faire des éliminations en fonction de la prédicativité ou non de la déclaration.

Nous allons mettre en évidence un système de règles d'inférence décidable dont on pourra prouver qu'il a pour conséquence $\models \Sigma$, et donc que le typage dans cette signature est décidable.

Nous aurions pu définir ce jugement dès le début, à la place de $\models \Sigma$. Mais cela aurait été plus pénible formellement car ce jugement ne travaille pas sur un unique PTSO, ce qui nous aurait empêché de raisonner à signature fixée sans avoir à l'expliciter. Une meilleure raison est que $\models \Sigma$ énonce des conditions relativement indépendantes de l'implantation de la signature concrète. Si l'on avait choisi, au lieu d'une simple liste, un graphe acyclique dirigé (ce qui permet d'établir des dépendances plus fines entre les constantes), cela aurait de lourdes conséquences sur les preuves des résultats métathéoriques. Ici, il suffit de montrer que le typage de la signature ($\Sigma \vdash$) implique $\models \Sigma$.

7.6.1 Typage des constantes

Pour les constantes, c'est très simple : nous vérifions simplement que la déclaration est correcte, et que le nom n'est pas déjà utilisé.

Définition 7.42 (prédicat `wf_cst`) *Une déclaration de constante ξ est bien formée dans la signature Σ si le contexte formé par le paramètre ξ_{P} et la déclaration ξ_{D} est valide, et si le nom n'est pas déjà défini dans Σ :*

$$\Sigma \vdash \mathrm{WfCst}(\xi) \stackrel{\text{def}}{\equiv} \begin{cases} \Sigma \mid [\_ : \xi_{\mathrm{P}}]\, \xi_{\mathrm{D}} \vdash \\ \xi_{\mathrm{D.X}} \notin \mathrm{Dom}(\Sigma) \end{cases}$$

7.6.2 Typage des inductifs

Arités de types inductifs

On a déjà vu que la vérification d'une déclaration inductive se faisait en deux temps. Dans un premier temps, on s'assure que les arités associées aux types inductifs sont bien formées. Cela permet d'ajouter dans la signature une déclaration de type inductif abstrait. Cela est nécessaire pour pouvoir typer les types des constructeurs, qui font référence au type inductif que nous sommes en train de définir.

Définition 7.43 (prédicat `wf_arity`) *Un type inductif abstrait est bien formé s'il vérifie les conditions suivantes :*

$$\Sigma \vdash \mathrm{WfArity}(\Psi) \stackrel{\text{def}}{\equiv} \begin{cases} \forall \Phi \in \Psi. \begin{cases} \Sigma \mid [\_ : \Phi_{\mathrm{P}}]\, [\_ : \Phi_{\mathrm{A}}]\, [\_ : \Phi_{\mathrm{S}}] \vdash \\ \Phi_{\mathrm{X}} \notin \mathrm{Dom}(\Sigma) \end{cases} \\ \mathrm{AllDiff}(FV(\overline{\Psi})) \end{cases}$$

Positivité stricte

Pour que cela soit cohérent, il faut imposer des restrictions sur la forme des types de constructeurs : il doivent être positifs pour que le point fixe converge.

Il est clair qu'autoriser une définition inductive I avec un constructeur C dont le type serait

$$C : (I \to \bot) \to I$$

serait incohérent, puisqu'il introduirait l'équivalence des propositions I et $\neg I$.

Mais nous avons encore une incohérence avec le constructeur suivant (qui est positif au sens large, car I apparaît à gauche de deux flèches dans l'argument du constructeur) :

$$C : ((I \to \mathrm{Prop}) \to \mathrm{Prop}) \to I$$

En effet, il permettrait de construire une bijection entre I et les parties des parties de I, ce qui permet d'encoder le paradoxe de Russel (ceci a été formalisé par Paulin dans [49]).

Nous nous restreindront aux constructeurs strictement positifs, c'est-à-dire n'apparaissant dans aucun sous-terme gauche de produit.

Dans la définition suivante $FV(T)$ représente l'ensemble des noms de types inductifs qui apparaissent dans T. Comme ces noms ne sont pas liés, la définition est évidente.

Définition 7.44 *L'ensemble des types de constructeurs strictement positifs* $\mathrm{Pos}^k(\rho)$*, est :*

$$\frac{\rho \notin FV(T)}{T \in \mathrm{Pos}^k(\rho)} \qquad \frac{I \in \rho \qquad \rho \notin FV(A)}{\mathrm{Ind}(I, \natural k, A) \in \mathrm{Pos}^k(\rho)}$$

$$\frac{T \in \mathrm{Pos}^k(\rho) \qquad U \in \mathrm{Pos}^k(\rho)}{T * U \in \mathrm{Pos}^k(\rho)} \qquad \frac{\rho \notin FV(T) \qquad U \in \mathrm{Pos}^{k+1}(\rho)}{\Pi x{:}T.\, U \in \mathrm{Pos}^k(\rho)}$$

$$\frac{\Theta \in \mathrm{Pos}^k(\rho)}{\mathrm{Record}\{\Theta\} \in \mathrm{Pos}^k(\rho)} \qquad \frac{}{[] \in \mathrm{Pos}^k(\rho)} \qquad \frac{T \in \mathrm{Pos}^k(\rho) \qquad \Theta \in \mathrm{Pos}^{k+1}(\rho)}{(x, T)\Theta \in \mathrm{Pos}^k(\rho)}$$

La liste de noms ρ comporte les noms des types inductifs mutuellement définis, et k représente l'indice de de Bruijn qui représente les paramètres du type inductifs.

Ici encore, nous définissons ce prédicat de manière mutuellement récursive avec son équivalent sur les arités d'enregistrement.

Le paramètre k sert à s'assurer que les arguments récursifs se font toujours avec les mêmes valeurs de paramètre.

On peut remarquer que cette définition de la positivité n'autorise à faire apparaître les types inductifs de ρ uniquement à des sous-termes sur lesquels se propagent les marques (voir la définition de `str_substp`, définition 6.38). Cela signifie que sous la condition de positivité, la fonction qui effectue le marquage des branches de Case n'aurait pas à se soucier des positions auxquelles les types inductifs de ρ apparaissent.

Types de constructeurs

Définition 7.45 (prédicat `constr_pos`) *L'ensemble des types de constructeur positifs est défini de la manière suivante :*

$$\frac{\kappa.\mathrm{A} \in \mathrm{Pos}^0(\rho) \qquad \rho \notin FV(\kappa.\mathrm{I})}{\kappa \in \textsc{Constructor}(\rho)}$$

Cette définition n'est pas encore définie formellement.

Lemme 7.46 (`constr_pos_dec`) *L'ensemble des types de constructeurs positifs est décidable.*

$$\forall \rho, T.\, \mathsf{Dec}\,(T \in \textsc{Constructor}\rho)$$

Définition 7.47 (prédicat `wf_constructor`,`wf_constr`) *Constructeurs adaptés à un type inductif abstrait :*

$$\Sigma \vdash \mathrm{WfConstr}(\Psi) \stackrel{\mathrm{def}}{\equiv} \begin{cases} \forall \Phi \in \Psi.\, \forall \kappa \in \Phi.\mathrm{C}. \begin{cases} \Sigma \mid [\_ : \Phi.\mathrm{P}] \vdash \kappa.\mathrm{A} : \Phi.\mathrm{CK} \\ \Sigma \mid [\_ : \Phi.\mathrm{P}]\,[\_ : \kappa.\mathrm{A}] \vdash \kappa.\mathrm{I} : \uparrow^1 \Phi.\mathrm{A} \\ \kappa \in \textsc{Constructor}(FV(\overline{\Psi})) \\ \kappa.\mathrm{X} \notin \mathrm{Dom}(\Sigma) \end{cases} \\ \mathrm{AllDiff}(FV(\Psi)) \end{cases}$$

$$\frac{}{[] \vdash} \qquad \frac{\Sigma \vdash \qquad \Sigma \vdash \mathrm{WfCst}(\xi)}{\Sigma; \xi \vdash}$$

$$\frac{\Sigma \vdash \qquad \Sigma \vdash \mathrm{WfArity}(\Psi) \qquad \Sigma; \overline{\Psi} \vdash \mathrm{WfConstr}(\Psi)}{\Sigma; \Psi \vdash}$$

FIG. 7.1: Règles de typage de la signature

En plus des vérifications déjà mentionnées dans la section précédente, nous vérifions que les types des arguments des constructeurs sont bien typés par la sorte Φ.CK. Si cette sorte est plus basse que Φ.S, cela signifie qu'il n'y a pas de "gros" constructeur et que nous n'avons pas à restreindre les sortes vers lesquelles nous pouvons faire des éliminations. Dans le cas contraire, seule l'élimination vers la sorte laplus basse Prop est autorisée, comme le définit la relation ElimSort.

7.6.3 Signature bien formée

Définition 7.48 (`wf_sign`) *Nous regroupons les définitions de la section précédente pour former un jugement vérifiant la bonne formation de toutes les déclarations qu'elle contient. Les règles sont présentées figure 7.1.*

Cette condition est à rapprocher de la définition Acceptable de Paulin dans [49].

7.6.4 Décidabilité du typage

Il reste une dernière condition à remplir, ce que nous ne pourrons faire puisqu'il s'agit de la normalisation forte. Nous admettons donc ce résultat, qui sera formulé ainsi :

Axiome 2

$$\frac{\Sigma \vdash \qquad \Sigma \mid \Gamma \vdash M : T}{M \in \mathcal{SN}^{\rightarrow \mathrm{CCI}}_{\Sigma,\Gamma}}$$

On devra admettre la normalisation forte du système (car cette fois, nous formalisons un système de puissance équivalente, bien que ce ne soit pas exactement le même, et donc le second théorème de Gödel s'applique très vraisemblablement), ce qui n'est pas une hypothèse tout à fait évidente. On n'est pas certain que le système proposé soit cohérent, mais on fait le pari que s'il s'avère incohérent, il n'y aurait pas beaucoup de travail pour le rendre cohérent.

On pourrait faire la preuve de normalisation forte à cet endroit si l'on était dans un système suffisamment puissant.

Avec cette condition, on peut prouver toutes les hypothèses faites dans la section précédente.

Lemme 7.49 (`wf_sign_ok`)

$$\Sigma \vdash \Rightarrow \models \Sigma$$

Preuve Par récurrence sur Σ. Ce résultat est facile car les conditions de $\models \Sigma$ sont incluses dans les diverses conditions des section précédentes, sauf la normalisation forte qui est admise. ■

Cela a pour conséquence le résultat-clé suivant :

Théorème 10 *Le typage de CCI restreint aux signatures bien formées est décidable.*

$$\forall \Sigma.\quad \Sigma \vdash \Rightarrow\ \mathrm{CCI}(\Sigma) \in \textsc{PtsTc}$$

$$\forall \Sigma, \Gamma, M, T.\quad \Sigma \vdash \Rightarrow\ \mathsf{Dec}\,(\Sigma \mid \Gamma \vdash M : T)$$

Algorithme 7.50 (`wf_def_dec`) *La bonne formation d'une définition globale dans Σ est décidable.*

$$\forall \xi \in \textsc{CstObj}.\, \mathsf{Dec}\,(\Sigma \vdash \mathrm{WfCst}(\xi))$$

Preuve C'est la conjonction de deux propositions décidables. La première l'est car la signature est supposée bien formée ($\models \Sigma$), et la deuxième se décide en appelant `search_global`, (lemme 6.11). ■

Lemme 7.51 (`wf_arity_dec`) *La vérification de l'arité des types inductifs est décidable.*

Preuve Il suffit de suivre la définition de WfArity, qui ne fait appel qu'à des prédicats décidables. Le typage est décidable puisque nous sommes dans une signature bien formée. ■

Lemme 7.52 (`wf_constr_dec`) *La bone formation des constructeurs d'une déclaration inductive est décidable. Pour éviter de refaire des vérifications, nous pourrons supposer que les arités sont bien typées.*

Algorithme 7.53 (`wf_sign_ext`) *L'extension bien formée d'une signature bien formée est décidable.*

$$\forall \Sigma, \sigma.\quad \Sigma \vdash \Rightarrow\ \mathsf{Dec}\,(\Sigma; \sigma \vdash)$$

Preuve Il suffit d'employer les trois algorithmes précedents en fonction de σ. ■

Et donc la bonne formation d'une signature est décidable.

Algorithme 7.54 (`wf_sign_dec`)

$$\forall \Sigma.\, \mathsf{Dec}\,(\Sigma \vdash)$$

Remarque : cela n'implique pas nécessairement que le typage est décidable puisque l'on peut dériver des jugements dans une signature mal formée, à partir du moment où on n'utilise pas les éléments mal formés. Plus grave, il peut y avoir plusieurs définitions pour un seul nom. Le jugement n'est probablement pas décidable. On se console en se disant que les signature pas bien formées n'ont aucun intérêt.

7.6.5 Construction du noyau

Notre jugement de typage est maintenant $\Sigma \mid \Gamma \vdash M : T$, ce qui fait que nous avons deux sortes d'environnements : l'un global (Σ), et l'autre local (Γ). Cela est redondant, puisque toutes les définitions que nous pouvons faire dans Γ peuvent être faites dans Σ par l'intermédiaire de l'opérateur $\mathrm{Cst}\{C\}$. Nous cacherons donc Γ dans le noyau et nous ferons comme si Σ était le seul environnement. Pour marquer ce fait, nous noterons $\Sigma \vdash M : T$ au lieu de $\Sigma \mid [] \vdash M : T$, et nous ferons de même pour tous les autres jugments.

Nous devons maintenant redéfinir l'interface du noyau de façon à substituer Σ à Γ et le jugement $\Sigma \vdash$ à $\Gamma \vdash$. Comme de plus, nous ne considérerons que des signatures bien formées, nous introduisons ce sous-ensemble pour représenter l'etat de notre système de preuves.

Définition 7.55 (type genv) *L'état de notre vérificateur de types sera une paire formé d'un environnement global Σ bien formé :*

$$\textsc{Genv} \stackrel{\text{def}}{\equiv} \{\Sigma \mid \Sigma \vdash\}$$

Définition 7.56 (type CCI_TC) *L'interface du noyau de notre système de preuves est :*

$$\begin{array}{lll}
\langle\ \texttt{inf_ppal_type}: & \forall G \in \textsc{Genv}.\forall M. & \mathsf{InfJdge}(G,\, M); \\
\quad \texttt{inf_ppal_sort}: & \forall G \in \textsc{Genv}.\forall x, M. & \mathsf{InfHyp}(G,\, x,\, M); \\
\quad \texttt{chk_typ}: & \forall G \in \textsc{Genv}.\forall M, T. & \mathsf{Dec}\,(G \vdash M : T)\,; \\
\quad \texttt{chk_decl}: & \forall G \in \textsc{Genv}.\forall \sigma. & (\exists^{\star} G' \in \textsc{Genv}.\, G' = G; \sigma) \ +\ \neg(G; \sigma \vdash); \\
\quad \texttt{chk_wk}: & \forall G \in \textsc{Genv}.\forall M, T. & T \in \mathcal{WT}_G \ \Rightarrow\ \mathsf{Dec}\,(G \vdash M : T)\,; \\
\quad \texttt{chk_wft}: & \forall G \in \textsc{Genv}.\forall M. & \mathsf{Dec}\,(M \in \mathcal{WT}_G) \\
\rangle &&
\end{array}$$

Théorème 11 (cci_kernel) *La signature précédente est instantiable.*

La construction de l'interface autour de ce noyau de fait exactement de la même manière que dans le développement des PTS, sauf que nous n'avons plus à rajouter la liste des noms associées aux objets globaux puisque Σ contient des noms uniques.

7.6.6 Utilisation du noyau

Malgré les deux axiomes calculatoires (algorithme de mise en forme normale de tête et décidabilité du sous-typage), nous avons extrait le noyau du vérificateur de preuves correspondant. Pour cela nous avons écrit (au sein de Coq) deux programmes non vérifiés pour instancier ces axiomes.

Le système résultant est utilisable, bien que peu d'effort ait été fait pour le rendre agréable à utiliser. Nous avons commencé à écrire un fichier de *prélude* regroupant les définitions de srtuctures de données élémentaires (booléens, entiers, connecteurs logiques, etc.).

Cette étape est indispensable, et nous fait parfois découvrir certaines faiblesses du formalisme et il faut revenir modifier les preuves. La première faiblesse que nous avons trouvée étant l'impossibilité de définir le type des entiers à l'aide de la déclaration suivante (rappelons que les types inductifs ont toujours exactement un paramètre et un argument, et que nous utilisons $\mathcal{M}^*$ comme argument fantôme) :

$$\begin{array}{l}
\Phi = \langle\ x = \texttt{nat};\ p = \mathcal{M}^*;\ a = \texttt{tyone};\ s = \mathrm{Set};\ ck = \mathrm{Set} \\
\qquad c = [\ \langle x = \texttt{0};\quad a = \mathcal{M}^*;\quad i = \varnothing\ \rangle; \\
\qquad\qquad \langle x = \texttt{S};\quad a = \texttt{nat};\quad i = \varnothing\ \rangle\ ]\ \rangle
\end{array}$$

Cela est dû au fait que $\mathcal{M}^*$ vit dans la sorte Prop qui ne peut être coercée vers la sorte des constructeurs (Set), car nous n'avions initialement pas inclus dans Set. Comme solution temporaire, nous avons choisi de rajouter cette inclusion[1].

Une meilleure solution serait d'assouplir un peu la formalisation des types inductifs de manière à autoriser les inductifs et les constructeurs à n'avoir aucun argument.

Mis à part ces quelques inconvénients, le système extrait montre des performances tout à fait acceptables, même si nous n'avons toujours pas résolu le problème mis à jour avec les PTS, à savoir que l'algorithme de conversion ne devrait pas expanser systèmatiquement les définitions.

[1] celle-ci n'est pas dans le système Coq actuel, mais la question se pose.

7.7 Autres extensions envisageables

Nous avons déjà proposé quelques extensions permettant d'améliorer sensiblement le confort d'écriture de fonctions par filtrage, en introduisant un motif attrape-tout et la possibilité de filtrer plusieurs avec la même branche. Nous avions aussi suggéré d'étendre la conversion de manière à pouvoir permuter deux branches dont les motifs ne se recouvrent pas (afin de préserver la confluence). Nous abordons maintenant des extensions plus profondes du système de types.

7.7.1 Types inductifs imbriqués

La version actuelle de Coq permet de définir le type des arbres de la manière suivante :

```
Coq < Inductive T : Set := node : (list T)->T.
```

Cette définition n'est pas acceptée par la condition de positivité décrite dans cette thèse, car le type `T` apparaît comme argument du type `list`. C'est ce qu'on appelle une occurrence *imbriquée*.

Le système reconnaît que si l'on expansait la définition des listes, nous obtiendrions une définition positive, donc cela n'introduit pas d'incohérence. Le fait que le système accepte cette définition n'implique pas pour autant que nous pourrons définir toutes les définitions récursives que l'on souhaiterait : Coq n'autorise pas les appels récursifs sur un élément arbitraire de la liste des fils. La définition suivante n'est pas acceptée :

```
Coq < Fixpoint size [t :T] : nat :=
Coq <   Cases t of
Coq <     (node l) => (S (sizel l))
Coq <   end
Coq < with sizel [l :(list T)] : nat :=
Coq <   Cases l of
Coq <     nil => O
Coq <   | (cons t l') => (plus (size t) (sizel l'))
Coq <   end.
Error during interpretation of command :
Fixpoint size [t :T] : nat :=
  Cases t of
    (node l) => (S (sizel l))
  end
with sizel [l :(list T)] : nat :=
  Cases l of
    nil => O
  | (cons t l') => (plus (size t) (sizel l'))
  end.
Error : Recursive call applied to an illegal term
 The 2-th recursive definition
 [l :(list T)]
  Cases l of
    nil => O
  | (cons t l') => (plus (size t) (sizel l'))
  end is not well-formed
```

Cependant, la définition suivante, calculant la hauteur de la branche la plus à gauche est correcte :

```
Coq < Fixpoint height [t :T] : nat :=
Coq <   Cases t of
Coq <     (node (cons t _)) => (S (height t))
```

```
Coq <   | (node nil) => 0
Coq <   end.
height is recursively defined
```

Ceci illustre bien la possible différence entre les occurrences positives (prédicat Pos) et les occurrences marquées (relation $\overset{\rho,L}{\gg}$). S'il est inutile de permettre le marquage d'occurrences non autorisées (puisque l'on ne marque que des types qui ont passé la condition de positivité), il est possible de ne pas marquer des occurrences reconnues comme positives. Le système semblera toutefois plus uniforme si ces deux définitions ne diffèrent pas.

Nous allons montrer que l'introduction des types inductifs imbriqués a de lourdes conséquences sur le système. En premier lieu, cela nous obligerait à étendre la relation de sous-typage, de manière à reconnaître comme covariante les occurrences imbriquées autorisée par la condition de positivité étendue. Avec la définition des arbres ci-dessus, pour pouvoir faire des appels récursifs avec n'importe quel fils, il faut marquer le type argument de `list` : si un arbre t possède la marque $+m$, la liste de ses fils devra être une liste d'arbres ayant la marque $-m$. Cela nous permettrait d'écrire la fonction calculant la taille d'un arbre comme ceci[2] :

```
Fixpoint size [t:T]: nat :=
  Cases t of
    node => [l:(list (T+m))](fold_right plus (map size l) 0)
```

Mais nous voulons pouvoir oublier la marque pour utiliser `l` comme une liste d'arbres quelconque. Il faut donc étendre le sous-typage de façon à reconnaître la covariance du constructeur de types list par rapport à son paramètre (si $A \leq A'$, alors $(\texttt{list}\ A) \leq (\texttt{list}\ A')$).

En l'état actuel, cela brise la résultat d'auto-réduction du filtrage. L'exemple suivant (comportant un motif non exaustif pour ne pas introduire trop de variables inutiles dans l'exemple) met en évidence le problème :

$$\langle Q\rangle \text{Cases}\ (\texttt{nil}\ A) : (\texttt{list}\ A')\ \text{of}\ \texttt{nil} \Rightarrow t\ \text{end} : (Q\ (\texttt{nil}\ A))$$

Nous filtrons la liste vide d'éléments de type A, que l'on voit comme une liste d'eléments de type A' (dans la mesure où A est un sous-type de A'). La branche t correspondand à nil doit donc être de type $(Q\ (\texttt{nil}\ A'))$. Or, le filtrage du terme $(\texttt{nil}\ A)$ aura le type $(Q\ (\texttt{nil}\ A))$. Comme le Case ci-dessus se réduit vers t, le résultat d'auto-réduction sera vérifié si $(Q\ (\texttt{nil}\ A'))$ est un sous-type de $(Q\ (\texttt{nil}\ A))$, ce que nous ne pouvons prouver dans le cas général.

Cela suggèrerait d'étendre l'égalité sur les constructeurs de manière à faire en sorte que les termes $(\texttt{nil}\ A')$ et $(\texttt{nil}\ A)$ soient convertibles, en ne tenant pas compte des paramètres des constructeurs. Cela ne devrait pas perturber la preuve de normalisation forte dans la mesure où les paramètres des constructeurs n'interviennent pas dans la réduction du filtrage.

Ce problème semble similaire à celui de considérer $\lambda x : A.\, M$ et $\lambda x : B.\, M$ convertibles. Mais si l'on identifie ces deux abstractions, on ne peut plus faire dépendre la réduction des types dans l'environnement.

Ce problème a été mis à jour grâce à la formalisation assez rapidement. La découverte d'un tel contre-exemple n'a pas porté à conséquence car on raisonnait à l'envers : au lieu de se rendre compte après coup que le système n'avait pas de bonnes propriétés, on a pu se rendre compte du problème et envisager des solutions, comme par exemple la conversion sans les paramètres.

[2] Noter que l'on n'est pas obligé d'écrire une fonction mutuellement récursive, mais nous pouvons réutiliser les fonctions standard sur les listes. Détecter que `map` n'applique la fonction passée en argument qu'aux éléments de la liste est assez difficile à faire syntaxiquement. Cela illustre bien l'avantage du système de marques sur la condition de garde syntaxique.

7.7.2 Types co-inductifs

Nous n'avons pas du tout traité le cas des types co-inductifs[3]. Ceux-ci permettent de définir des structures de données sans imposer que la relation de sous-structure soit bien fondée. Notamment, il est possible de définir les *flux*, qui sont des listes potentiellement infinies.

Si l'aspect inductif (la possibilité de faire du filtrage) est préservé avec les types co-inductifs, il est bien évidemment exclu de faire des raisonnement par récurrence structurelle, puisque comme nous l'avons dit, ces structures peuvent être de "hauteur" infine. Cela nous empêche donc d'écrire des fonctions par point fixe dont l'argument récursif serait un objet co-inductif. Nous disposons en revanche d'un co-point fixe pour construire des objets de taille éventuellement infinie. Ce co-point fixe se comporte calculatoirement comme l'autre point fixe, mais l'expansion est gardée différemment : un co-point fixe ne s'expanse que s'il est en argument d'un Case, puisqu'à ce moment, nous avons besoin de "dégeler" l'objet co-inductif pour observer quel est son constructeur de tête. La terminaison est assurée par le fait qu'un étape d'expansion du co-point fixe fait appraître au moins un constructeur.

Cette nouvelle condition de garde peut elle aussi se vérifier syntaxiquement (comme dans la version actuelle de Coq), ou à l'aide du systèmes de marques comme dans [25]. Pour suivre ce qui est proposé dans cet article, il faut modifier le typage des constructeurs de façon à ce qu'ils propagent les marques : si un constructeur est appliqué uniquement à des objets portant une marque $-m$ (en ne tenant compte que des arguments marquables, cf $\overset{\rho,L}{\gg}$), alors le constructeur pourra porter la marque $+m$: si tous les sous-termes d'un constructeur sont de taille strictement inférieur à m, alors le constructeur est de taille inférieure au sens large à m. Pour un co-point fixe construisant un objet de type I, nous introduisons dans le contexte une variable f avec le type I^{-m} (permettant de faire des appels récursifs), et nous vérifions que le corps du co-point fixe est de type I^{+m}, ce qui assure qu'il ajoute un constructeur au dessus de f.

Nous pouvons définir la liste infinie formée de zéros (`stream` étant défini comme un type coinductif ayant un constructeur `cons` identique à celui des listes) :

```
(zeros: stream) = CoFix([m:Mark; zeros: stream-m](cons 0 zeros):stream+m).
```

Pour permettre au co-point fixe d'ajouter plus d'un niveau de constructeur par expansion, il suffit d'ajouter la règle de sous-typage $I^{+m} \leq I^{-m}$ pour tout type co-inductif I. Cette inclusion est l'inverse de celle des types inductif.

Un raffinement supplémentaire consisterait à ne pas distinguer seulement "+" ou "-", mais avoir un entier (relatif) qui compte le nombre de constructeurs destructurés ou rajoutés. Pour les types inductif $+m$ serait noté (k, m) avec $k \leq 0$ et $-m$ serait (k, m) avec $k < 0$. En revanche pour les coinductifs, $+m$ correspondrait à $k > 0$ et $-m$ à $k \geq 0$.

Cela permettrait de tenir compte des cas où l'on destructure, puis reconstruit des objets coinductifs, sachant que l'on fait au moins une construction de plus que de destructurations.

[3] Nous ne considérons pas les types coinductifs comme une fonctionalité mineure de Coq, introduits dans Coq par Gimenez [24]. Ceux-ci sont particulièrement utiles pour formaliser les systèmes réactifs. La seule raison du choix de ne pas les formaliser est que la preuve présentée dans cette thèse ne les utilise pas, et donc on peut se diriger vers une version bootstrappée de Coq sans types co-inductifs.

Chapitre 8

Conclusion

Il est maintenant temps de faire un bilan du travail accompli lors de cette thèse, et d'esquisser quelle suite lui donner. Nous distinguons plusieurs registres de remarques :

- concernant l'utilisabilité de Coq. Cette preuve, qui est assez volumineuse, nous permet d'apprécier les fonctionalités importantes, celles qui manquent, ainsi que les défauts d'architecture du système.
- faire un bilan du formalisme de Coq : évaluer l'écart entre les présentations informelles et formelles.
- critique du système de type proposé. Nous avons déjà formulé quelques propositions d'extensions du formalisme dans le chapitre précédent, nous ne reviendrons pas sur ce point.

Globalement, le bilan est positif sur la plupart des points ci-dessus : nous avons quasiment atteint notre objectif de produire un noyau de système de preuve certifié et utilisable, basé sur un formalisme puissant. Les preuves non menées à termes l'ont été plus par manque de temps que du fait d'une réelle difficulté à les produire.

En tout cas, cela montre que les systèmes de preuves modernes permettent le développement de preuves de programmes de taille et de complexité considérables.

8.1 Développement en Coq

La spécification d'un système de type comme celui de Coq se fait de manière relativement naturelle. Cela est dû au fait que nous disposons d'une logique d'ordre supérieure. Les types inductifs apportent un confort d'utilisation réellement considérable, tant pour définir les structures de données manipulées par les programmes à certifier, que pour définir des prédicats à la Prolog.

Les preuves nous ont permis de trouver de manière indirecte des incomplétudes dans le code de Coq : en formalisant la théorie, certains points posaient quelques problèmes. La confrontation avec la solution adoptée dans le code a permis de découvrir quelques incomplétudes, i.e. certaines preuves correctes pourrait être refusées. Ainsi, le simple fait de formaliser permet de trouver des problèmes, même sans effectivement se servir du code extrait.

Contrairement à une idée fortement répandue, il n'est pas absolument nécessaire que la preuve soit déjà prouvée sur papier avant de la passer en Coq. Une idée relativement précise de comment faire la preuve est souvent suffisante. Evidemment, avoir la preuve faite sur

papier peut servir à ne pas s'écarter du schéma de preuve que l'on a en tête. Cette thèse en témoigne puisqu'un bon nombre de résultats sont inédits. Il est est en revanche préférable d'avoir une forte conviction : Coq ne sert pas à savoir si un théorème est vrai ou non. Un autre point pour lequel le système est bien adapté est le renforcement de théorèmes déjà prouvés (i.e. déterminer des hypothèses plus simples à prouver).

Nous avons expérimenté un certain nombre de techniques de preuves. Nous avons vu qu'un développement modulaire, où l'on pose temporairement des axiomes que l'on récapitule à l'aide de structures, permettait d'avoir un style de preuve inversé, ce qui est parfois plus commode que le style direct.

Certaines techniques de preuve se sont montrées très pratiques, comme par exemple l'utilisation d'axiomes temporaires.

Il est souvent pratique de travailler dans des contextes où l'on a des axiomes, voire des hypothèses incohérentes. Cela n'est pas très dangereux dans la mesure où il y a peu d'automatisation et l'utilisateur ne perd pas le contrôle. Tant que le système ne se rend pas compte de l'incohérence, tout va bien. Dès qu'il s'en rend compte tout devient trivial à prouver, et il faut réajuster les hypothèses (sauf si l'on s'est mis volontairement dans un contexte incohérent pour réfuter une des hypothèses faites).

Nous retiendrons quelques principes d'utilisation de Coq. La liste mériterait d'être allongée afin d'aider les nouveaux utilisateurs à se familiariser plus facilement au style du système.

- les scripts les plus courts sont généralement les plus robustes (et ils nous obligent à faire des étapes de raisonnement plus petites, ce qui nous rapproche du style déclaratif).
- ne jamais tolérer qu'un script de preuve d'un théorème simple soit très long. Si le script du premier jet est trop long, repasser le script, voir où il y a des redondances, des sous-parties qui pourraient être prouvées séparément. Cela peut nécessiter aussi de nouvelles astuces de preuve. Même si cela peut paraître fastidieux, voir relever de la maniaquerie, nous avons noté que cela permettait vraiment de comprendre les théorèmes de la manière la plus épurée possible, ce qui rend une explication informelle de ce qu'il se passe beaucoup plus simple.

L'un des plus gros reproches que l'on peut faire, c'est que les scripts de preuve sont peu modulaires, et sont relativement instables vis-à-vis de changements mineurs, sauf si l'on prend un certain nombre de précautions, ce qui s'acquiert avec l'expérience. En revanche, on peut mettre en évidence des énoncés de théorèmes qui seront très peu sensibles aux changements du formalisme. Cela suggère fortement qu'il est préférable d'avoir un langage de tactiques plutôt déclaratif.

Le mécanisme de section est une forme de modularité qui s'avère insuffisant lorsque l'on a des paramètres que l'on conserve longtemps. En effet, une section ne peut s'étendre sur plusieurs fichiers que l'on compilerait de manière séparée. Pour cela, un système de modules avec foncteurs aurait été d'un grand secours. Une autre inconvénient des sections est qu'après le déchargement des hypothèses locales à la section, les preuves sont quantifiées universellement de manière individuelle, ce qui oblige à faire de très nombreux passages d'arguments supplémentaires entre les constantes de la section.

8.2 Sur les formalismes décrits

Nous espérons que cette présentation aidera à mieux comprendre le système de Coq, en particulier la partie sur les types inductifs, qui ont toujours eu la réputation d'être assez volumineux. La présentation faite ici reste tout de même assez technique, mais nous

espérons que les explications informelles précédant les théorèmes formels suffiront déjà à faire comprendre les grandes lignes du formalisme. Nous avons porté un soin particulier à ne pas trop nous éloigner de ces grandes lignes.

Les algorithmes prouvés ne sont pas tous très efficaces, mais le développement est suffisamment modulaire pour permettre de réparer localement les sources d'inefficacité. En revanche, l'architecture semble tout à fait raisonnable. Un intérêt conséquent de ce travail est de fournir un modèle de preuves que l'on peut réutiliser.

8.3 Bootstrap

Il y a une grande similitude entre le langage mathématique employé dans cette thèse (et décrit en introduction) et le système logique de Coq (présenté informellement dans le premier chapitre). Cette similitude est évidemment volontaire afin d'aboutir au bootstrap. Nous espérons avoir pris suffisamment de précautions pour que ces différents niveaux où les mêmes concepts apparaissent ne se téléscopent pas dans l'esprit du lecteur.

En principe, il devrait être possible de traduire les développements de cette thèse dans le formalisme décrit dans les chapitres 6 et 7 sans avoir à faire de modifications trop complexes. Essentiellement, il s'agirait d'η-allonger les constructeurs et les types inductifs, et de rajouter les marques dans les corps des points fixes. La traduction d'un terme bien gardé (suivant la condition syntaxique comme elle est définie dans le système Coq actuel) en un terme annoté avec des marques doit pouvoir se faire automatiquement.

L'extraction devrait être formalisée afin que le système formalisé soit complètement assimilable à celui de Coq. Il serait alors possible de faire le bootstrap, c'est à dire la génération par notre système du code source certifié de sa propre implantation.

Nous envisageons réellement le bootstrap comme un objectif à atteindre. Il permettrait de développer les modifications apportées au noyau de façon sûre. Ainsi la programmation de Coq ne se ferait plus en Objective Caml, mais en Coq. Autrement dit, le source de Coq ne serait plus un ensemble de fichiers Objective Caml, mais un développement Coq. Objective Caml jouerait le rôle du langage machine, comme environnement d'exécution d'un programme exprimé dans un langage de haut niveau.

Pour que cela soit réalisable, il faudrait améliorer l'ergonomie du système formalisé en recollant les différentes parties de cette thèse : intégrer les messages d'erreur aux fonctions de typage, formaliser des fonctions de réduction efficaces, avoir un mécanisme de section ou de modules, écrire des tactiques aidant à la construction des preuves. Cette dernière tâche étant la plus facile, puisqu'il n'est pas nécessaire de faire de preuves de corrections, grâce à l'architecture basée sur un noyau.

Annexe A

Relations et ordres bien fondés

Soit un type X et une relation R sur ce type.

Définition A.1 (`Acc`) *L'ensemble des éléments de X accessibles pour la relation R est le plus petit ensemble Y vérifiant la condition :*

$$\forall x.\,(\forall y.\,y\ R\ x\ \Rightarrow\ y \in Y)\ \Rightarrow\ x \in Y$$

Cet ensemble sera noté Acc_R. L'ordre R sera dit bien fondé si tous les éléments de X sont accessibles.

Définition A.2 (prédicat `Lexi`) *Ordre lexicographique au niveau i ($f, g \in X^{\mathbb{N}}$) :*

$$f \prec_i^R g \ \stackrel{\text{def}}{\equiv}\ \exists j{<}i.\ \begin{cases} \forall k{<}j.\, f(k)\ R^*\ g(k) \\ f(j)\ R\ g(j) \\ \forall k{>}j.\, f(k) \in \mathrm{Acc}_R \end{cases}$$

Théorème A.3 (`Lexi_acc`) *Si toutes les images par f des entiers strictement inférieurs à un entier i sont accessibles pour R, alors f est accessible pour $\prec_i^R$:*

$$(\forall k{<}i.\, f(k) \in \mathrm{Acc}_R)\ \Rightarrow\ f \in \mathrm{Acc}_{\prec_i^R}$$

Ce résultat a comme corollaire immédiat :

Théorème A.4 (`Lexi_wf`) *Si R est bien fondé, alors pour tout i, $\prec_i$ est bien fondé.*

Evidemment, $\bigcup_{i\in\mathbb{N}} \prec_i^R$ n'est pas necéssairement bien fondé. Par exemple, la suite de fonctions

$$f_j(k) = \begin{cases} 0 & \text{si } k{<}j \\ 1 & \text{sinon} \end{cases}$$

est une suite infinie décroissante pour la réunion des $\prec_i^<$, alors que pour tout j et k, $f_j(k)$ est accessible pour $<$.

Définition A.5 *La séquence de deux ordres R_1 et R_2 :*

$$x\ (R_1; R_2)\ z\ \stackrel{\text{def}}{\equiv}\ \exists y.\, x\ R_1\ y\ \wedge\ y\ R_2\ z$$

Définition A.6 *On dira que R_1 commute avec R_2 si $R_2; R_1 \subseteq R_1; R_2$.*

Définition A.7 (`union`) *L'union de deux ordres R_1 et R_2 :*

$$x\ (R_1 \cup R_2)\ y \stackrel{\text{def}}{\equiv} x\ R_1\ y \vee x\ R_2\ y$$

Théorème A.8 *Si :*

- $R_2; R_1 \subseteq R_1^*; R_2^+$
- R_1 *bien fondé*
- $x \in \mathrm{Acc}_{R_2}$

alors $x \in \mathrm{Acc}_{R_1 \cup R_2}$.

Annexe B

Fonctions de réduction

B.1 Termes purs

```
(* Les lambda-termes purs *)
type lambda =
  | Rel of int
  | Lam of lambda
  | App of lambda * lambda
```

B.2 Substitutions

```
(* Les fonction de relocation et de substitution *)
let rec shift_rec n k = function
  | Rel i    -> if i<k then Rel i else Rel (i+n)
  | Lam t    -> Lam (shift_rec n (k+1) t)
  | App(a,b) -> App(shift_rec n k a, shift_rec n k b)

let shift n k t = if n = 0 then t else shift_rec n k t

let rec subst_rec n k = function
  | Rel i    -> if i<k then Rel i
                else if i=k then (shift k 0 n)
                else Rel (i-1)
  | Lam t    -> Lam (subst_rec n (k+1) t)
  | App(a,b) -> App(subst_rec n k a, subst_rec n k b)

let subst n t = subst_rec n 0 t

let rec hnf = function
  | App(a,b) -> (match (hnf a) with
                 | Lam f -> hnf (subst b f)
                 | a'    -> App(a',b))
  | t -> t
```

```
let rec strong f t =
  match (f t) with
  | Rel i    -> Rel i
  | Lam u    -> Lam (strong f u)
  | App(a,b) -> App(strong f a, strong f b)

let norm = strong hnf
```

B.3 Machine KN récursive terminale

```
type clos = Clos of env * lambda
 and env  = clos list

let rec clos_rel e i =
  match (e,i) with
  | ((c::_), 0) -> c
  | ((_::e), n) -> clos_rel e (n-1)
  | ([], n)     -> Clos([], Rel n)

(* Optimisation: ne jamais enfermer une variable seule *)
let mk_clos e = function
  | Rel i -> clos_rel e i
  | b     -> Clos(e,b)

(* Le type des contextes de termes *)
type stk =
  | Ztop
  | Zappl of clos * stk
  | Zappr of lambda * stk
  | Zlam of stk

(* Machine KN *)
let rec kn_tr k env t stk =
  match (t,stk) with
  | (App(a,b),_) -> kn_tr k env a (Zappl (mk_clos env b, stk))
  | (Lam f, Zappl(arg,s)) -> kn_tr k (arg::env) f s
  | (Lam f, s) ->
      let nk = k+1 in
      let nvar = Clos([], Rel (-nk)) in
        kn_tr nk (nvar::env) f (Zlam s)
  | (Rel i, _) -> kn_tr_rel k env i stk

(* Cas particulier des variables *)
and kn_tr_rel k env i stk =
  match (i,env) with
  | (0, (Clos(e,t)::_)) -> kn_tr k e t stk
  | (_, (_::e))         -> kn_tr_rel k e (i-1) stk
  | (_, [])             -> zip_tr k (Rel (i+k)) stk
```

```
(* Reconstruction du terme *)
and zip_tr k t stk =
  match stk with
  | Ztop                -> t
  | Zappl(Clos(e,u), s) -> kn_tr k e u (Zappr(t,s))
  | Zappr(a,s)          -> zip_tr k (App(a,t)) s
  | Zlam s              -> zip_tr (k-1) (Lam t) s

let norm_kn_tr t = kn_tr 0 [] t Ztop
```

B.4 Relocations et substitutions

Il est intéressant de noter que dans la suite, on manipulera les relocations et les substitutions de manière totalement abstraite : on dispose de quelques constructeurs, correspondant aux modifications subies lorsque l'on parcours un terme, et d'un destructeur, qui effectue l'opération de relocation ou de substitution sur les variables.

```
type ('a,'b) sum = Inl of 'a | Inr of 'b

(* Relocations *)
type elift = Lid | Llift of int * elift | Lshift of elift * int

(* Les constructeurs *)
let lid = Lid

let llift el =
  match el with
  | Lid -> Lid
  | Llift (k,e) -> Llift(k+1, e)
  | _ -> Llift(1,el)

let lshift k el =
  match (k,el) with
  | (0,_) -> el
  | (_,Lshift(e,n)) -> Lshift(e,n+k)
  | _ -> Lshift(el,k)

(* Le destructeur *)
let rec reloc_db acc el n =
  match el with
  | Lid -> n+acc
  | Llift(k,e) -> if n >= k then reloc_db (acc+k) e (n-k) else n
  | Lshift(e,k) -> reloc_db acc e (n+k)
let reloc_rel el n = reloc_db 0 el n


(* Substitutions polymorphes *)
type 'a subs =
  | Sid
  | Slift of int * 'a subs
```

```
  | Sshift of int * 'a subs
  | Scons of 'a subs * 'a

(* Les constructeurs *)
let subs_id = Sid

let subs_lift = function
  | Sid -> Sid
  | Slift(n,s) -> Slift(n+1,s)
  | s -> Slift(1,s)

let subs_shift_cons = function
  | (0, s, t) -> Scons(s,t)
  | (k, Sshift(n,s1), t) -> Scons(Sshift(k+n, s1), t)
  | (k, s, t) -> Scons(Sshift(k, s), t)

(* Le destructeur *)
let rec exp_rel lams s n =
  match s with
  | Sid -> Inr (lams+n)
  | Scons(st,t) ->
      if n=0 then Inl (lams,t) else exp_rel lams st (n-1)
  | Slift(k,e) ->
      if n >= k then exp_rel (lams+k) e (n-k) else Inr (lams+n)
  | Sshift(k,e) -> exp_rel (lams+k) e n
let expand_rel s n = exp_rel 0 s n
```

L'interface est la suivante :

```
type ('a, 'b) sum = | Inl of 'a | Inr of 'b

type elift
val lid : elift
val llift : elift -> elift
val lshift : int -> elift -> elift
val reloc_rel : elift -> int -> int

type 'a subs
val subs_id : 'a subs
val subs_lift : 'a subs -> 'a subs
val subs_shift_cons : int * 'a subs * 'a -> 'a subs
val expand_rel : 'a subs -> int -> (int * 'a, int) sum
```

B.5 Termes avec effets de bord

```
type red_state = Norm | Red

type mterm = { mutable norm: red_state; mutable term: fterm }
```

```
 and fterm =
  | Frel
  | Fapp of mterm * mterm
  | Flam of mterm * mterm subs * lambda
  | Fshift of int * mterm
  | Fclos of mterm subs * lambda

let is_norm v = v.norm = Norm
let set_norm v = v.norm <- Norm
let new_clos t = { norm = Red; term = t }
let frel = { norm = Norm; term = Frel }

let rec lift_fterm n ft =
  match (n,ft.term) with
  | (0, _)           -> ft
  | (_, Fshift(k,m)) -> lift_fterm (n+k) m
  | _                -> { norm = ft.norm; term = Fshift(n,ft) }

let clos_rel e i =
  match expand_rel e i with
  | Inl(n,mt) -> lift_fterm n mt
  | Inr n     -> lift_fterm n frel

(* Optimisation: ne jamais enfermer une variable seule *)
let mk_clos e t =
  match t with
  | Rel i -> clos_rel e i
  | t     -> new_clos (Fclos(e,t))

type ('a, 'b) stk =
  | Ztop
  | Zappl of 'a * ('a, 'b) stk
  | Zappr of 'b * ('a, 'b) stk
  | Zlam of mterm subs * lambda * ('a, 'b) stk
  | Zshift of int * ('a, 'b) stk
  | Zupdate of mterm * ('a, 'b) stk

let zshift n s =
  match (n,s) with
  | (0,_)           -> s
  | (_,Zshift(k,s)) -> Zshift(k+n,s)
  | _               -> Zshift(n,s)

let zupdate (m,stk) =
  Zupdate(m,stk)

(* Attention: les mises a jour sur v2 ne profiteront plus a v1 *)
let update v1 v2 =
  v1.norm <- v2.norm;
  v1.term <- v2.term;
```

```
  v1
```

B.5.1 Variantes

Effets de bord

Sans effet de bord (en fait la définition de `zupdate` fait qu'aucune marque de mise à jour n'est créée, et donc la fonction `update` n'est jamais appelée) :

```
let set_norm v = ()
let zupdate (m,stk) = stk
let update v1 v2 = v2
```

Booléen indiquant si un terme est en forme normale

Sans les raccourcis (le fait de supprimer les effets de bords supprime les raccourcis non triviaux, i.e. les variables) :

```
let is_norm v = false
```

GC des références non partagées

Avec les *weak pointers* :

```
type fterm =
  ...
  | Zupdate of mterm Weak.t * ('a, 'b) stk
let zupdate (m,stk) =
  let wp = Weak.create 1 in
  let _ = Weak.set wp 0 (Some m) in
  Zupdate(wp,stk)
let update wp v2 =
  match Weak.get wp 0 with
    Some v1 ->
      v1.norm <- v2.norm;
      v1.term <- v2.term;
      v1
  | None -> v2
```

Sur de petits exemples, c'est en fait moins efficace, car les structures sont plus grosses (il y a une indirection supplémentaire), mais sur de très gros exemples, on y gagne (en mémoire).

B.6 Machine lazy

```
let simpl_stk h stk =
  let rec stk_rec depth = function
    | Zshift(k,s)  -> stk_rec (k+depth) s
    | Zupdate(m,s) ->
        let _ = update m (lift_fterm depth h) in
        stk_rec depth s
    | s            -> (depth, s) in
  stk_rec 0 stk
```

```
let rec klnt e t stk =
  match t with
  | Rel i    -> kln (clos_rel e i) stk
  | App(a,b) -> klnt e a (Zappl (mk_clos e b, stk))
  | Lam f    -> kln (new_clos (Flam(mk_clos (subs_lift e) f, e, f))) stk

and kln m stk =
  match m.term with
  | Fclos(e,f)    -> klnt e f (zupdate(m,stk))
  | Fshift(k,a)   -> kln a (zshift k stk)
  | Frel          -> zip frel stk
  | Fapp(a,b)     ->
      if is_norm m then zip m stk (* raccourci *)
      else kln a (Zappl (b, zupdate(m,stk)))
  | Flam (bd,e,f) ->
      (match simpl_stk m stk with
      | (depth, Zappl(arg,s)) -> klnt (subs_shift_cons(depth,e,arg)) f s
      | (depth, s)            ->
          let stk' = zshift depth s in
          if is_norm bd
          then (set_norm m; zip m stk') (* petit raccourci *)
          else kln bd (Zlam (e, f, stk')))

(* Precondition: m.norm=Norm, donc m est en forme normale *)
and zip m stk =
  match stk with
  | Ztop          -> m (* Arret de la machine *)
  | Zappr(a,s)    -> zip {norm=Norm; term=Fapp(a,m)} s
  | Zlam (e,f,s)  -> zip {norm=Norm; term=Flam(m,e,f)} s
  | Zshift(k,s)   -> zip (lift_fterm k m) s
  | Zupdate(m',s) -> zip (update m' m) s
  | Zappl(a,s)    ->
      let s' = Zappr(m,s) in
      if is_norm a then zip a s' (* petit raccourci *)
      else kln a s'


let to_lambda t =
  let rec lbda_rec lfts m stk =
    match m.term with
    | Frel        -> zip_lbda (Rel (reloc_rel lfts 0)) stk
    | Fapp(a,b)   -> lbda_rec lfts a (Zappl ((lfts,b), stk))
    | Flam(u,e,f) -> lbda_rec (llift lfts) u (Zlam (e,f,stk))
    | Fshift(k,m) -> lbda_rec (lshift k lfts) m stk
    | Fclos _     -> failwith "to_lambda: found a closure"
  and zip_lbda t = function
    | Ztop                -> t
    | Zappr(u,s)          -> zip_lbda (App(u,t)) s
    | Zlam(_,_,s)         -> zip_lbda (Lam t) s
```

```
    | Zappl((lfts,b),s)    -> lbda_rec lfts b (Zappr(t,s))
    | Zshift _ | Zupdate _ -> assert false
  in lbda_rec lid t Ztop

let norm_kl t =
  to_lambda (klnt subs_id t Ztop)
```

Bibliographie

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, January 1990. Also Digital Equipment Corporation, Systems Research Center, Research Report 54, February 1990.

[2] T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993.

[3] F. Barbanera and S. Berardi. Proof-irrelevance out of excluded middle and choice in the calculus of constructions. *J. of Functional Programming*, 6(3) :519–525, 1996.

[4] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[5] B. Barras. Coq en coq. Rapport de Recherche 3026, INRIA, October 1996.

[6] B. Barras. Verification of the interface of a small proof system in coq. In E. Gimenez and C. Paulin-Mohring, editors, *Proceedings of the Workshop on Types for Proofs and Programs TYPES'96*, volume 1512 of *LNCS*, pages 28–45, Aussois, France, December 1996. Springer-Verlag.

[7] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[8] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual version 6.1. Technical Report 0203, Projet Coq-INRIA Rocquencourt-ENS Lyon, August 1997.

[9] S. Boutin. *Réflexions sur les quotients.* thèse d'université, Paris 7, April 1997.

[10] S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[11] T. Coquand. *Une Théorie des Constructions.* PhD thesis, Université Paris 7, January 1985.

[12] T. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[13] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[14] C. Cornes. *Conception d'un Langage de Haut Niveau de Représentation de Preuves : récurrence par filtrage de motifs, unification en présence de types inductif primitifs. synthèse de lemmes d'inversion.* Thèse d'université, Paris 7, November 1997.

[15] Y. Coscoy, G. Khan, and L. Théry. Extracting text from proofs. Rapport de recherche 2459, INRIA, January 1995.

[16] P. Crégut. An abstract machine for λ-terms normalization. In Gilles Kahn, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 333–340, Nice, France, June 1990. ACM Press.

[17] P. Crégut. *Machines à Environnements pour la Réduction Symbolique et l'Évaluation Partielle*. Thèse de doctorat, Université Paris VII, June 1991.

[18] Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2) :362–397, March 1996.

[19] N.J. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34 (5), pp. 381–392, 1972.

[20] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[21] H. Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the Workshop on Types for Proofs and Programs TYPES'94*, volume 996 of *LNCS*, pages 14–38. Springer-Verlag, Båstad, Sweden, 1995.

[22] H. Geuvers and M.-J. Nederhof. A modular proof of strong normalization for the Calculus of Constructions. *J. of Functional Programming*, 1(2) :155–189, 1991.

[23] E. Gimenez. Codifying guarded definitions with recursive schemes. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 39–59. Springer-Verlag LNCS 996, 1994.

[24] E. Gimenez. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. Thèse de doctorat, ENS-Lyon, December 1996.

[25] E. Gimenez. Structural recursive definitions in type theory. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 397–408, Aalborg, Denmark, 1998. Springer-Verlag LNCS 1443.

[26] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[27] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1989.

[28] M.J. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. LNCS 78. Springer-Verlag, 1979.

[29] T. Hardin, L. Maranget, and B. Pagano. Functional back-ends within the lambda-sigma calculus. In *ACM Sigplan International Conference on Functional Programming, Philadelphia*, 1996.

[30] R. Harper and R. Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1) :107–136, October 1991.

[31] G. Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science*. WorldScientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[32] G. Huet. Residual theory in λ-calculus : a formal development. *J. of Functional Programming*, 4,3 :371–394, 1994.

[33] G. Huet. The zipper data structure. *Journal of Functional Programming*, 1997. À paraitre.

[34] J.-M. Hullot. *Compilation de Formes Canoniques dans des Théories Équationnelles.* Thèse de doctorat, Université Paris-Sud, Orsay, November 1980.

[35] J. Mc Kinna and R. Pollack. Pure type systems formalized. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 289–305, 1993.

[36] P. Lescanne. From $\lambda\sigma$ to λv : A journey through calculi of explicit substitutions. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 60–69, Portland, Oregon, January 17–21, 1994. ACM Press.

[37] J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul.* thèse de doctorat d'état, Paris 7, 1978.

[38] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[39] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[40] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[41] P.-A. Melliès and B. Werner. A generic normalisation proof for pure type systems. In E. Gimenez and C. Paulin-Mohring, editors, *Proceedings of the Workshop on Types for Proofs and Programs TYPES'96*, volume 1512 of *LNCS*, pages 254–276, Aussois, France, December 1996. Springer-Verlag.

[42] N. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1988.

[43] C. Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie des types.* Thèse de doctorat, Université Paris 7, November 1997.

[44] B. Nordström. *Terminating General Recursion*, volume 28. BIT, 1988.

[45] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS : Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[46] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures : Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2) :107–125, February 1995.

[47] C. Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Mathematics of Program Construction'95.* Springer-Verlag LNCS 947, 1995.

[48] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse de doctorat, Paris 7, January 1989.

[49] C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[50] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15 :607–640, 1993.

[51] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3 :119–149, 1983.

[52] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2 :325–355, 1986.

[53] Lawrence C. Paulson. *Logic and Computation : Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[54] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[55] Erik Poll. Expansion Postponement for Normalising Pure Type Systems. *Journal of Functional Programming*, 8(1) :89–96, January 1998.

[56] R. Pollack. *The Theory of Lego : a Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[57] R. Pollack. On extensibility of proof checkers. In P. Dybjer, B. Nordström, and J. Smith, editors, *Proceedings of the Workshop on Types for Proofs and Programs TYPES'94*, volume 996 of *LNCS*, pages 140–161, Båstad, Sweden, 1995. Springer-Verlag.

[58] R. Pollack. A verified typechecker. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 365–380, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.

[59] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In H. Barendregt and T.Nipkow, editors, *Proceedings of the 1993 Workshop on Types for Proofs and Programs*, pages 19–61, Nijmegen, The Netherlands, May 1993. Springer-Verlag LNCS 806.

[60] B. Werner. *Une Théorie de Constructions Inductives.* Thèse de doctorat, Université Paris 7, May 1994.

Table des figures

Index

Résumé :

Les systèmes de preuves sont des programmes permettant de vérifier automatiquement la validité de preuves mathématiques exprimées dans un certain formalisme logique. Une des applications principales de ces systèmes est la certification de logiciels. Comme pour tous les programmes, on peut se poser la question de la correction de l'implantation du système de preuves lui-même, puisqu'il est censé garantir la correction d'autres programmes.

Dans cette thèse, nous formalisons et vérifions les algorithmes mis en œuvre dans l'implantation d'un système de preuves comme Coq. Tout d'abord cela permet d'accroître la confiance en ce système, en éliminant pratiquement tout risque d'une implantation non fidèle au formalisme logique que Coq est censé implanter, le Calcul des Constructions Inductives. D'autre part, le choix de faire cette vérification à l'aide de Coq nous permet de constater la praticabilité de la méthode de développement de programmes certifiés en Coq à grande échelle.

De par le formalisme constructif de Coq, il est possible d'engendrer automatiquement (par extraction) le source en ML des algorithmes certifiés, réduisant encore les risques d'erreurs lors de la transcription des algorithmes dans le système de preuves. Nous avons procédé à l'extraction du code d'un vérificateur de preuves, en ayant toutefois admis la correction de quelques algorithmes. Cela donne lieu à une implantation d'efficacité tout à fait satisfaisante.

Mots clés :

Logique, Vérification de Programmes, Théorie des Types, Calcul des Constructions Inductives, Extraction, Bootstrap, Machines Abstraites de Réduction.

Abstract :

Proof systems are programs that allow the automatic checking of the correctness of mathematical proofs, expressed in a given logical formalism. One of the main application of these systems is software verification. As with any program, we may be concerned about the existence of errors in the implementation of the system itself, particularly since it is supposed to guarantee the correctness of other programs.

In this thesis, we formalise and verify the algorithms involved in the implementation of a proof system such as Coq. Firstly, this increases the reliability of the system, by virtually banning any chance of a disagreement between the implementation and the intended logical formalism, the Calculus of Inductive Constructions. Secondly, the choice of doing so using Coq allows us to test the feasibility of software verification in the large using this tool.

Thanks to the constructive formalism of Coq, it is possible to automatically generate (by extraction) the ML source of the verified algorithms, further reducing the chance of error during the transcription of the actual algorithms in the proof system. We performed the extraction of the code for a proof-checker, while assuming the soundness of several algorithms. This resulted in an implementation with very satisfying efficiency.

Keywords :

Logic, Software Verification, Type Theory, Calculus of Inductive Constructions, Extraction, Bootstrap, Reduction Abstract Machines.