THÈSE DE DOCTORAT

présentée

À L'UNIVERSITÉ PARIS 6 - PIERRE ET MARIE CURIE

pour l'obtention du Diplôme de

DOCTEUR DE L'UNIVERSITÉ PARIS 6
SPÉCIALITÉ : INFORMATIQUE FONDAMENTALE

présentée et soutenue publiquement
par

David DELAHAYE

le 20 décembre 2001

Titre :

Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve

Une étude dans le cadre du système Coq

Directeurs de thèse :

Mme Christine PAULIN-MOHRING
M. Benjamin WERNER

Jury :

Mme	Thérèse HARDIN	Présidente
Mme	Catherine DUBOIS	Rapporteurs
M.	Yves BERTOT	
M.	Pierre CRÉGUT	Examinateurs
M.	James McKINNA	

À mes parents, à Benjamin
A Micaela, a los Mayero,
sans oublier...
Pierrot, Roméo y Quica

Remerciements

Je tiens, tout d'abord, à remercier vivement Benjamin Werner pour avoir accepté de diriger ma thèse et de m'avoir proposé ce sujet qui m'a occupé pendant ces quatre dernières années. Benjamin a su me guider dans ce domaine finalement très large, tout en me laissant une initiative totale, et, en particulier, il est parvenu à contenir une certaine frénésie de code, qu'il n'est pas rare d'*attraper* lorsque l'on fait un tour du côté du bâtiment 8 *virtuel* de l'INRIA-Rocquencourt. Benjamin m'a également permis de prendre quelques repères dans le monde de la recherche et d'acquérir l'expérience indispensable pour pouvoir continuer mon parcours de jeune chercheur de manière autonome.

J'adresse un grand merci à Thérèse Hardin pour m'avoir orienté très tôt vers l'informatique théorique et d'avoir cru en mes possibilités de pouvoir faire de la recherche dans ce domaine. Je suis heureux qu'elle m'ait fait l'honneur de présider mon jury. Je remercie Catherine Dubois et Yves Bertot, d'avoir accepté d'être mes rapporteurs et d'avoir amélioré la qualité de mon manuscrit de manière significative. Merci aussi à Christine Paulin-Mohring, Pierre Crégut et James McKinna pour m'avoir fait l'honneur d'examiner ce travail et d'avoir eu la gentillesse de faire partie de mon jury.

Je remercie profondément le projet LogiCal (ex-projet Coq), qui m'a accueilli durant ma thèse. En particulier, merci à Christine Paulin-Mohring, Gilles Dowek, Benjamin Werner, Micaela Mayero, Judicaël Courant, Alexandre Miquel, Jean Duprat, Jean Goubault-Larrecq, et tout spécialement aux *hackers* du groupe, Hugo Herbelin, Jean-Christophe Filliâtre, Bruno Barras, sans oublier les ex-membres Cristina Cornes, César Muñoz, Eduardo Gimenez, Patrick Loiseleur, Henri Laulhère, ainsi que notre assistante préférée, Nelly Maloisel.

Je ne pourrais sans doute jamais oublier l'ambiance de l'INRIA-Rocquencourt et tout particulièrement celle apportée par les projets *bordant* le projet LogiCal. Je remercie le projet Cristal, avec Michel Mauny, Pierre Weis, Xavier Leroy, Didier Rémy, François Pottier, Christian Rinderknecht, Émilie Sayag, François Pessaux et Robert Harley, le projet Moscova, avec Damien Doligez, Luc Maranget, Fabrice Le Fessant, Sébastien Ailleret, Sylvie Loubressac, et le projet Verso, avec Laurent Mignet, Fanny Wattez, Bruno Tessier, Danny Moreau. Merci aussi aux lapins, aux chats, aux pies, aux corbeaux et à l'équipe de football du site, de m'avoir si souvent distrait, ainsi qu'aux gardiens de m'avoir si souvent fait peur en débarquant dans mon bureau, passé 22h, pour voir si tout allait bien (mises à part des palpitations pouvant dépasser les 240, tout allait généralement bien).

Je tiens à remercier tout spécialement Daniel de Rauglaudre, qui a été un soutien de toute heure, et ce, même dans mes passades de noctambule pour raisons de codage ininterrompu (sachant que si vous voyez un codeur à 7h du matin, ce n'est pas parce qu'il s'est levé tôt, mais parce qu'il ne s'est pas couché). Daniel a toujours été d'une aide systématique me montrant le dernier hack à la mode, m'expliquant qu'il faut écrire `do {e1 ;e2}` et non `e1 ;e2`, ou m'apprenant qu'il a 219 229 liens de parentés avec Juan Carlos Ier (non, tant que cela ?). Je ne saurais pas non plus oublier les longues discussions (parfois nocturnes) sur le film qui vient de sortir, mais aussi sur des questions plus profondes comme la logique fait-elle

partie des mathématiques ou faut-il la laisser à l'informatique théorique (bonne question) ? Pour tout cela, merci Daniel.

Un merci tout particulier à Yves Bertot, Pierre Crégut, David Nowak et Micaela Mayero, qui ont *essuyé les plâtres* des premières versions de mon langage de tactiques et qui, en mettant à jour diverses bugs, m'ont permis de le stabiliser. Merci également à Claude Marché, Loïc Pottier et Renaud Rioboo, pour leur aide dans les discussions préliminaires à l'élaboration de la tactique `Field`, ainsi qu'à Micaela Mayero pour sa collaboration dans la réalisation de cette tactique.

Je souhaite aussi remercier ici Pascal Manoury pour son cours de Maîtrise, où, pour la première fois, j'ai pu découvrir, intrigué, ce symbole bizarre qu'est le λ et, avec une certaine stupeur, que les preuves pouvaient être codées par des λ-termes.

Enfin, un immense merci à ma famille. En particulier, merci à mes parents, qui m'ont permis de faire ces si longues études et qui m'ont soutenu quels que soient mes choix. Cette thèse représentent l'avènement de ces années et j'espère m'être montré à la hauteur. Merci à mon frère Benjamin, avec qui j'ai pu passer de bons moments de détente, tant nécessaires, en lui donnant notamment des corrections mémorables à *SoulCalibur* (c'était peut-être l'inverse). Merci à mes beaux-parents, avec qui j'entretiens, malgré les dogmes préétablis, les meilleures relations du monde, et qui ont toujours été présents, surtout dans la dernière ligne droite. Pour finir, aucun qualificatif, aucun superlatif n'est assez fort pour exprimer ma gratitude envers Micaela, qui, au-delà des apparences de petite chose fragile, possède probablement la plus grande force de caractère qui m'ait été donné de rencontrer. Si elle possédait déjà mon coeur, cette thèse lui a désormais donné mon admiration.

Göteborg, le 1er décembre 2001.

Table des matières

Introduction

De la difficulté d'exprimer les preuves mathématiques

Le langage mathématique est souvent considéré comme un langage universel, en ce sens qu'il tend à unifier l'expression de *faits*, qu'ils soient abstraits ou concrets. Par exemple, si l'on souhaite dire que, dans toute thèse de doctorat, il existe au moins une page intitulée "Introduction", on pourra l'exprimer, en théorie des ensembles, par :

$$\forall t \in \mathcal{T}. \exists p \in \mathcal{P}. page\_de\ (p, t) \Rightarrow titre\_intro(p)$$

où $\mathcal{T}$représente l'ensemble des thèses de doctorats, $\mathcal{P}$, l'ensemble des pages, $page\_de(p, t)$, une relation signifiant que p est une page de t, et $titre\_intro(p)$, un prédicat établissant que p a le titre "Introduction". Même si ce fait peut être formulé de différentes manières, suivant le choix des ensembles et des prédicats/relations, il est formellement exprimé, et peut être compris tel quel par tout mathématicien, connaissant la théorie des ensembles.

Toujours grâce au langage mathématique, les preuves peuvent aussi être exprimées de manière formelle. Par exemple, étant données deux propositions A et B, si l'on souhaite montrer que $A \wedge B \Rightarrow A \vee B$, on peut formuler la démonstration en déduction naturelle comme suit :

$$\dfrac{\dfrac{\dfrac{\dfrac{A \wedge B, A \vdash A}{A \wedge B, A \vdash A \vee B}\,\vee\text{-intro1}}{A \wedge B \vdash A \Rightarrow A \vee B}\Rightarrow\text{-intro} \qquad \dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A}\,\wedge\text{-élim1}}{A \wedge B \vdash A \vee B}\Rightarrow\text{-élim}}{\vdash A \wedge B \Rightarrow A \vee B}\Rightarrow\text{-intro}$$

Cette preuve est formelle dans la mesure où elle est formée de pas de déductions élémentaires d'un système de démonstrations (ici la déduction naturelle), constitué d'un nombre fini de règles. De même que pour l'expression des faits, un mathématicien est donc, là encore, capable de vérifier cette preuve, sans autre connaissance que la déduction naturelle.

Malgré cette forme d'expression, à caractère universel, il peut donc sembler étonnant qu'en ouvrant n'importe quel ouvrage mathématique, ni les propositions, ni les démonstrations ne soient formulées de cette manière. En effet, curieusement, on y retrouve *un langage naturel*, et tout lecteur néophyte est en mesure d'en appréhender au moins la syntaxe. Ainsi, la proposition formulée précédemment pourrait ressembler à :

> **Proposition 4 (Pages d'introduction)** *Pour tout thèse t, il existe au moins une page p de t, telle que le titre de p soit "Introduction".*

où l'ensemble des thèses, l'ensemble des pages, ainsi que les propriétés d'être la page d'une thèse et d'être le titre d'une page ont été préalablement définis.

Il en est, de même, pour la démonstration précédente, qui serait exprimée de la manière suivante :

> **Preuve** *Si on suppose que l'on a $A \wedge B$, alors on peut, en particulier, en déduire A, et si on a A, alors on a $A \vee B$.*
>
> $\square$

Il en résulte une preuve plus courte et plus compacte, que dans la version en déduction naturelle, et on comprend, dès lors, pourquoi les mathématiciens ne sont pas enclins à construire des preuves formelles. Il ne s'agit pas seulement d'une question de confort syntaxique, mais aussi d'être capable d'exprimer des preuves complexes, de manière concise. Cette méthode reste cependant problématique, puisque le langage naturel est ambigu. En effet, dans la proposition sur les pages d'introduction dans les thèses de doctorat, plusieurs interprétations, ou plus exactement plusieurs *formalisations* sont possibles. Par exemple, dans *telle que le titre de p soit "Introduction"*, on peut comprendre qu'il s'agit d'un prédicat $titre\_intro(p)$ ou bien d'une relation $titre(p, "Introduction")$. Même si on croit deviner un seul sens, on s'aperçoit qu'en réalité, il y a plusieurs sémantiques formelles, et que l'on perd le caractère unique dans l'interprétation du langage.

Pour rester dans un langage formel, tout en restant le plus concis possible, on pourrait imaginer éliminer systématiquement les coupures dans les preuves mathématiques. En déduction naturelle, une coupure désigne une preuve se terminant par l'application successive d'une règle d'introduction, puis d'élimination d'un connecteur logique. Par exemple, dans la preuve formelle précédente, il y a, en effet, une coupure pour le connecteur $\Rightarrow$. Pour éliminer cette coupure, il suffit de remplacer les prémisses de $\Rightarrow$ -élim par le séquent $A \wedge B, A \vdash A$ (branche gauche du $\Rightarrow$ -élim), duquel on enlève l'hypothèse A et qui sera prouvé par $A \wedge B \vdash A \wedge B$ (branche droite du $\Rightarrow$ -élim). On obtient ainsi la preuve sans coupures suivante :

$$\dfrac{\dfrac{\dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A}\wedge\text{-élim1}}{A \wedge B \vdash A \vee B}\vee\text{-intro1}}{\vdash A \wedge B \Rightarrow A \vee B}\Rightarrow\text{-intro}$$

Le théorème d'élimination des coupures assure que cette transformation peut toujours être effectuée et que le processus termine. On obtient ainsi une preuve de $A \wedge B \Rightarrow A \vee B$ moins volumineuse, mais ce n'est pas toujours le cas. Par exemple, si le séquent $\Gamma, A \vdash A$ apparaissait plusieurs fois dans la branche gauche du $\Rightarrow$ -élim, il faudrait transformer chaque occurence en $\Gamma \vdash A$ et le prouver avec $\Gamma, A \wedge B \vdash A \wedge B$. Il y aurait donc autant de *recopies* de preuves que d'occurences de $\Gamma, A \vdash A$. L'élimination des coupures n'est donc pas *forcément* une bonne méthode pour écrire des preuves concises, d'autant que, par ailleurs, la coupure joue un rôle crucial en mathématiques. En effet, soit la coupure suivante :

$$\dfrac{\dfrac{\dfrac{\pi}{\vdash A}}{\vdash \forall x.A}\forall\text{-intro}}{\vdash A[x\backslash t]}\forall\text{-élim}$$

Cette coupure peut être éliminée en démontrant $\vdash A[x\backslash t]$ avec π, où toutes les occurrences de x sont remplacées par t. Cependant, cette élimination n'est souhaitable, car il s'agit de montrer $\vdash A[x\backslash t]$, en utilisant le lemme $\vdash \forall x.A$, déjà prouvé. Éliminer la coupure revient à redémontrer ce lemme avec une instance particulière. Ce style de preuve n'est donc pas raisonnable, et les professeurs de mathématiques au lycée le savent bien, *il n'y a que les mauvais élèves qui éliminent les coupures.*

Les mathématiciens ne sont donc pas prêts de renoncer à utiliser le langage naturel pour exprimer les démonstrations, puisque les logiciens ne sont pas en mesure de leur apporter des solutions satisfaisantes. C'est pourquoi, dans les meilleurs ouvrages de mathématiques (voir [10], par exemple), la logique est généralement présentée, en prélude, comme une *imposante machinerie*, dont il est nécessaire de se séparer au plus vite, avant d'entamer les fondations des mathématiques. De ce fait, on serait presque tenté de penser que la logique et le reste des mathématiques n'ont aucune interaction.

Le point de vue de l'informaticien

Pour valider une preuve automatiquement (par une machine), le langage naturel utilisé par les mathématiciens ne convient pas. Comme on a pu le voir avec l'exemple concernant les pages d'introduction des thèses de doctorats, plusieurs interprétations sont possibles, alors qu'une machine (typiquement un programme), contrairement à un être humain, n'a pas la capacité de choisir une interprétation, et obéit, de ce fait, à une sémantique précise, c'est-à-dire, une sémantique formelle. L'avis de l'informaticien est donc beaucoup plus tranché, concernant la représentation des propositions et des démonstrations. Une machine ne peut traiter que des théories formelles, et il n'y a pas d'autres alternatives. Cela a pour conséquence qu'il n'est pas possible de vérifier, telles quelles, les démonstrations mathématiques. Il faut, au préalable, effectuer un travail d'adaptation, à savoir une *formalisation*. Cette étape préliminaire peut sembler, au premier abord, paradoxale dans la mesure où une preuve mathématique est censée représenter un objet formel, mais, en réalité, la pratique des mathématiciens est que la preuve n'est pas tant dans le texte décrivant les déductions, mais plutôt dans l'abstraction qui en est faite.

Cette adaptation des preuves mathématiques peut s'avérer très coûteuse en temps. En effet, il faut tout d'abord supprimer les éventuelles ambiguïtés dues au langage naturel. Ensuite, il faut traduire les preuves en séries de déductions formelles élémentaires, ce qui se révèle, en général, éminemment problématique, puisque, par souci d'être toujours plus concis, certaines parties de démonstrations sont laissées implicites. Ces *trous* dans les preuves ne sont pas là parce que le mathématicien n'a pas su les démontrer, mais parce qu'ils sont considérés comme étant *suffisamment triviaux* pour se passer d'*explications*. Inutile de dire à quel point cette notion est subjective, et une proposition suffisamment triviale pour l'un peut aisément ne pas l'être pour l'autre. Malgré cela, cette technique de preuves est très fréquemment, voire systématiquement utilisée par les mathématiciens, et comme elle ne peut être formellement décrite, les parties de preuves en question doivent être explicitées (souvent en un nombre non élémentaire de déductions). De par le passé, et même encore actuellement, des efforts ont été faits pour vérifier des parties des mathématiques dans des outils d'aide à la preuve. Par exemple, on peut citer le codage du *Grundlagen der Analysis* d'Edmund Landau [50] par L. S. Jutting, en AUTOMATH [21], ou la formalisation de l'analyse réelle par John Harrison [42], en HOL [35].

Pour éviter un codage coûteux des preuves mathématiques, une solution naturelle serait que les mathématiciens écrivent directement leurs démonstrations dans le langage (formel) compris par la machine. Cette idée semble raisonnable dans la mesure où il ne s'agit plus, pour les mathématiciens, de construire des preuves formelles volumineuses, mais de donner des instructions (potentiellement succinctes) à la machine pour construire ces preuves formelles. De plus, un bonus non négligeable s'offre alors aux mathématiciens, à savoir que leurs démonstrations sont vérifiées automatiquement et formellement par une *source extérieure* (la machine), ce qui est un gage supplémentaire de leur validité (ce n'est pas une certitude, car des erreurs d'implantation de la logique formelle considérée peuvent impliquer l'incohérence de cette logique). Toutefois, le langage utilisé par la machine doit posséder quelques *bonnes* propriétés, de manière à convaincre le mathématicien de *laisser tomber le papier et le crayon*. Notamment, les preuves doivent être faciles à écrire, tout en étant lisibles (par un être humain), de manière à laisser au mathématicien à la fois un certain pouvoir de concision qu'il possédait dans le langage naturel, et la possibilité de revenir sur une preuve, surtout si cette dernière est partielle.

L'industriel change la donne

Depuis plusieurs années maintenant, on ne fait plus des preuves formelles seulement pour vérifier des parties des mathématiques, mais aussi pour vérifier des propriétés de programmes. Il s'agit essentiellement de programmes critiques, qui ne sont pas *tolérants aux pannes*, et qui sont destinés, en général, à être intégrés à des systèmes embarqués. Les enjeux industriels sont importants, et le marché ne cesse de s'épanouir, demandant encore et toujours plus de méthodes formelles. À titre d'exemples, on peut citer Matra, qui utilise un outil basé sur la méthode B, France Télécom, qui a opté pour Coq, ou encore, outre-atlantique, la NASA, avec PVS.

Dans ces méthodes formelles, il y a globalement deux directions. La première est appelée *model-checking*, et consiste à écrire un programme, puis à vérifier ensuite certaines propriétés (critiques) de ce programme. Par exemple, si l'on écrit un programme qui gère une barrière de passage à niveau, on veut pouvoir vérifier que si un train passe, la barrière est bien baissée. La deuxième direction est uniquement basée sur la spécification formelle (du problème), de laquelle on *extrait* un programme exécutable. Cette extraction est plus ou moins facile selon la logique utilisée, et certaines spécifications peuvent s'avérer plus exécutables que d'autres. Ainsi, si on veut définir le prédicat $p(l_1, l_2)$, qui exprime que la liste l_1 est une permutation de la liste l_2, on peut soit le faire *axiomatiquement* en posant que $\forall l.p(l, l)$, etc, ce qui n'est pas exécutable, soit en définissant une fonction par récurrence sur l_1 et l_2, qui est directement exécutable.

Ainsi, l'informatique ne vient plus seulement *au secours* du mathématicien, mais aussi de l'industriel. Il y a donc un nouveau partenaire, qui serait presque privilégié, dans la mesure où, si un mathématicien peut finalement trouver assez anecdotique de vérifier formellement sa preuve, un industriel voit, dans cette démarche, la possibilité de minimiser les risques (voir le *bug* du Pentium III, qui a fait perdre des milliards de dollars à Intel). Il est donc important de se demander quelles peuvent être les attentes de l'industriel, en ce qui concerne le langage dont il se servira pour exprimer les preuves. De manière raisonnable, on peut concevoir que ces attentes ne sont pas différentes de celles du mathématicien, bien qu'elles ne soient pas motivées par les mêmes raisons. Par exemple, l'industriel voudra pouvoir écrire facilement ses preuves, ce qui est synonyme de gain de temps (et donc d'argent), mais aussi pouvoir les lire ultérieurement. Ce dernier point a une double justification. D'abord, il s'agit d'avoir une maintenance rapide de ses preuves, sachant qu'en période de prototypage, les spécifications changent beaucoup et vite. Ensuite, une validation n'a de sens que si elle peut être explicitée, ce qui signifie qu'une preuve formelle doit pouvoir être informellement appréhendée. Ce *va-et-vient* entre spécifications formelle et informelle est courante dans ce type de projets, et correspond à des schémas généraux de génie logiciel (cycle en V, cycle en spirale, ...).

Des preuves automatiques ?

Depuis les années 30, on sait que le rêve d'Hilbert de pouvoir créer une machine qui, étant donnée une proposition P, répond "Oui", si P est valide, ou "Non", dans le cas contraire, était vain[1]. En effet, Church [15] montre, en 1936, que la logique du premier ordre est indécidable en général, et pire encore, Gödel [32] avait montré, en 1931, que dans tout système formel cohérent contenant l'arithmétique, il existait une proposition valide non démontrable (la cohérence du système, par exemple). Ainsi, une machine peut aider à réaliser des preuves formelles, mais ne peut pas, de manière générale, les construire automatiquement.

Malgré cela, l'automatisation reste un point crucial, sans lequel il ne saurait être question de faire des preuves sur machine. En effet, comme nous l'avons dit précédemment, le mathématicien utilise beaucoup de *raccourcis*, concernant des propositions qu'il juge triviales, et il semble légitime, de son point de vue, qu'il puisse faire de même, le plus souvent possible, sur machine, sans avoir à détailler les preuves correspondantes. L'industriel a aussi ce même sentiment, qui est d'autant plus exacerbé que, plus les preuves sont traitées automatiquement, plus cela implique un gain de temps significatif.

Ainsi, pour faire de la démonstration automatique, on peut, dans un premier temps, utiliser des résultats connus, qui montrent que certains fragments de la logique, comme la logique propositionnelle, ou certaines théories, comme la géométrie, sont décidables. On peut alors coder les algorithmes correspondants (pas toujours efficaces). Ensuite, pour le reste, on ne peut que construire des méthodes basées sur des heuristiques, qui sont forcément incomplètes et qui capturent des classes de propositions différentes. Suivant les heuristiques, les systèmes peuvent alors présenter des automatisations très différentes, si bien qu'une proposition pouvant être démontrée automatiquement dans un système donné, peut ne pas l'être dans l'autre, et *vice-versa*. Pour éviter à l'utilisateur de *jongler* entre plusieurs systèmes, suivant ce qu'il cherche à démontrer (ce qui l'obligerait à apprendre plusieurs langages et à formuler plusieurs fois sa spécification), il est important de lui donner la possibilité d'enrichir l'automatisation du système, au moyen d'un langage spécifique, dédié à l'écriture de nouvelles automatisations.

De ce langage d'automatisation, on peut raisonnablement attendre qu'il soit d'abord suffisamment complet, pour ne pas limiter l'écriture de nouvelles automatisations. Ensuite, il doit être de haut-niveau (par rapport à l'implantation), de manière à ce que le codage d'automatisations ne nécessite pas trop de détails concernant l'implantation du système. Enfin, il faut qu'il soit portable, afin que les nouvelles automatisations soient robustes vis-à-vis des évolutions internes du système (au niveau de l'implantation).

L'objectif de ce travail

Nous nous proposons, dans ce qui suit, d'étudier à la fois les langages de preuves, c'est-à-dire les langages permettant d'exprimer formellement les preuves pour qu'elles soient vérifiables par une machine, et les langages d'automatisation, c'est-à-dire les langages qui permettent d'étendre l'automatisation d'un système d'aide à la preuve. Il y aura donc deux parties distinctes. Dans la première partie, nous ferons d'abord une classification des différents types de langages de preuves (chapitre 1), ainsi qu'un tour d'horizon de plusieurs systèmes représentatifs de ces langages (chapitres 2, 3, et 4), avant d'identifier les champs d'application les plus adéquats de ces langages et de proposer une solution alternative, qui

[1] Il s'agit, en réalité, du 33ème problème d'Hilbert, énoncé par Hilbert en 1900, à la conférence internationale des mathématiques de Paris, et appartenant à une proposition d'agenda qui comprenait une liste des 33 problèmes les plus urgents à résoudre pour le siècle à venir.

tend à unifier ces langages (chapitre 5). Dans la deuxième partie, nous verrons d'abord des exemples de langages d'automatisation, puis nous remarquerons la nécessité de construire un nouveau langage possédant les critères que nous avons vus précédemment. Nous construirons alors ce nouveau langage (chapitre 6), et nous donnerons ensuite des exemples d'utilisation non triviaux (chapitres 7 et 8). Enfin, nous présenterons deux outils dédiés à ce nouveau langage (chapitre 9).

Première partie

Langages de preuves

Chapitre 1

Classification

Dans un premier temps, nous nous proposons de faire un tour d'horizon des différents types de langages de preuves existants, afin de les comparer et de justifier, si besoin est, la nécessité d'une solution alternative. De manière annexe, cela permettra également de se familiariser, même si c'est de manière sommaire, avec une petite sélection d'outils d'aide à la preuve.

1.1 Les différents langages

De manière générale, on peut caractériser les langages de preuves en trois catégories distinctes :

- Les langages procéduraux
- Les langages déclaratifs
- Les langages de termes

Les langages procéduraux sont dirigés par la proposition que l'on cherche à démontrer et donnent des ordres (instructions) à une machine de preuve, ordres qui correspondent à des pas plus ou moins élémentaires de la logique formelle sur laquelle repose le prouveur. Les langages déclaratifs sont basés sur une méthode qui consiste à poser (déclarer) des lemmes intermédiaires, jusqu'à ce que le système soit automatiquement capable de les combiner afin d'obtenir la preuve globale. Les langages de termes sont utilisés dans les prouveurs utilisant l'isomorphisme de Curry-Howard (intuitionniste). La preuve est un terme dont le type est la proposition que l'on cherche à montrer. La preuve peut être incomplète, ce qui se traduit par des *trous* dans le terme, que l'on peut instantier ultérieurement (raffinement).

Historiquement, les premiers langages de preuves furent plutôt déclaratifs avec Mizar (Andrzej Trybulec, 1978, [88]) et Nqthm (R. S. Boyer et J. S. Moore, 1979, [12, 13]) même si le premier vérificateur de preuves fut AUTOMATH (N. G. De Bruijn, 1970, [21]) avec un langage de termes[1]. Ceci peut s'expliquer dans la mesure où l'on cherchait, à l'époque, plus à utiliser la machine pour faire des preuves mathématiques et si possible avec un style assez proche du langage naturel. Par ailleurs, on avait plutôt tendance à construire sa preuve au préalable ce qui est un peu opposé à la notion de preuve interactive proposée dans les systèmes procéduraux. Il a fallu attendre les années 80 pour voir apparaître les

[1] En effet, le langage d'AUTOMATH est un langage de termes, mais pas au sens de Curry-Howard. Il s'agit d'un langage complètement isomorphe à la logique, où chaque symbole de fonction correspond à une règle donnée, dont le nombre de prémisses fixe l'arité de la fonction. Une preuve réalisée dans la logique formelle peut ainsi être directement traduite dans le langage d'AUTOMATH.

premiers langages de termes, issus pour la plupart du prouveur procédural Edinburgh LCF (M. J. C. Gordon, R. Milner et C. P. Wadsworth, 1979, [36]), avec les premières versions de Coq (Thierry Coquand et Gérard Huet, 1984, [19]) qui, très vite, laissèrent la place à un langage plutôt procédural même si l'utilisateur a toujours la possibilité de donner des termes à raffiner[2]. Il en fut de même pour d'autres systèmes comme Lego (Randy Pollack, 1988, [72]). Actuellement, seul Alfa, successeur d'ALF (Thierry Coquand et Bengt Nordström, 1991, [66]), utilise une manipulation directe des objets preuves en tant que termes. Les systèmes plus modernes, comme HOL (M. J. C. Gordon, 1988, [35]) ou PVS (Sam Owre, Natarajan Shankar et John Rushby, 1992, [67]) par exemple, sont majoritairement procéduraux dans la mesure où l'on utilise plus seulement les prouveurs pour faire des mathématiques, mais aussi dans le domaine de l'informatique pour faire, entre autres, du *model-checking*. Cependant, des études récentes montrent un net regain des langages déclaratifs, comme avec le Mizar-mode pour HOL (John Harrison, 1996, [41]), Declare (Don Syme, 1998, [82]) ou les travaux de Vincent Zammit (1998, [93]).

De cette brève classification, on peut, d'ores et déjà déduire, qu'il y a eu peu d'efforts dans le domaine des langages de preuves. En effet, mis-à-part les travaux récents de John Harrison, Don Syme et Vincent Zammit, concernant l'intérêt des langages déclaratifs, les priorités se sont plutôt tournées vers la métathéorie ou l'automatisation, tandis que la manière d'exprimer les preuves n'a pas fait l'objet d'un réel débat.

1.2 Caractérisation, chronologie et hiérarchie

Pour avoir une idée du paysage des principaux éditeurs de preuves existants, on peut se référer à la figure 1.1. Sur cette figure, sont représentés les trois mondes de langages de preuves cités précédemment avec, comme nous pouvons le remarquer, une intersection non vide entre les langages procéduraux et les langages de termes (Coq et Lego). Les flèches reliant certains prouveurs ont une sémantique d'héritage. Une flèche allant du prouveur A vers le prouveur B signifie que B utilise les concepts de A, voire est complètement basé sur A. Bien évidemment, cette liste est loin d'être exhaustive et une liste beaucoup plus complète, avec les références correspondantes et maintenue à jour par Freek Wiedijk, peut être consultée à l'adresse suivante :

http ://www.cs.kun.nl/~freek/digimath/index.html

Dans la suite, afin de comparer les trois styles de preuves mais aussi de comparer les prouveurs ayant le même style de langage, nous considérerons une bonne partie des outils de preuves de la figure 1.1.

1.3 Comparaison

Avant d'apporter des éléments de réponse au problème de savoir ce que peut être un *bon* langage de preuves, il est nécessaire de faire un petit comparatif au moyen d'une sélection relativement représentative d'outils d'aide à la preuve. En fait, nous nous proposons de réaliser deux types de comparaisons : une comparaison longitudinale entre prouveurs utilisant le même style de preuve et une comparaison transversale entre prouveurs utilisant des styles de preuve distincts.

[2] C'est effectivement possible, mais le système n'est pas très flexible et c'est utilisé essentiellement de manière interne.

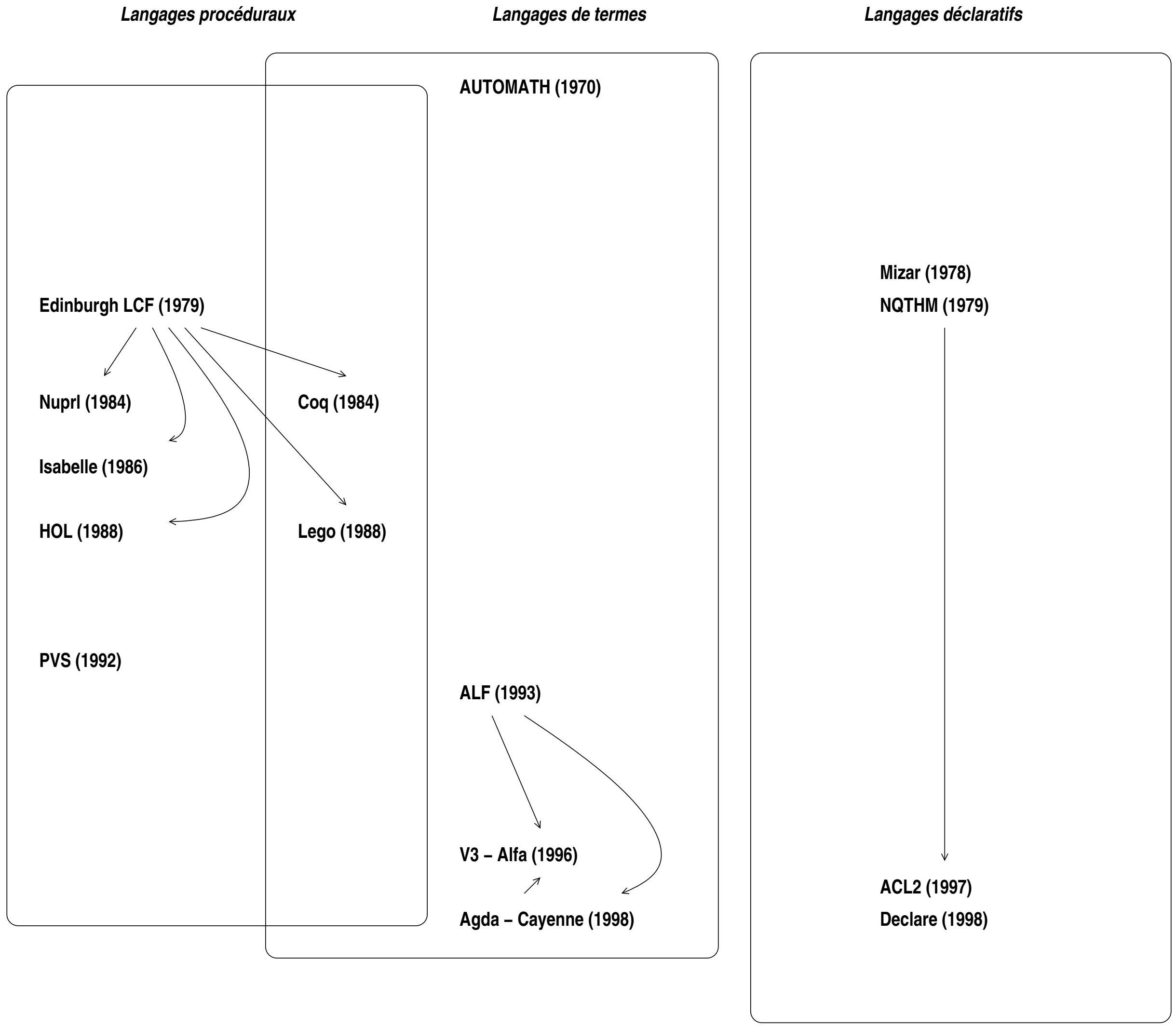


FIG. 1.1 – Classification des principaux outils d'aide à la preuve.

Pour juger les trois styles de preuves que nous nous proposons d'étudier, nous allons utiliser certains critères, dont la plupart sont inspirés de [40], et qui sont : la lisibilité, la facilité d'écriture, l'orientation de la preuve (*backward/forward*), la maintenance (système et des spécifications), le mode de preuve (*toplevel/batch*), les performances et éventuellement la difficulté d'implantation.

Afin d'effectuer une comparaison *cohérente*, nous allons utiliser un exemple récurrent. L'exemple doit être relativement simple de manière à ce que la preuve ne soit pas excessivement longue, selon l'éditeur de preuves utilisé, et surtout pour que nous puissions dégager facilement la structure de la preuve, que nous connaissons informellement.

1.4 Exemple

1.4.1 Définition

Comme exemple récurrent, on se propose de montrer la décidabilité de l'égalité sur les entiers naturels $\mathbb{N}$. Cette proposition peut s'exprimer de la manière suivante :

$$\forall n, m \in \mathbb{N}. n = m \vee \neg(n = m)$$

Nous avons appelé cette proposition décidabilité de l'égalité dans la mesure où nous donnerons une preuve intuitionniste dans tous les cas. Ainsi, on peut toujours *réaliser*, du moins théoriquement, la preuve de ce théorème vers un programme qui étend donné deux entiers naturels répond oui si les deux entiers sont égaux et non sinon. Cette notion est donc indépendante du choix de la logique de l'outil d'aide à la preuve ou de la sorte choisie pour placer le lemme en question, qui peuvent parfaitement être classiques.

1.4.2 Preuve informelle

Mathématiquement, la proposition précédente peut se montrer de la manière suivante :

1. On raisonne par récurrence sur n.
   Le cas de base nous donne $0 = m \vee \neg(0 = m)$ à montrer :
   (a) On raisonne par cas sur m.
       On doit d'abord montrer que $0 = 0 \vee \neg(0 = 0)$, ce qui est trivial car on sait que $0 = 0$.
   (b) Ensuite, on doit montrer que $0 = m + 1 \vee \neg(0 = m + 1)$, ce qui est trivial car on sait que $\neg(0 = m + 1)$.
2. Pour le cas récurrent, on suppose que l'on a $n = m \vee \neg(n = m)$ (H) et on doit montrer que $n + 1 = m \vee \neg(n + 1 = m)$:
   (a) On raisonne par cas sur m.
       On doit d'abord montrer que $n+1 = 0 \vee \neg(n+1 = 0)$ à montrer, ce qui est trivial car on sait que $\neg(n + 1 = 0)$.
   (b) Ensuite, on doit montrer que $n + 1 = m + 1 \vee \neg(n + 1 = m + 1)$. On raisonne alors par cas selon l'hypothèse de récurrence H :
       i. Soit $n = m$: on peut conclure car on peut en déduire que $n + 1 = m + 1$.
       ii. Soit $\neg(n = m)$: on peut conclure car on peut en déduire que $\neg(n+1 = m+1)$.

Pour avoir un point de vue un peu plus global sur la preuve de ce lemme, la figure 1.2 donne un schéma permettant de mieux concevoir la structure de la preuve.

Comme convenu, la preuve est plutôt courte et fait appel à différents aspects de techniques de preuves. Parmi celles-ci, on a du raisonnement par récurrence, par cas (pas seulement sur les entiers naturels), et derrière les *trivial*, se cachent l'application d'autres lemmes élémentaires, éventuellement de la réécriture ou simplement du calcul. En effet, il s'agira de formaliser les étapes de cette preuve en explicitant toutes les déductions, afin de créer un objet qui puisse être vérifié par une machine[3]. En particulier, les preuves dites triviales doivent être détaillées, afin que la machine puisse les accepter et que, par la même occasion, la comparaison sur les aspects du langage de preuves ne soit pas faussée par l'automatisation des outils d'aide à la preuve.

[3] À partir d'ici, le mot *preuve* commence à être surchargé dans la mesure où il peut être soit une suite de déductions logiques, soit une suite d'instructions présentées à une machine.

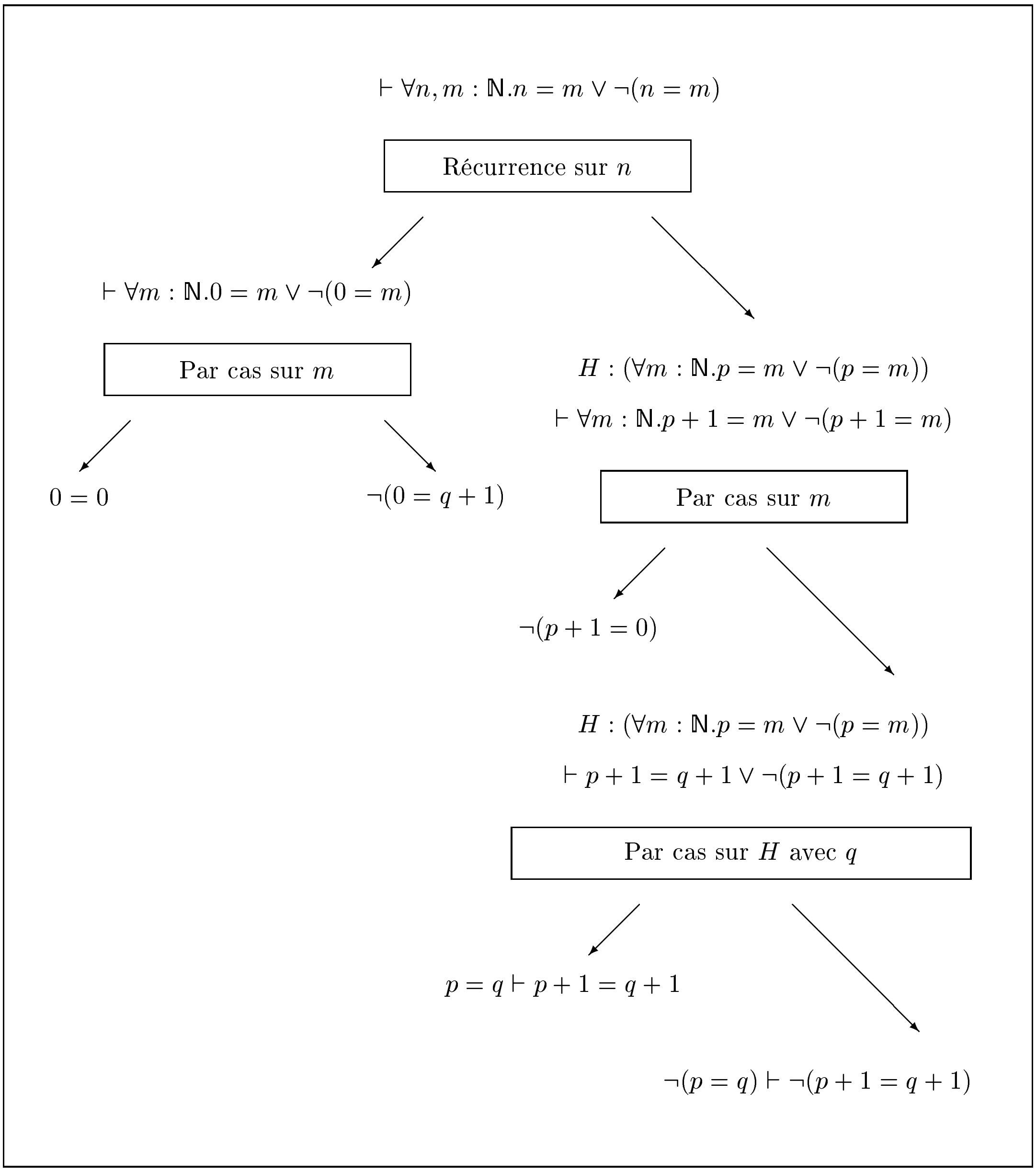


FIG. 1.2 – Preuve de décidabilité de l'égalité sur $\mathbb{N}$

Chapitre 2

Langages procéduraux

Dans ce chapitre, nous allons étudier quelques outils d'aide à la preuve (parmi les principaux) dont le langage de preuves est procédural. L'objectif est en réalité double : principalement, on souhaite établir quels sont les atouts et les points faibles d'un tel style de langage et, de manière plutôt annexe, dégager, s'il y en a vraiment, les nuances qui peuvent exister entre langages procéduraux.

Pour réaliser notre banc d'essais, nous utiliserons l'exemple choisi au chapitre 1. Parmi les prouveurs procéduraux de la figure 1.1, nous avons choisi d'utiliser PVS, HOL et Coq. Nous n'utiliserons pas Edinburgh LCF, qui, à notre connaissance, n'est plus utilisé, mais il est largement représenté par ses descendants directs (voir la figure 1.1), dont HOL et Coq, que nous allons présenter. Pour Isabelle [71, 70], le prouveur est générique et il s'instantie, en particulier, avec la logique de HOL, que nous testerons. Enfin, Lego, ainsi que Nuprl [17, 43, 44], sont très proches de Coq, et n'apporteront rien d'essentiellement nouveau, dans le cas de notre exemple, par rapport au codage de la preuve en Coq.

2.1 PVS

2.1.1 Contexte

PVS est un outil d'aide à la preuve datant de 1992, et se définissant comme un système de vérification de prototypes (PVS pour Prototype Verification System). Sam Owre, Natarajan Shankar et John Rushby ([67]) sont à l'origine de ce prouveur dont l'implantation est réalisée en Common Lisp et utilise (X)Emacs comme interface.

PVS utilise une logique classique basée sur une variante de la théorie des types simples à l'ordre supérieur. En fait, les types sont un peu plus que simples dans la mesure où, outre le type fonction, le type produit cartésien et le type enregistrement, il y a aussi de la récurrence structurelle, des sous-types explicites et des types dépendants. Les formules (nom donné aux propositions) sont de type booléen (utilisation de la logique classique). Le sous-typage, avec une interprétation purement ensembliste, rend la vérification de type indécidable et son utilisation peut générer automatiquement des conditions de correction de typage, appelées TCC's en PVS pour Type-Correctness Conditions et que l'utilisateur devra montrer (manuellement). Les TCC's sont de deux sortes : montrer que certains types sont non vides (pour éviter que des constantes soient déclarées sur des types inhabités) et montrer que des opérations définies sur un sous-type soient compatibles avec le supertype.

PVS est très automatisé[1] par rapport à des prouveurs comme HOL ou Coq, mais moins que d'autres outils comme Otter ou ACL2 [47, 48]. Ceci est dû, entre autres, à l'utilisation de la logique classique (avec un séquent classique[2]), mais aussi et surtout, grâce à des procédures de décision ciblées utilisant des heuristiques qui fonctionnent dans la plupart des cas.

2.1.2 Script de la preuve

Les spécifications en PVS sont une suite de *théories*. Une théorie est une signature avec des déclarations de types et de constantes ainsi que des axiomes, définitions et théorèmes[3]. Les théories peuvent être paramétrées par des types et des valeurs. Dans notre cas, le type des entiers naturels étant prédéfini en PVS, notre théorie est très courte :

```
eq_nat_dec: THEORY
BEGIN
  n,m: VAR nat
  eq_nat_dec: THEOREM (n=m) OR (NOT (n=m))
END eq_nat_dec
```

La déclaration de `n` et `m` en variable de type `nat` permet de réaliser une quantification universelle implicite[4] sur tous les axiomes et théorèmes les utilisant.

Une fois la théorie saisie, il suffit de se placer sur le `THEOREM eq_nat_dec` et de déclencher le prouveur interactif (M-x prove). En PVS, il n'existe pas de mode de preuve *batch*. En batch, on ne peut que rejouer les preuves (préalablement effectuées en interactif) d'une théorie. La preuve de notre théorème, rejouée en interactif, ressemble donc à un dialogue avec PVS qui peut être consulté en annexe A, section A.1.1. Les commandes, appelées *règles* en PVS, nécessaires pour compléter la preuve sont les suivantes (les numéros en début de ligne ne font pas partie des règles et nous permettront de commenter le script) :

```
1   (induct "m")
2   (induct "n")
3   (flatten-disjunct)
4   (skolem!) (flatten-disjunct)
5   (skolem!) (flatten) (induct "n")
6   (flatten-disjunct)
7   (skolem!) (case "j!2 = j!1")
8   (flatten-disjunct)
9   (flatten-disjunct)
```

En se référant à la figure 1.2 du chapitre 1, on peut tenter de retrouver la structure de la preuve informelle. À la ligne 1, on fait la récurrence sur le premier argument de

[1] Nous aurons l'occasion de le voir par la suite, mais il est vrai que l'automatisation de PVS est très puissante, et il est parfois difficile de la contrôler pour réaliser des pas plus élémentaires de preuves (à des fins pédagogiques, par exemple).

[2] De manière générale, les preuves en logique classique sont plus *faciles* qu'en logique intuitionniste. Par exemple, si l'on veut montrer qu'il existe deux nombres irrationnels a et b, tels que, a^b est un nombre rationnel, la preuve en logique classique fait quelques lignes (on utilise le fait que $\sqrt{2}$ à la puissance $\sqrt{2}$ est rationnel ou non), alors qu'en logique intuitionniste, la proposition est également vraie, mais la preuve nécessite trois pages, faisant appel à des lemmes de théorie des nombres non triviaux.

[3] C'est une vision très algébrique des spécifications.

[4] La quantification universelle explicite se fait avec `FORALL`.

type `nat` (ici `m`[5]). En ligne 2, on est dans le cas `m=0`, et on fait la récurrence sur `n`[6]. À la ligne 3, on doit alors prouver que $0 = 0 \vee \neg(0 = 0)$, ce qui est fait directement par `(flatten-disjunct)`. `(flatten-disjunct)` déstructure tous les connecteurs $\vee$ jusqu'à une certaine profondeur. Ici, nous n'avons pas précisé de profondeur et `(flatten-disjunct)` est donc complètement équivalent à la règle `(flatten)` du point de vue sémantique. Toutefois, `(flatten-disjunct)` semble plus informatif sur ce qu'il fait et nous l'avons préféré ici à `(flatten)` car il ne nous a pas été possible ici de rendre plus élémentaire la preuve de $0 = 0 \vee \neg(0 = 0)$[7]. En ligne 4, on traite le cas récurrent où l'on doit montrer que $\forall j : \mathbb{N}. j = 0 \vee \neg(j = 0) \rightarrow (j + 1 = 0) \vee \neg(j + 1 = 0)$. On introduit la variable de récurrence (renommée ici de `n` vers `j` par `induct`, voir l'annexe A) par `(skolem!)` qui skolémise en générant ses propres symboles[8] (ici `j!1`, voir l'annexe A). Ensuite, on introduit l'hypothèse de récurrence (inutile ici) et choisissons le bon côté (droit ici) de la disjonction en un seul coup avec `(flatten-disjunct)`. En ligne 5, c'est le cas qui correspond à la partie récurrente de la première récurrence sur `m` et on doit montrer $\forall j : \mathbb{N}.(\forall n : \mathbb{N}. n = j \vee \neg(n = j)) \rightarrow (\forall n : \mathbb{N}. n = j + 1 \vee \neg(n = j + 1))$. On introduit `j` par `(skolem!)` puis l'hypothèse de récurrence (qui nous servira ultérieurement) par `(flatten)`[9]. On peut alors faire la dernière récurrence sur `n` (qui est en fait un raisonnement par cas) par `(induct "n")`. En ligne 6, on traite le cas de base toujours directement avec `(flatten-disjunct)`. Enfin, en ligne 7, dans le cas récurrent, on introduit la variable de récurrence (j) avec `(skolem!)` puis on raisonne par cas sur l'égalité des variables de récurrence avec `(case "j!2 = j!1")`. Ce `case` devrait nous générer une TCC de la forme $j!2 = j!1 \vee \neg(j!2 = j!1)$ qui peut être montrée par l'hypothèse de la première récurrence. Toutefois, ce n'est pas le cas car cette TCC est une instance du tiers exclu qui fait partie de la logique de PVS. Cette TCC est donc directement montrée grâce au tiers exclu[10]. En ligne 8, on traite le cas $j!2 = j!1$, en introduisant l'hypothèse de la troisième récurrence (inutile) et en choisissant la bonne clause (gauche) avec `(flatten-disjunct)`. De même, en ligne 9, on traite le cas de la négation avec toujours l'introduction de l'hypothèse de récurrence, puis le choix du côté (droit) avec `(flatten-disjunct)`.

La preuve que nous avons donnée est un script construit étape par étape dans la boucle interactive de PVS et ce n'est donc qu'une liste de règles. Il est possible de ne donner qu'une seule commande à PVS de manière à ce que le théorème soit prouvé d'un seul coup. On utilise, pour cela, des combinateurs de règles, appelés stratégies en PVS, qui permettent d'assembler, de manière très précise mais limitée, des règles primitives du système mais aussi des règles plus complexes (construites à l'aide de stratégies) afin d'accroître la concision des

[5]Lorsque le théorème a été saisi, `n` a bien été saisi *avant* `m` mais PVS réarrange les variables selon un ordre lexicographique.

[6]En PVS, il n'y a pas de règle particulière pour raisonner par cas. On fait donc une récurrence dans laquelle on n'utilise pas l'hypothèse de récurrence.

[7]On voit ici qu'il est difficile en PVS d'affiner la granularité de la preuve et donc de la structurer précisément. C'est un revers d'une très forte automatisation qui peut être assez décevant si on est intéressé par l'arbre de preuve. Par ailleurs, cela demande à l'utilisateur d'accorder une certaine confiance au prouveur qu'il n'est peut-être pas prêt à accorder.

[8]On peut être plus précis en donnant des noms aux variables skolémisées avec `(skolem)`. Ici, par exemple, on pourrait utiliser `(skolem 1 "q")` pour utiliser la variable `q` au lieu de `j!1`. Le `1` indique que l'on applique la skolémisation à la première formule à droite du séquent (PVS utilisant un séquent classique, il peut y avoir plusieurs formules à droite d'un séquent).

[9]Ici, le prouveur ne peut qu'introduire le produit. C'est pourquoi `(flatten)` est complètement suffisant par rapport à `(flatten-disjunct)`.

[10]Ce n'était pas réellement voulu au départ mais, en PVS, il n'y a pas de moyen de raisonner directement par cas sur une hypothèse. On a donc fait un raisonnement par cas sur une formule que l'on pensait pouvoir résoudre grâce à l'hypothèse de récurrence mais ce n'est pas le cas. La trop forte automatisation nous empêche non seulement de détailler correctement les preuves mais peut, quelquefois à notre insu, transformer une preuve intuitionniste en une preuve classique.

scripts de preuves et, par la même occasion, de leur donner un peu de structure. En utilisant les stratégies de PVS, le script de preuve précédent peut alors être codé en une seule règle, pour laquelle on peut consulter la réponse du toplevel[11] de PVS en annexe A, section A.1.2. La règle en question est la suivante :

```
(spread! (induct "m")
  ((spread! (induct "n")
    ((flatten-disjunct)
     (then (skolem!) (flatten-disjunct))))
   (spread! (then (skolem!) (flatten) (induct "n"))
    ((flatten-disjunct)
     (spread! (then (skolem!) (case "j!2 = j!1"))
       ((flatten-disjunct)
        (flatten-disjunct)))))))
```

où `(spread! rule lrules)` applique la règle `rule` au but courant puis la i^ème^ règle de la liste de règles `lrules` au i^ème^ sous-but généré par l'application de `rule`. `spread!` lève une erreur s'il n'y a pas exactement le même nombre de règles dans `lrules` que de sous-buts générés. La stratégie (`then` $\texttt{rule}_1$ $\texttt{rule}_2$ `...` $\texttt{rule}_n$) applique la règle $\texttt{rule}_1$ au but courant puis la règle $\texttt{rule}_2$ aux sous-buts générés par cette application et ainsi de suite jusqu'à la règle $\texttt{rule}_n$).

2.1.3 Observations

Une première remarque sur les preuves, que l'on a données à la machine de preuves de PVS, concerne la lisibilité. On a effectivement, dans ces scripts, bien du mal à repérer l'endroit où l'on se situe exactement dans la preuve informelle schématisée dans la figure 1.2. Au début, cela semble plutôt simple, puis, finalement, on perd très vite le fil conducteur de la preuve. Dans la première version de la preuve mécanisée, sans les numéros de lignes censés servir de points de repères, il est difficile de reconnaître la preuve voire impossible de la construire pour une personne ne connaissant ni la preuve ni la sémantique exacte des règles de PVS. En effet, cette preuve manque clairement d'annotations (par des formules) nous indiquant ce qui a été généré et ce que l'on est en train de prouver. Par exemple, un élément assez déstabilisant pour le lecteur de la preuve est l'apparition de nouvelles variables qui sont complètement libres dans le script mais bien évidemment liées dans l'arbre de preuve. Dans notre script, si on laisse de côté le cas de `(induct "m")` et `(induct "n")` où l'on peut considérer que ces variables sont connues de l'utilisateur (ce sont les variables généralisées du théorème saisi), on est typiquement dans ce genre de situation avec `(case "j!2 = j!1")` où l'on ne connaît ni `j!2` ni `j!1`. Ces variables ont été introduites par la skolémisation que l'on qualifiera de *rapide* (règle `(skolem!)`), très pratique, mais qui laisse peu de traces. Dans ce cas, PVS nous offre la possibilité d'être plus informatif avec `skolem`, qui peut donner le nom de la variable skolémisée, mais, sachant qu'en PVS, on fait ses preuves exclusivement en mode interactif, on n'est pas particulièrement enclin à être très précis mais plutôt concis avec des règles les plus succinctes possibles.

Dans la deuxième version de la preuve, les stratégies donnent plus de structure à la preuve. On voit un peu plus clairement à quel niveau s'applique les règles et combien de sous-buts sont générés. Toutefois, l'indentation y est pour beaucoup et on ne sait toujours pas quels sont les sous-buts générés. Par ailleurs, la construction de preuves à base de

[11] De manière générale, on désigne par toplevel d'un système, une boucle permettant un usage interactif du système et constitué d'une succession de phases lecture-évaluation-affichage.

stratégies se révèle utile sur des petits exemples (comme ici) mais sur des exemples plus réalistes, l'indentation devient trop importante et, de ce fait, non significative. Il faut alors interrompre le fil directeur construit par les stratégies afin de revenir à l'indentation initiale. La preuve est donc une liste de blocs de stratégies que l'on ne sait pas regrouper de manière à faire une preuve cohérente.

Une remarque directement liée à la lisibilité est qu'il est impossible de faire des preuves sans le vérificateur de preuves. Il existe ainsi une certaine dépendance de l'utilisateur (vis-à-vis de PVS), qui ne peut pas construire de preuves formelles en dehors du système. Par ailleurs, le format des preuves, une fois interprétées, n'est pas réellement compréhensibles en dehors de l'interpréteur, et donc vérifiables en dehors de celui-ci par un autre vérificateur. Il y a donc ici une deuxième marque de dépendance, où l'utilisateur doit implicitement faire confiance au vérificateur, lorqu'une preuve a été complétée.

La contrepartie du défaut de lisibilité de PVS est que les preuves sont plutôt faciles à écrire et ce, même si l'on fait abstraction de l'automatisation puissante dont PVS est doté. Bien sûr, cela n'est le cas que si l'utilisateur travaille en interactif. En mode batch, il n'est clairement pas permis d'imaginer pouvoir construire une preuve dépassant les dix règles[12]. Dans la boucle de l'interpréteur, l'utilisateur peut être très succinct dans ses commandes en omettant les symboles de skolémisation, les propositions qu'il manipule voire les témoins d'existentielles. D'un point de vue pratique, il s'agit d'une méthode de construction de preuves certes très agréable mais qui se compare aisément à la production de code par certains programmeurs sans souci de documentation. On arrive rapidement à une situation où le code (les preuves) n'est plus maintenable faute de spécification claire.

À propos de la maintenance des preuves PVS, on peut dire que les scripts ne sont pas très robustes aux changements du système et plus particulièrement à l'évolution des règles. Parfois, les extensions sont conservatives et il n'y a donc pas de problèmes à porter les bibliothèques de preuves mais quelquefois, on décide de faire des choix différents qui ne peuvent pas être conservatifs provoquant des erreurs d'interprétation. Dans ce dernier cas, le fait de n'avoir pas été très précis dans les scripts est une grave lacune qui impose beaucoup de temps passé à corriger les preuves. Par ailleurs, les extensions non conservatives peuvent s'avérer d'autant plus problématiques que PVS est très automatisé, ce qui nécessite, dès lors, un bon contrôle des règles puissantes (puisque fortement utilisées). Par contre, de tels scripts sont moins sensibles aux changements de spécifications. En effet, le renommage de définitions, la permutation d'arguments, l'ajout d'arguments et autres modifications des spécifications ont généralement que très peu d'influence sur les preuves. Ceci est dû au fait que l'utilisateur est moins sollicité, dans ses preuves, pour donner de l'information et donc expliciter certaines formules, si bien que plus l'automatisation est forte, plus les preuves ont de chances significatives d'être vérifiées sans erreur.

Comme on l'a vu dans le script (même si ce n'était pas l'objet ici), PVS est fortement automatisé. Cela est directement relié au problème de lisibilité évoqué précédemment. Les scripts ont d'autant moins de structure que PVS a réussi à résoudre rapidement. On arrive même à des situations où le script n'a presque plus de sémantique, car réduit à l'utilisation d'une ou deux règles, qui sont des sortes de *rouleaux-compresseurs* capturant une multitude de lemmes. Ces preuves sont alors peu portables et ne peuvent être vérifiées que par l'interprète de PVS, d'où, encore fois, la nécessité d'une confiance implicite de la part de l'utilisateur[13]. Un autre problème soulevé par l'automatisation de PVS est que l'on n'a pas

[12]La sémantique peu claire de l'automatisation y est pour beaucoup dans la mesure où l'on ne sait jamais exactement si la règle aura réussi à résoudre tel ou tel but. Cela crée très vite des scripts asynchrones avec l'état de preuve de l'interpréteur et donc des erreurs difficiles à localiser.

[13]Ce n'est pas particulièrement problématique pour le mathématicien, qui a l'habitude d'avoir confiance dans les preuves de propositions qu'il juge triviales, mais cela peut le devenir dans un contexte plus *industriel*,

de moyens pour détailler correctement les pas élémentaires utilisés par les stratégies. Certaines règles sont plus informatives que d'autres mais on ne peut pas contrôler précisément le chemin emprunté pour construire la preuve. Des commandes détaillant les preuves élaborées par les stratégies seraient tout à fait souhaitables bien que les preuves automatiques sont souvent loin d'être optimales et peuvent donner des preuves dont la taille peut clairement décourager l'utilisateur voulant regarder d'un peu plus près. Enfin, un dernier point plutôt orthogonal et qui n'a pas vraiment d'incidence sur le langage de preuves, est que l'on a du mal à contrôler le comportement des stratégies (finalement assez peu configurables). Souvent, un pas très élémentaire de preuve, décomposé à des fins pédagogiques par exemple, ne peut pas être réalisé. D'un point de vue plus théorique, PVS utilisant une logique classique, les utilisateurs voulant faire exclusivement des preuves intuitionnistes sont très désavantagés car ne pouvant pas utiliser les stratégies, qui, à tout moment et quand elles le peuvent, font appel au tiers exclu. Ce manque de contrôle pénalise donc assez fortement certaines utilisations du prouveur qui peuvent être complètement souhaitables.

Enfin, une dernière remarque concerne l'efficacité du vérificateur de PVS. Malgré un code interprété (en Lisp sous Emacs), on peut dire que les règles et même les stratégies sont plutôt rapides, bien que notre petit exemple ne permette pas réellement de s'en rendre compte.

2.2 HOL

2.2.1 Introduction

HOL [35] est un outil d'aide à la preuve directement issu de LCF (que l'on désigne maintenant comme Edinburgh LCF [36] puisque la version originale a été implantée à Edinburgh dans le début des années 70 par l'équipe de Robin Milner). Gérard Huet (INRIA) a porté le code écrit en Stanford Lisp [74] vers Franz Lisp [30] et ce portage fut développé par la suite par Larry Paulson à Cambridge. Cette version devint alors Cambridge LCF [68, 69]. HOL a été implanté, par Mike Gordon et Tom Melham, au-dessus d'une des premières versions de Cambridge LCF et ce système a donc hérité de nombreuses caractéristiques de LCF de manière générale. La première version distribuée fut HOL88 en 1988 bien que le système était déjà utilisé des années auparavant. Au tout début des années 1990, HOL90, une implantation de HOL utilisant Standard ML [60, 61], a été développé par Konrad Slind, à l'origine à l'université de Calgary. Une version utilisant également Standard ML a été développée commercialement par ICL Secure Systems et est devenu maintenant ProofPower.

La logique utilisée en HOL est une variante de la logique des types simples de Church avec tiers exclu. En effet, HOL reposant entièrement sur son métalangage (pour construire ses tactiques[14] mais aussi en tant que toplevel), il en utilise aussi le système de types. Ainsi, il peut utiliser gratuitement le polymorphisme (bridé) de Standard ML pour, entre autres, l'égalité et les quantificateurs. Un avantage de cette hiérarchie est que les univers des types et des termes sont distincts, ce qui tend à clarifier nettement la sémantique (ensembliste).

Concernant l'automatisation, HOL est plus automatisé que Coq (potentiellement dû au contexte de logique classique, qui permet soit plus d'automatisations, soit des automatisations plus simples[15]), mais moins que PVS qui, comme on l'a vu précédemment, possède des

---

où la validation d'un projet est aussi importante que l'explicitation de cette validation.

[14] Le mot "tactique" provient du vocabulaire LCF et désigne toute fonction codée dans le métalangage (appelé ML [34] à l'époque de LCF) pour automatiser des preuves formelles.

[15] Par exemple, la logique propositionnelle est décidable à la fois en logique classique et en logique intuitionniste. Toutefois, en logique propositionnelle classique, on peut faire des preuves sans coupures et sans contractions, alors qu'en logique propositionnelle intuitionniste, on peut faire des preuves uniquement sans coupures. La conséquence est que l'implantation de la procédure de décision en logique classique est immé-

procédures de décision puissantes. Toutefois, un avantage de HOL par rapport à PVS est que le sytème est complètement ouvert à l'utilisateur qui peut coder ses propres tactiques dans le métalangage de HOL (Standard ML).

2.2.2 Codage de la preuve

Pour réaliser notre preuve, nous avons utilisé HOL Light [39], développé par John Harrison en 1996 et qui fonctionne avec Caml Light [53]. C'est une version réimplantée et simplifiée des autres versions de HOL (surtout HOL90). C'est un outil très complet, petit et qui, de fait, s'exécute parfaitement sur un large panel de machines[16].

Il n'y a pas (encore) de système d'environnement dédié à HOL Light et tout se passe dans le toplevel de Caml Light. Les preuves sont effectuées dans ce toplevel ainsi que la construction des tactiques[17]. Il n'y a pas non plus de mécanisme de compilation de preuves que l'on peut éventuellement sauvegarder sur disque. Ainsi, au chargement, toutes les preuves de la bibliothèque sont exécutées avant de donner la main à l'utilisateur[18]. Une quotation est utilisée pour donner une syntaxe concrète aux termes de la logique et un filtre (petit programme en C) permet de préprocesser les scripts afin de déplier la fonction de quotation.

La preuve de décidabilité de l'égalité sur les entiers naturels se fait donc de la manière suivante :

```
g `!(n:num) (m:num). (n=m) \/ ~(n=m)`;;
e INDUCT_TAC;;
e INDUCT_TAC;;
e DISJ1_TAC;;
e ARITH_TAC;;
e DISJ2_TAC;;
e ARITH_TAC;;
e INDUCT_TAC;;
e DISJ2_TAC;;
e ARITH_TAC;;
e (FIRST_ASSUM (fun thm -> DISJ_CASES_TAC (SPEC `m:num` thm)));;
e DISJ1_TAC;;
e (ASM_REWRITE_TAC []);;
e DISJ2_TAC;;
e (REWRITE_TAC [SUC_INJ]);;
e (FIRST_ASSUM ACCEPT_TAC);;
```

diate, alors qu'en logique intuitionniste, elle nécessite des détections de cycles ou la construction d'un calcul des séquents spécifique (voir [28]).

[16]Ceci est dû, entre autres, à la grande portabilité de Caml Light, et au fait que HOL Light ne nécessite qu'une configuration modeste pour s'exécuter.

[17]Avec HOL90 et, par la suite, HOL Light, on assiste à une évolution du statut du métalangage qui n'est plus seulement dédié à la construction de tactiques. ML ayant mûri vers un véritable langage de programmation, il est aussi utilisé comme langage d'implantation. On peut voir cette évolution comme une recherche d'homogénéisation assez logique qui cherche à limiter le nombre de langages intervenant dans le prouveur ainsi qu'une volonté de ne plus être limité dans l'écriture des tactiques. Toutefois, il est important aussi de constater que se pose alors le problème (nouveau) de l'interface utilisateur pour la construction des tactiques. L'expérience montre que cette interface est, en général, un peu floue, peu ou pas documentée et jamais exhaustive. L'utilisateur est donc tenté d'appeler ou de créer des fonctions plus profondes du système et s'expose donc à une maintenance potentiellement difficile en fonction des évolutions du système. Ainsi, dans cette évolution, on perd une certaine portabilité.

[18]Selon la machine, cela peut prendre plusieurs minutes et un système de preuves compilées serait sans doute une amélioration significative pour HOL Light.

Dans ce script, la fonction g (fonction de Caml Light) permet d'entrer dans le mode de preuve. Le théorème à montrer est saisi tout de suite après, entre deux *backquotes* (quotation pour les termes). Dans cette formule, `num` représente le type des entiers naturels, `!`, la quantification universelle, `\/`, le ou logique $\vee$ et `~`, la négation $\neg$. Chaque tactique est ensuite donnée par l'intermédiaire de la fonction `e`. La politique en HOL Light est de mettre le nom des tactiques entièrement en majuscules.

À la ligne 2, avec la tactique `INDUCT_TAC`, on réalise la première récurrence implicitement sur la première variable universellement quantifiée, à savoir `n`. À la ligne 3, on est dans le cas `n=0` et on réalise alors le raisonnement par cas sur `m` avec `INDUCT_TAC`[19]. Aux lignes 4 et 5, on est dans le cas où `n=0` et `m=0`, on choisit alors le cas gauche avec `DISJ1_TAC` puis on résout l'égalité `0 = 0` avec la tactique d'automatisation sur les entiers naturels `ARITH_TAC`[20]. Le cas successeur de `m` est traité aux lignes 6 et 7 où l'on choisit le cas droite avant d'appeler `ARITH_TAC`. Ensuite, on doit traiter le cas successeur de `n` en faisant un raisonnement par cas sur `m` avec `INDUCT_TAC` à la ligne 8. Pour le cas `m=0`, on doit choisir le cas droit avec `DISJ2_TAC` à la ligne 9 puis résoudre avec `ARITH_TAC` à la ligne 10. Reste alors le cas successeur-successeur que l'on traite à la ligne 11 en raisonnant par cas sur l'hypothèse de la première récurrence effectuée (sur `n`) et instantiée avec `m`. À cette ligne, `SPEC` permet d'instantier (spécialiser) les hypothèses universellement quantifiées, `DISJ_CASES_TAC` élimine le connecteur $\vee$ et enfin `FIRST_ASSUM` parcourt l'ensemble des hypothèses pour appliquer la fonction en argument à la première hypothèse pour laquelle ce sera possible[21]. Dans le cas `n=m`, on choisit le cas gauche avec `DISJ1_TAC` à la ligne 12, et pour conclure, on réécrit dans la conclusion avec `ASM_REWRITE_TAC` [22], qui est une tactique de réécriture utilisant les hypothèses comme base de règles (on peut lui donner d'autres lemmes en complément dans la liste en argument). Enfin, pour le dernier cas, il faut choisir le cas droit avec `DISJ2_TAC` à la ligne 14. Pour conclure, on réécrit dans la conclusion par la tactique `REWRITE_TAC` (qui n'utilise pas les hypothèses comme base de règles mais uniquement la liste qui lui est donnée en argument) avec le lemme `SUC_INJ` qui statue que `!m n. (SUC m = SUC n) = (m = n)`. La conclusion a alors exactement la même forme que l'hypothèse du dernier raisonnement par cas et on termine la preuve en ligne 16 en utilisant la première hypothèse de la forme de la conclusion. Ceci est fait, entre autres, par `ACCEPT_TAC` qui résout si le théorème donné en argument est bien de la forme de la conclusion.

Tout comme en PVS, il n'y a pas de mode batch en HOL Light. Cette preuve est donc saisie et vérifiée en interactif dans le toplevel de Caml Light. La session complète peut être consultée en annexe A, section A.2.1. Toutefois, il est possible de donner, de même qu'en PVS, cette preuve d'un seul bloc. Ainsi, on peut, si on le désire, déclarer le lemme et même demander le temps de vérification pour la preuve globale. Pour cette preuve, cela ressemble au script suivant :

```
let eq_nat_dec = time prove
```

[19] En HOL Light, il n'y a pas de tactique pour réaliser du raisonnement par cas sur les entiers naturels. On utilise donc le schéma de récurrence sans utiliser l'hypothèse de récurrence.

[20] On aurait pu aussi utiliser directement la tactique `REFL_TAC` qui résout ce genre d'égalités triviales, mais nous avons opté pour `ARITH_TAC` par souci d'homogéinité par rapport au cas successeur.

[21] En HOL Light, les hypothèses sont numérotées par défaut et on ne peut pas les utiliser directement dans les termes. Ainsi, l'utilisation d'une hypothèse numérotée est plutôt difficile et une série de fonctionnelles (plus ou moins précises) permettent d'accéder à ces hypothèses. Il est également possible de nommer (là encore, plus ou moins facilement) certaines hypothèses pour les utiliser ensuite par leur nom (mais sans toutefois pouvoir les utiliser directement dans les termes). De manière générale, la gestion des hypothèses est assez peu ergonomique.

[22] On aurait bien aimé pouvoir utiliser à nouveau la tactique `ARITH_TAC` mais, étrangement, elle échoue. La sémantique de cette tactique n'étant pas décrite formellement, nous n'avons pas poussé plus loin notre investigation.

```
('!(n:num) (m:num). (n=m) \/ ~(n=m)',
 INDUCT_TAC THENL
   [INDUCT_TAC THENL [DISJ1_TAC THEN ARITH_TAC;DISJ2_TAC THEN ARITH_TAC];
    INDUCT_TAC THENL
      [DISJ2_TAC THEN ARITH_TAC;
       (FIRST_ASSUM (fun thm -> DISJ_CASES_TAC (SPEC 'm:num' thm))) THENL
         [DISJ1_TAC THEN (ASM_REWRITE_TAC []);
          DISJ2_TAC THEN (REWRITE_TAC [SUC_INJ]) THEN
            (FIRST_ASSUM ACCEPT_TAC)]]]);;
```

Dans ce script, le lemme `eq_nat_dec` est déclaré par le mécanisme de déclaration de Caml Light. Ainsi, les lemmes sont des liaisons de l'environnement de Caml Light. Pour construire le lemme, on utilise la fonction `prove` qui prend un couple dont la première composante est la formule à montrer et la deuxième composante, la tactique (d'un seul bloc) qui prouve cette formule. Pour avoir le temps d'exécution, on utilise la fonction `time` qui prend en argument une fonction et son argument et rend le résultat de l'application de cette fonction à son argument en affichant le temps d'exécution. La réponse du toplevel a été mise en annexe A, section A.2.2. Pour construire ce script, on a utilisé des *tacticals*[23] (deux pour être exact), c'est-à-dire des combinateurs de tactiques. Tout d'abord, il y a `THEN` qui est une fonction infixe de Caml Light de la forme `tac`$_1$ `THEN tac`$_2$. Elle applique `tac`$_1$ au but courant puis `tac`$_2$ aux sous-buts générés par cette application. Enfin, `THENL` est aussi une fonction infixe de la forme `tac`$_1$ `THEN [tac`$_2$ `; ... ; tac`$_n$`]`. Elle applique `tac`$_1$ au but courant puis `tac`$_2$ au premier sous-but généré par cette application, ..., et `tac`$_n$ au $(n-1)$ème. Il peut y avoir plus de sous-buts générés que de tactiques dans la liste et la sémantique reste la même. Par contre, s'il y a plus de tactiques que de sous-buts générés, une erreur est levée.

2.2.3 Remarques

Une première remarque à propos du langage de preuves de HOL concerne la lisibilité des scripts. Tout comme en PVS, le lecteur perd très vite ses repères et, à un endroit précis du script, il ne peut pas toujours donner précisément l'état de la preuve, puisque la structure de la preuve n'apparaît pas dans le script. Un script n'a donc clairement pas de sens sans la boucle de vérification interactive qui permet de rejouer des scripts afin de distinguer les lignes (directrices) de la preuve. Les tactiques de HOL Light demandent même encore moins d'information que dans le cas PVS. Ceci est visible pour la tactique d'induction `INDUCT_TAC` qui fait la récurrence implicitement sur la première variable quantifiée, ainsi que pour le raisonnement par cas sur l'hypothèse de la première récurrence (ligne 11) où l'on a juste à donner le terme qui spécialise. Le fait de ne pas pouvoir manipuler convenablement les hypothèses n'arrange guère la situation puiqu'on est obligé d'utiliser des `FIRST_ASSUM` (lignes 11 et 16) qui ne donnent aucune information sur l'hypothèse sélectionnée.

De même qu'en PVS, il est indiscutable que l'usage des tacticals permet de donner un peu plus de structure à la preuve. Mais, ces combinateurs ne sont guère plus précis qu'en PVS (on retrouve quasiment les mêmes[24]) et sur des exemples plus réalistes, on imagine bien le peu d'aide qu'ils fournissent.

En HOL, il n'y a pas de format de preuves indépendant du système. Les preuves sont décrites comme étant une suite de déductions, encodées par un type concret ML. Comme en PVS, on peut estimer que ce format propriétaire est très nuisible à la transparence des

[23]Cela fait partie du vocabulaire de HOL Light, et plus généralement de celui de LCF.

[24]Il faut dire que la preuve est (volontairement) petite et qu'il n'y a pas, *a priori*, beaucoup d'autres possibilités.

preuves et qu'il n'est pas clair qu'un utilisateur (voire un client) ait envie de faire une entière confiance en un prouveur dont l'implantation peut potentiellement contenir des bugs. Ainsi, la preuve d'un théorème peut augmenter, de manière significative, la confiance dans sa validité, mais, en aucun cas, cela ne peut l'assurer, ni fournir les moyens pour le faire.

Concernant la facilité d'écriture, HOL est plutôt gagnant par rapport à PVS. En effet, l'automatisation de HOL étant bien moindre que celle de PVS, la granularité de la preuve s'en trouve fortement améliorée. Ainsi, la petite preuve, que nous venons de réaliser, peut être construite complètement fidèlement au cheminement que nous avions prévu. Aucun pas de preuve n'est effectué subrepticement par des tactiques de HOL Light contrairement à PVS qui cherchait à conclure le plus vite possible (même si on ne le désirait pas). Ce contrôle est très appréciable même si, comme nous l'avons dit précédemment, il pourrait être grandement amélioré par une meilleure gestion du contexte local.

À propos de la maintenance, les changements du système risquent d'avoir plus de conséquences néfastes en HOL qu'en PVS puisque les preuves sont décrites plus précisément (du fait d'une automatisation moins forte). S'il y a plus de tactiques dans le script de preuves, il y a plus de chances d'avoir des problèmes de compatibilité si certaines tactiques évoluent de manière non conservative. Par ailleurs, ces tactiques n'étant pas de grosses machines de preuves comme en PVS, les développeurs sont plus tentés de les faire évoluer de manière non conservative quand cela s'avère nécessaire puisqu'ils estiment qu'il y aura peut-être beaucoup d'occurrences (dans les scripts de preuves) à modifier mais faiblement à chaque fois. En ce qui concerne les changements de théories, HOL y reste complètement sensible et sans doute plus que PVS puisque il est moins automatisé.

Ce manque d'automatisation de HOL oblige, il est vrai, à être plus précis qu'en PVS, où il suffit parfois d'une ou deux règles pour conclure (de manière non triviale). Néanmoins, il ne faut pas croire que la lisibilité des scripts HOL s'en trouve particulièrement accrue. C'est peut-être vrai pour des petites preuves mais sur des exemples plus réalistes, cette différence est très vite gommée.

Enfin, on peut faire une dernière remarque sur l'efficacité de HOL Light. La vérification des preuves semble efficace comme on peut le voir dans la section 2.2.2 avec la preuve utilisant les tacticals. Aucune différence notable n'a pu être observée avec PVS mais il faut dire que cette preuve est un petit exemple. Cependant, il est clair que l'efficacité pourrait être augmentée si HOL Light était porté en Objective Caml (même en bytecode). L'idéal serait bien sûr de créer une structure d'environnement afin de pouvoir fournir toutes les bibliothèques sous la forme d'un fichier ML compilé en natif et linké à un toplevel spécifique.

2.3 Coq

2.3.1 Historique

Coq, au même titre que HOL (voir section 2.2.1), est issu de LCF et plus particulièrement de Cambridge LCF (version de LCF portée de Stanford Lisp en Franz Lisp). Toutefois, Coq l'est dans une moindre mesure (par rapport à HOL) puisque la première implantation, commencée en 1984 et distribuée en 1989 (version 4.10) par Gérard Huet et Thierry Coquand à l'INRIA de Rocquencourt, n'a pas été réalisée à partir d'une version de Cambridge LCF mais à partir de rien en utilisant Caml, un langage fonctionnel de la famille de ML développé au sein du même projet. De LCF, a été repris le système de tactiques[25] (écrites ici en Caml) et pouvant être exécutées par le prouveur. Cette première version est basée sur la

[25]Voir la section 2.2.1 pour avoir des précisions sur ce terme. On reviendra, plus précisément et plus longuement, sur le système de tactiques (en particulier dans le cadre de Coq) dans la partie II.

théorie des types (Whitehead-Russell, 1910) à l'ordre supérieur en utilisant un λ-calcul typé appelé Calcul des Constructions, CoC (Thierry Coquand, 1985, [19]). Ce système logique (à la base intuitionniste) permet, au-delà du codage de la théorie des ensembles, d'encoder intégralement les preuves du calcul (au moyen de λ-termes) selon l'isomorphisme de Curry-Howard. Ce codage des preuves permet d'en extraire des programmes (vers des langages fonctionnels), qui, dès lors, sont sûrs de vérifier la spécification qui en a été faite[26].

Parmi les évolutions les plus significatives du système, on notera, par la suite, l'introduction des définitions inductives primitives par Christine Paulin en 1991 (version 5.6), la construction d'un nouveau moteur de preuves à base d'arbres et d'un contexte permettant à l'utilisateur d'écrire ses propres tactiques par Chet Murthy en 1994 (version 5.10), l'apparition du filtrage à la ML par Cristina Cornes en 1996 (version 6.1) et enfin, une restructuration en profondeur du code par Jean-Christophe Filliâtre et Hugo Herbelin ainsi que la conception d'un nouveau langage de tactiques (dont on parlera dans la partie II) par l'auteur en 2001 (version 7.0, [84]). De manière externe, une interface graphique, appelée d'abord CtCoq [7, 9] puis Pcoq [51], a aussi été développée par le projet Lemme, à l'INRIA de Sophia-Antipolis, et se base sur une approche générale de conception d'interfaces utilisateurs pour les outils d'aide à la preuve [86, 8].

Du point de vue de l'automatisation, Coq est moins automatisé que PVS et HOL de par, entre autres, l'utilisation de la logique intuitionniste[27], dans laquelle les preuves peuvent être plus difficiles qu'en logique classique. Cependant, tout comme en HOL, le système est ouvert et l'utilisateur peut écrire ses propres tactiques en ML (en Objective Caml, plus précisément). La différence par rapport à HOL est que l'interfaçage de ces tactiques est moins évident en Coq dans la mesure où le choix d'un toplevel dédié (afin d'avoir un contrôle total, notamment de la syntaxe) a été fait. Cet inconvénient a grandement motivé l'enrichissement du métalangage à toplevel[28] (ainsi que de sa communication avec le monde des tactiques écrites en ML). Malgré cette ouverture, le nombre de tactiques reste assez faible même si l'objectif ici est plutôt de fournir à l'utilisateur un contexte solide et clair pour programmer ses tactiques[29].

2.3.2 Script de la preuve

Pour réaliser la preuve, nous avons utilisé la dernière version de Coq (V7), bien qu'en réalité, aucune spécificité de la V7 ne soit nécessaire, et que le script ait de fortes chances d'être vérifié tel quel dans des versions antérieures. Cette preuve peut être construite au moyen du script suivant :

```
Lemma eq_nat:(n,m:nat)n=m\/~n=m.
Proof.
  Induction n.
  Intro.
  Case m.
```

[26]Ce principe peut être vu comme étant complètement le dual du *model-checking*, où l'utilisateur écrit son programme puis fait des preuves pour montrer que son programme vérifie la spécification.

[27]Dans certaines *sortes* de Coq, il est possible de faire des preuves classiques mais aucune procédure de décision dédiée n'est disponible. La politique serait de ne pas concurrencer tous les prouveurs sur leurs domaines de prédilection. Toutefois, dans cette optique, on ne peut plus à proprement parler de polyvalence de Coq, ce qui peut être néfaste pour son utilisation dans la mesure où l'utilisateur n'est pas forcément prêt à passer d'un système à l'autre selon ses besoins.

[28]Voir la partie II.

[29]La politique de l'équipe (LogiCal) est qu'au même titre que le développement des bibliothèques, l'automatisation de Coq est plutôt du domaine de l'utilisateur, qui peut potentiellement, suivant ses applications, contribuer à l'essor du système.

```
6     Left.
7     Auto.
8     Right.
9     Auto.
10    Intros.
11    Case m.
12    Right.
13    Auto.
14    Intro.
15    Case (H n1).
16    Left.
17    Auto.
18    Right.
19    Auto.
20  Save.
```

En Coq, pour entrer dans le mode d'édition de preuves, il y a plusieurs moyens, ici, nous avons choisi la commande `Lemma` qui permet de nommer (`eq_nat`) la proposition (à droite des `:`) que l'on veut montrer. Dans la syntaxe utilisée pour formuler la proposition, il n'y a pas trop de différence avec HOL : les premières parenthèses sont utilisées pour représenter un produit dépendant[30] (`n`, `m` sont les variables quantifiées et `nat` est leur type), `\/` est le ou logique $\vee$ et `~`, la négation $\neg$. Les tactiques sont ensuite données directement au toplevel (sans commande intermédiaire comme c'était le cas en HOL Light) en se terminant par un point (lexème qui déclenche l'évaluation). Une convention syntaxique (non rigide) est que les tactiques commencent toujours par une majuscule, et ne contiennent généralement pas de majuscules dans le reste du mot. Elles sont suivies par d'éventuels arguments (parfois optionnels) pouvant être donnés par certains mots-clés comme, par exemple, `with`, `using` ou `:=`. Dans tous les cas, il est préférable de se reporter au manuel de référence [84] pour connaître la syntaxe exacte[31] d'une tactique ainsi que de ses variantes.

À la ligne 2, `Proof` n'est pas une tactique mais une commande qui ne fait rien et qui sert à présenter la preuve sous une forme plus parenthésée (avec la commande `Save` à la fin du script). À la ligne 3, on entame la preuve avec la récurrence sur la première variable universellement quantifiée, à savoir `n`, au moyen de la tactique `Induction`[32]. Ensuite, on est dans le cas `n=O` et on introduit la deuxième variable universellement quantifiée `m` avec `Intro` à la ligne 4 pour pouvoir effectuer le raisonnement par cas sur cette deuxième variable avec `Case`[33] à la ligne 5. À la ligne 6, on est dans le cas `n=O` et `m=O`, on choisit donc le cas gauche avec la tactique `Left`[34] et, il ne reste plus qu'à résoudre la proposition `O=O` avec la tactique d'automatisation (de base) `Auto`[35]. Aux lignes 8 et 9, on traite le cas successeur

[30] Dans le vocabulaire des systèmes de types, la quantification universelle est aussi appelée produit dépendant, et l'implication, produit non dépendant. Par produit, on entendra donc soit une quantification universelle, soit une implication.

[31] Même si ce toplevel dédié permettant de donner un peu de sucre syntaxique aux tactiques semble bien peu flexible, car il nécessite d'avoir le manuel de référence à portée de main, il est important de comprendre que cette syntaxe (si elle est utilisée judicieusement) permet de donner un peu de sémantique aux appels de tactiques (contrairement à de simples applications comme en HOL Light) contribuant, par là-même, à accroître sensiblement la lisibilité des scripts.

[32] Cette tactique s'applique directement sur les produits (ce qui peut s'avérer propice). Si le produit visé par la récurrence a déjà été introduit, il faut alors utiliser la tactique `Elim`.

[33] Cette tactique ne peut pas s'appliquer directement sur un produit comme c'est le cas pour `Induction`.

[34] Cette tactique porte un nom dont l'intuition ne peut pas être plus proche de la sémantique. C'est plutôt agréable et, de plus, elle fonctionne également avec des types isomorphes au ou logique.

[35] Cette tactique est une procédure de résolution à la Prolog utilisant des bases de lemmes (appelés, dans ce cas, des `Hints`).

de m en choisissant cette fois le cas de droite avec Right et en résolvant avec Auto. À la ligne 10, on est dans le cas successeur pour n et on réalise à nouveau un raisonnement par cas sur m avec Case[36]. Aux lignes 12 et 13, on est dans le cas successeur pour n et m=O, on choisit donc le cas droite avec Right puis on conclut avec Auto. Reste alors à montrer le cas successeur-successeur, ce qui se fait, à la ligne 15, en raisonnant par cas sur l'hypothèse de récurrence (nommée automatiquement H par le système) sur n instantiée avec n1 provenant du dernier raisonnement par cas sur m (ligne 11). Dans le cas où n0=n1, n0 provenant de la récurrence sur n (ligne 3), on choisit, aux lignes 16 et 17, le cas gauche avec Left[37] puis on résout avec Auto. Enfin, le dernier cas où n0 et n1 ne sont pas égaux est traité aux lignes 18 et 19, en choisissant le cas droite avec Right et en concluant avec Auto. À la ligne 20, la commande Save permet d'extraire un terme-preuve de l'arbre de preuve, vérifie que son type est bien convertible à la proposition montrée et enregistre dans l'environnement, sous le nom eq_nat, cet objet contenant essentiellement le terme et son type.

Cette preuve peut être saisie interactivement dans le toplevel de Coq[38] et la preuve complète avec les réponses du toplevel peut être consultée en annexe A, section A.3.1. Contrairement à PVS et HOL Light, Coq possède également un mode batch ainsi qu'un système de compilation de fichiers[39], même si, on le verra, il est quasiment impossible de construire une preuve directement dans ce mode, c'est-à-dire sans l'aide de la boucle interactive. De même qu'en PVS et HOL Light, on peut donner cette preuve d'un seul bloc (en n'utilisant qu'un seul point) à l'aide des combinateurs de tactiques, appelés, comme en HOL Light, tacticals. En utilisant les tacticals pour compacter le script, on peut obtenir la preuve suivante (saisie en interactif) :

```
Lemma eq_nat:(n,m:nat)n=m\/~n=m.
Proof.
Time Induction n;
  [Intro;Case m;[Left;Auto|Right;Auto]|
   Intros;Case m;
     [Right;Auto|
      Intro;Case (H n1);[Left;Auto|Right;Auto]]].
Save.
```

Étant donné que la preuve est donnée d'un seul coup, on en a profité pour tester le temps d'exécution de la preuve avec la commande Time. Les traces d'exécution ont été mises en annexe A, section A.3.2. Dans ce script, on a utilisé deux tacticals. D'abord, $\texttt{tac}_1$;$\texttt{tac}_2$ applique $\texttt{tac}_1$ au but courant puis $\texttt{tac}_2$ à tous les sous-buts générés par cette dernière application. Ici, pour décrire le plus précisément la structure de la preuve, nous avons utilisé ce tactical uniquement lorsqu'il n'y avait qu'un seul sous-but généré. Le second tactical utilisé est $\texttt{tac}_0$;[$\texttt{tac}_1$|...|$\texttt{tac}_n$], qui applique $\texttt{tac}_0$ au but courant puis $\texttt{tac}_1$ au premier

[36]m a déjà été introduit précédemment (ligne 4).

[37]Le raisonnement par cas sur H génère un produit non dépendant en conclusion du but mais il n'est pas nécessaire de l'introduire au préalable pour utiliser la tactique Left (il en est évidemment de même pour Right).

[38]Suivant l'architecture utilisée, il y a, en V7, deux versions pour le toplevel de Coq : une version compilée en *bytecode* (coqtop.byte) qui permet de charger (dynamiquement) des tactiques écrites en ML et une version compilée en natif (coqtop.opt), en moyenne 5 fois plus rapide que le bytecode, mais qui ne peut pas charger de tactiques en ML (le seul moyen pour les utiliser est de les *linkées* quand on produit le binaire mais c'est un processus complètement statique). Voir le manuel de référence [84] pour plus détails sur l'utilisation de ces toplevels et notamment sur la commande coqmktop qui permet de construire des toplevels *customisés*.

[39]Il s'agit de la commande coqc. Les fichiers sources .v sont compilés en .vo et le langage utilisé est exactement le même qu'en interactif. Voir le manuel de référence [84] pour plus de détails.

sous-but généré, ..., et $\texttt{tac}_{\texttt{n}}$ au n-ième sous but généré. Le nombre de sous-buts générés doit être exactement égal au nombre de tactiques entre les `[...]`[40].

2.3.3 Observations

Concernant la lisibilité des scripts Coq, la remarque est complètement similaire à celle que l'on a pu faire à propos des scripts PVS ou HOL Light. L'état de la preuve est rapidement perdu au fil des instructions données au prouveur (même sur ce petit exemple). Dans cette optique, on atteint un certain paroxysme lorsque l'on voit apparaître de nouvelles variables (ligne 15, avec les variables `H` et `n1`) qui ne sont liées qu'au moment de l'évaluation de l'instruction en question. Généralement, ceci se produit lorsque l'utilisateur introduit des variables sans les nommer (le système s'en charge alors selon une politique de nommage qui lui est propre). Ensuite, l'utilisateur peut être amené à se servir de certaines de ces hypothèses en utilisant directement leur nom[41], ce qui fait apparaître ces variables fraîches. Ce principe est très préjudiciable à la robustesse des scripts qui peuvent ne plus être corrects si la politique de nommage du système venait à être modifiée[42].

Les tacticals améliorent quelque peu la lisibilité du script mais la remarque reste strictement la même que pour PVS et HOL Light, à savoir que sur des exemples plus réels, on perd quasiment aussi vite le fil de la preuve. Par ailleurs, il est important de comprendre qu'en procédural, on construit principalement les preuves en interactif et que l'utilisation (très précise) des tacticals a plutôt lieu localement (sur des parties de preuve). La structuration de la preuve complète avec les tacticals ne peut donc se faire que dans une seconde passe, qui, en général, n'est pas faite ou effectuée de manière partielle[43].

Dans une optique proche et en ce qui concerne le format utilisé pour coder les preuves, Coq tire très nettement son épingle du jeu par rapport à PVS et HOL Light. En effet, Coq utilise la notion d'isomorphisme de Curry-Howard pour traduire ses preuves en termes d'un λ-calcul typé, plus particulièrement le Calcul des Constructions Inductif (CCI). Ce format permet non seulement de revérifier très facilement les preuves (il suffit de typer les termes preuves) mais aussi de fournir une preuve qui peut être vérifiée par un outil externe à Coq[44].

À propos de la facilité d'écriture, là encore, la tendance est de préférer Coq à PVS pour la possibilité de contrôler sa preuve très précisément. On peut facilement réaliser, par

[40] C'est une petite différence de sémantique avec le tactical similaire `THENL` de HOL Light. Cette condition supplémentaire peut sembler rigide mais s'avère utile pour détecter très tôt des bugs, dûs soit à des erreurs lors de la construction de la preuve, soit à des changements du système. Le plus flexible reste sans aucun doute la stratégie `spread` de PVS qui se décline en trois stratégies : `spread` tout court qui permet d'avoir moins de règles à appliquer que de sous-buts générés, `spread!` qui lève une erreur si le nombre de règles à appliquer ne correspond pas au nombre de sous-buts et `spread@` qui lève un warning si le nombre de règles à appliquer ne correspond au nombre de sous-buts générés.

[41] Par rapport à HOL Light, on peut facilement référencer les hypothèses (par leur nom), ce qui, de manière pratique, est très appréciable mais, du point de vue de la lisibilité, cela peut être tout aussi néfaste qu'en HOL Light si l'utilisateur utilise une technique "paresseuse" de nommage des hypothèses.

[42] Toutefois, cet argument doit être atténué dans la mesure où Coq permet le nommage de certaines hypothèses, mais, de ce point de vue là, le système n'est pas rigide, si bien qu'il est possible de ne jamais nommer les hypothèse que l'on introduit. Par ailleurs, certaines tactiques (comme `Inversion`, par exemple) introduisent les hypothèses automatiquement sans que l'utilisateur ait la possibilité de les nommer. Ainsi, de manière générale, il est préférable d'introduire les hypothèses potentiellement réutilisables en les nommant lorsque c'est possible, c'est-à-dire lorsque les tactiques ne le font pas pour l'utilisateur ou lorsqu'il ne s'agit pas d'écrire une tactique.

[43] L'idée est que l'utilisation des tacticals correspond à une notion plutôt batch de construction de preuve et donc éminemment opposée au mode interactif qui est le principe naturel pour élaborer les preuves en procédural.

[44] Le client peut, il est vrai, implanter sa propre version de *typechecker* pour le CCI et revérifier la preuve fournie par Coq. Bien évidemment, il doit avoir confiance dans la cohérence du CCI ainsi que dans l'algorithme de typage, ce qui est assuré par un certain nombre de résultats théoriques connus et accessibles.

exemple, des preuves intuitionnistes (puisque c'est la base du système) sans être ennuyé par des tactiques qui introduisent brutalement le tiers exclu. Par rapport à HOL Light, la gestion du contexte local est bien plus intuitive et flexible en Coq. On gagne un pouvoir d'expression non négligeable et un temps précieux dans l'utilisation des hypothèses.

Concernant la maintenance, les remarques sont globalement les mêmes que pour HOL Light, à savoir que les scripts sont très sensibles aux évolutions du système mais, par ailleurs, beaucoup moins aux changements de spécifications. Toutefois, le système de gestion du contexte local peut, même s'il est indéniable qu'il est plus agréable qu'en HOL Light, s'avérer source de non-robustesse si l'utilisateur ne nomme pas fermement les hypothèses qu'il réutilise par la suite. En HOL Light, l'utilisateur peut très bien ne pas nommer ses hypothèses et les référencer ensuite au moyen d'itérateurs (prédéfinis). De tels scripts ne poseront pas de problèmes de référencement d'hypothèses, puisqu'on s'est complètement affranchi du nom des hypothèses, même si, de ce fait, la lisibilité en pâtit.

Par rapport à l'automatisation, Coq est, on l'a dit précédemment, le moins automatisé des trois prouveurs que nous avons présenté. Mais comme pour HOL Light, même si les preuves sont, de ce fait, décrites plus précisément, elles ne s'en trouvent pas plus lisibles pour autant. Les scripts sont constitués de tactiques qui ont un peu plus de sémantique qu'en PVS (puisque plus ciblées) mais ces mêmes scripts ont tendance a être plus gros qu'en PVS ce qui, d'un autre côté, fait perdre le fil de la preuve.

Enfin, concernant les performances, sur le temps de vérification de preuve proprement dit, on ne peut pas dire qu'il y a une différence significative entre ces trois systèmes[45]. Cependant, par rapport à PVS et HOL Light, Coq effectue, lorsque la preuve est vérifiée et si l'utilisateur désire sauvegarder sa preuve, une phase supplémentaire qui consiste à traduire l'arbre de preuve en λ-terme (dont on parlé précédemment). Cette phase peut, selon les cas, consommer non seulement assez de temps mais aussi beaucoup de mémoire, pouvant rendre Coq, de ce point de vue là, moins performant que ses concurrents. Il faut toutefois voir que c'est un prix à payer totalement délibéré et complètement nécessaire pour convaincre de potentiels clients de la validité des certifications pouvant être effectuées en Coq.

[45] En tous cas, le développement est bien trop petit pour s'en rendre réellement compte.

Chapitre 3

Langages déclaratifs

Dans ce chapitre, il s'agit, de même que dans le chapitre 2, de montrer quelques exemples d'outils d'aide à la preuve utilisant un langage déclaratif. De manière annexe, il nous sera également possible d'observer, toujours au niveau du langage de preuves, les nuances éventuelles pouvant apparaître entre de tels systèmes.

Comme outils, nous avons choisi d'étudier Mizar [88] et ACL2 [47, 48] (descendant direct de Nqthm [12, 13]), qui nous ont semblé probablement les plus représentatifs dans le style déclaratif, d'autant qu'ils sont, de plus, très différents, notamment au niveau de leur automatisation, ce qui, on le verra, influe fortement sur la description des preuves. Comme on a pu le voir dans le chapitre 1, il existe d'autres systèmes utilisant un langage déclaratif, comme Declare [82] ou le prototype de Vincent Zammit [93]. On peut aussi citer le Mizar-mode pour HOL Light, réalisé par John Harrison. Cependant, tous ces outils restent quelque peu expérimentaux, ce qui explique pourquoi nous avons préféré ne pas les considérer dans notre comparaison.

3.1 Mizar

3.1.1 Introduction

Le projet Mizar a débuté dans le début des années 70 sous l'impulsion d'Andrezej Trybulec de la *Plock Scientific Society*, en Pologne. L'idée d'origine était d'implanter un logiciel visant à assister les mathématiciens dans le processus de préparation d'articles. En particulier, il s'agissait de développer un environnement permettant d'écrire des articles traditionnels de mathématiques, avec comme base la logique classique et la théorie des ensembles. La déduction naturelle fut choisie comme style de logique et, quant à la théorie des ensembles, plusieurs furent essayées (Zermelo-Fraenkel [94], Morse-Kelley [62]), mais c'est l'axiomatisation de Tarski-Grothendieck [87], qui fut retenue.

Les premières implantations furent réalisées en 1974-75, avec un vérificateur de preuves modeste pour la logique propositionnelle. En 1977, le système fut étendu avec les quantificateurs pour donner Mizar-QC, qui, dans les années qui suivirent, allait donner Mizar-FC, avec la possibilité d'utiliser des notations fonctionnelles et des définitions. En 1981, Mizar-2 voit le jour avec, entre autres, des types structurés, des définitions en compréhension et l'intégration de fragments de l'arithmétique. Par la suite, il y eut d'autres implantations plutôt expérimentales, comme Mizar-3, Mizar-HPF ou Mizar-MSE. En 1986, Mizar-2 est restructuré pour donner Mizar-4, où l'on peut utiliser des axiomes qui ne seront pas vérifiés. Enfin, en 1988, Andrezej Trybulec complète le langage de Mizar-4, qui est simplement appelé Mizar.

Dans cette version, les *articles* (nom donné aux fichiers de preuves) peuvent dépendre les uns des autres et les nouveaux axiomes sont désormais interdits (tout doit être prouvé).

De manière générale, à travers toutes les versions de Mizar, un gros effort a été réalisé sur le langage objet, afin qu'un mathématicien ne soit pas trop désorienté pour exprimer ces lemmes et en décrire les preuves. De ce fait, les contributions à la *Mizar Mathematical Library* (MML) sont considérables avec, actuellement, 695 articles et plus de 120 auteurs[1]. En contre-partie, l'automatisation du vérificateur a été presque totalement laissée de côté[2], si bien que les preuves en Mizar sont très détaillées.

Mizar n'est pas réellement documenté. Seule la syntaxe autorisée pour les articles est disponible, mais sans aucune indication sémantique. La philosophie, entendue (d'après [79]) et sensiblement atypique, du groupe Mizar est de leur rendre visite (à Bialystok), afin d'apprendre à manipuler le système en réalisant une contribution à la MML. Toutefois, on pourra consulter [65], qui se rapproche le plus d'une documentation, bien que difficilement appréhendable, ou [92], qui est une tentative d'introduction à Mizar. Par ailleurs, certains articles ou manuscrits du site Web de Mizar peuvent potentiellement apporter certaines informations sur le système et sont, en partie, accessibles à l'adresse suivante :

http ://mizar.org/

On y trouvera, de surcroît, les différentes distributions binaires de Mizar[3], ainsi que la dernière version de la MML.

3.1.2 Article de la preuve

Nous allons revenir sur l'exemple de décidabilité de l'égalité sur les entiers naturels et décrire la preuve en Mizar. Comme on l'a vu précédemment, les spécifications Mizar (lemmes et preuves) sont appelées articles et peuvent dépendre les unes des autres. Pour connaître la syntaxe du langage, on pourra se référer à l'adresse suivante :

http ://mizar.org/language/

qui décrit la grammaire du système (mais pas la sémantique correspondante).

Une fois l'article créé, la validation se fait en deux étapes. On utilise d'abord l'*Accomodator*, qui va produire un environnement spécifique pour l'article, à partir des déclarations d'environnement effectuées dans l'article et de la base de données disponible (typiquement les articles de la MML). Ensuite, on peut effectuer la validation proprement dite avec le *Verifier*, qui affiche des remarques sur les fragments non acceptés du texte source. De là, on doit éventuellement corriger l'article et recommencer jusqu'à ce qu'aucune erreur ne soit signalée. L'article sera alors considéré comme valide.

Dans le cas de notre preuve, nous proposons l'article suivant :

```
environ

 constructors NAT_1;
 requirements ARYTM, SUBSET;
```

[1] Le total fait environ 42 Mo.

[2] Toutefois, la règle d'inférence de base by est plus forte qu'il n'y paraît et utilise un système de *corrélations sémantiques*, introduit par Roman Suszko, dans ses travaux sur la logique qui n'est pas de Frege. Pour plus de détails sur le by, voir [91].

[3] À titre indicatif, Mizar est codé en Pascal.

```
 notation ARYTM, NAT_1, SUBSET_1;
 clusters ARYTM, ARYTM_3;
 theorems REAL_1, NAT_1;
 schemes NAT_1;

begin

reserve k,m,n,m0,n0 for Nat;

theorem
  eq_nat: n = m or not (n = m)
  proof
    A1: for m holds 0 = m or not (0 = m)
    proof
      A2: 0 = 0 or not (0 = 0)
      proof
        A3: 0 = 0;
        hence thesis by A3;
      end;
      A4: (0 = m0 or not (0 = m0)) implies (0 = m0 + 1 or not (0 = m0 + 1))
      proof
        assume 0 = m0 or not (0 = m0);
        A5: not (0 = m0 + 1) by NAT_1:21;
        hence thesis by A5;
      end;
      for k holds 0 = k or not (0 = k) from Ind(A2,A4);
      hence thesis;
    end;
    A6: (for m holds (n0 = m or not (n0 = m))) implies
        (for m holds ((n0 + 1) = m or not ((n0 + 1) = m)))
    proof
      assume A7: for m holds (n0 = m or not (n0 = m));
      A8: (n0 + 1) = 0 or not ((n0 + 1) = 0)
      proof
        A9: not ((n0 + 1) = 0) by NAT_1:21;
        hence thesis by A9;
      end;
      A10: ((n0 + 1) = m0 or not ((n0 + 1) = m0)) implies
           ((n0 + 1) = (m0 + 1) or not ((n0 + 1) = (m0 + 1)))
      proof
        assume (n0 + 1) = m0 or not ((n0 + 1) = m0);
        A11: n0 = m0 implies ((n0 + 1) = (m0 + 1) or not ((n0 + 1) = (m0 + 1)))
        proof
          assume A12: n0 = m0;
          A13: (n0 + 1) = (m0 + 1) by A12,REAL_1:10;
          hence thesis by A13;
        end;
        A14: not(n0 = m0) implies
             ((n0 + 1) = (m0 + 1) or not ((n0 + 1) = (m0 + 1)))
        proof
```

```
55             assume A15: not(n0 = m0);
56             A16: not ((n0 + 1) = (m0 + 1)) by A15,REAL_1:10;
57             hence thesis by A16;
58           end;
59           hence thesis by A11,A14,A7;
60         end;
61         for k holds ((n0 + 1) = k or not ((n0 + 1) = k)) from Ind(A8,A10);
62         hence thesis;
63       end;
64       for k,m holds k = m or not (k = m) from Ind(A1,A6);
65       hence thesis;
66     end;
```

De la ligne 1 à la ligne 8, il s'agit des déclarations d'environnement de l'article, à savoir ses dépendances vis-à-vis des autres articles de la bibliothèque[4]. En ligne 12, on *réserve* certains identificateurs pour le type des entiers naturels (`Nat`), ce qui permet d'utiliser une quantification universelle implicite, par la suite, dans l'expression des propositions. En lignes 14 et 15, on exprime notre théorème (`n` et `m` sont bien universellement quantifiés), et, de la ligne 16 à 66, c'est la preuve proprement dite.

L'avantage de la preuve Mizar est qu'elle est très proche d'une preuve mathématique ordinaire et se passe presque de commentaires, compte-tenu du choix judicieux des mots-clés effectué par le groupe Mizar. En effet, comme on a d'abord décidé de faire une récurrence sur `n`, on établit d'abord, en ligne 17, le cas de base comme un nouveau lemme à prouver. Ce lemme intermédiaire est associé à un *label*, pour pouvoir être référencé par la suite dans le reste de la preuve. Ensuite, en ligne 19, c'est le cas de base du raisonnement par cas sur `m`, qui est résolu en posant le cas gauche (trivialement résolu par le système), en lignes 21 et 22. En ligne 24, c'est le cas récurrent sur `m` (il n'y a effectivement pas de raisonnement par cas seulement en Mizar). En ligne 26, on suppose l'hypothèse de récurrence sur `m` (que l'on n'utilisera donc pas), puis on résout, en lignes 27 et 28, en choisissant le cas droit (prouvé par le théorème `NAT_1:21`. Le cas de base est ainsi prouvé, en ligne 30 et 31, en utilisant le schéma d'induction sur les entiers naturels (`Ind`) avec les lemmes intermédiaires concernant le raisonnement par cas sur `m` (`A2` et `A4`). En ligne 33, on traite le cas récurrent sur `n`. On commence par introduire l'hypothèse de récurrence (que l'on va utiliser plus tard), en ligne 36. On fait alors le deuxième raisonnement par cas sur `m`. Le cas de base est posé en ligne 37, et résolu trivialement, en lignes 39 et 40, en prenant le cas droit. Le cas récurrent est énoncé en lignes 42 et 43. On introduit l'hypothèse de récurrence (qui, de même que précédemment, ne sera pas utilisée) en ligne 45. De là, on raisonne par cas sur l'égalité des deux prédécesseurs (`n0` et `m0`). En ligne 46, on suppose que `n0` et `m0` sont égaux, puis on suppose l'égalité, en ligne 48, pour conclure l'égalité des deux successeurs, en lignes 49 et 50. En lignes 52 et 53, on est dans le cas où `n0` et `m0` sont distincts et on suppose alors l'inéquation, en ligne 55, pour montrer que les deux successeurs ne sont pas égaux, en lignes 56 et 57. Le cas récurrent-récurrent est complété, en ligne 59, en utilisant les deux lemmes intermédiaires précédents, ainsi que l'hypothèse de récurrence sur `n`[5]. Des lignes 61 à 62, on utilise, de même que précédemment, le schéma de récurrence sur les entiers naturels pour conclure sur le raisonnement par cas de `m`, et encore une fois, en lignes 64 et 65, pour la récurrence sur `n`, afin de compléter la preuve globale.

[4] On peut remarquer, au passage, qu'il s'agit de donner non seulement les articles utilisés, mais aussi la catégorie des objets nécessaires (constructeurs, théorèmes, schémas, ...).

[5] Le pas d'inférence réussi par `by` n'est ici pas trivial, puisqu'il faut *deviner* l'instantiation de l'hypothèse de récurrence sur `n` avec `m0`.

Pour valider notre article, on utilise d'abord l'Accomodator pour construire l'environnement :

```
Accommodator, Mizar Ver. 6.0.10 (Linux/FPC)
Copyright (c) 1990,2000 Association of Mizar Users
Processing: eq_nat_dec.miz

-Parsing-Vocabularies-Constructors-Clusters-Notation
-Theorems-Schemes
```

Ensuite, on lance le Verifier pour effectuer la validation :

```
Verifier, Mizar Ver. 6.0.10 (Linux/FPC)
Copyright (c) 1990,2000 Association of Mizar Users
Processing: eq_nat_dec.miz

Parser   [  68]   0:00
Analyzer [  68]   0:00
Checker  [  68]   0:00
Time of mizaring: 0:00
```

Aucune erreur n'est signalée, notre preuve est donc correcte.

3.1.3 Remarques

Comme on a pu déjà le voir précédemment, on peut noter que le script est très lisible. Il ressemble, en effet, assez fortement à une preuve mathématique traditionnelle exprimée en langage naturel. Le raisonnement est, il est vrai, globalement dirigé vers l'avant (mode *forward*), où l'on part de ce que l'on connaît (axiomes ou théorèmes déjà prouvés) vers ce que l'on cherche à prouver, en *construisant* des lemmes intermédiaires de plus en plus *complexes*, de manière à pouvoir conclure. Toutefois, dans certains cas, il arrive que le lemme intermédiaire posé (et à prouver en premier) se réduise assez trivialement sur le lemme global qu'il faut montrer. Dans ces situations, même si *en principe*, le mode est forward, le raisonnement peut être aussi considéré comme dirigé vers l'arrière (mode *backward*), puisque l'utilisateur s'est clairement servi du lemme original pour construire son lemme intermédiaire. La lecture de telles preuves peut donc être sujet à interprétation et on peut penser que l'on a un mode mixte, comme lorsque l'on construit des preuves mathématiques *sur le papier*. Une autre raison à la bonne lisibilité de ces preuves réside justement dans l'existence de ces lemmes intermédiaires, qui permettent de connaître *pratiquement* partout l'état de la preuve. On comprend donc aisément que Mizar fonctionne exclusivement en mode batch et qu'un toplevel n'est pas nécessaire dans le processus de construction de preuves.

Cette grande lisibilité a tendance, en contre-partie, à rendre, d'une certaine manière, les preuves difficiles à écrire. Les lemmes intermédiaires obligent, en effet, à répéter beaucoup d'informations, et cela peut être considéré comme fastidieux. C'est à relier directement avec la maintenance des articles, qui sont, de ce fait, très sensibles aux changements de spécification. Par exemple, si l'on doit inverser les arguments d'une fonction, il est clair que le portage ne devrait pas être difficile, mais s'il faut rajouter un argument, il n'est pas évident de pouvoir trouver les *bons* outils d'édition pour corriger les articles concernés par la modification. À l'opposé, la maintenance système semble plutôt facile, puisque Mizar est peu automatisé et que les preuves sont très détaillées. Il n'y a pas de procédures de décision et la seule *tactique* à surveiller est `by`, qui peut être amenée, comme on l'a vu dans notre preuve, à effectuer des pas d'inférence non triviaux. Toutefois, si elle est bien *contrôlée*, c'est-à-dire si elle évolue de manière conservative, elle ne doit pas poser de problèmes particuliers.

Enfin, de manière un peu annexe, on peut remarquer que Mizar n'a pas de format de preuves indépendant. Il doit donc être clair pour l'utilisateur, qu'il doit faire confiance non seulement à la théorie de Mizar (cela semble plutôt raisonnable), mais aussi à l'implantation qui en est faite (c'est déjà plus difficile). Ainsi, comme pour les prouveurs PVS et HOL Light, précédemment étudiés, une preuve vérifiée est donc un gage supplémentaire de confiance, mais pas une certitude absolue. Par ailleurs, en ce qui concerne les performances, notre petit exemple n'est pas suffisant pour être significatif, mais, en moyenne, sur des validations effectuées sur certains articles de la MML, on a pu observé des performances tout à fait acceptables.

3.2 ACL2

3.2.1 Historique

ACL2 [47, 48] est le descendant direct du prouveur Nqthm, qui est, comme Mizar, assez ancien et date aussi des années 70. Nqthm a été développé par Robert S. Boyer et J. Strother Moore, à l'Université du Texas, à Austin [12, 13]. Par la suite, une interface interactive puissante (Pc-Nqthm) fut réalisée pour le système, par Matt Kaufmann en 1987 [46]. Depuis 1988, ACL2 (*A Computational Logic for Applicative Comman Lisp*) lui succède et fut d'abord développé par la société *Computational Logic*, puis ensuite, à partir de 1997, à l'Université du Texas (toujours à Austin).

La logique d'ACL2 est celle de Boyer-Moore [12, 13], à savoir une logique classique du premier ordre avec des fonctions récursives totales permettant la récurrence mathématique. Quant à la syntaxe, on utilise exclusivement Common Lisp [81]. Ainsi, on peut voir les spécifications d'ACL2 comme des théories de fonctions récursives (définies ou axiomatisées), pour lesquelles on établit (et prouve) des propriétés. Le mot calculable (*computational*) prend alors tout son sens dans l'acronyme ACL2, et ce, pour trois raisons principales. D'abord, parce que tout calcul effectif (mathématique) peut être codée comme une fonction récursive. Ensuite, parce que la plupart des spécifications d'ACL2 sont directement exécutables, et les fonctions correspondantes peuvent être données à d'autres compilateurs Common Lisp. Enfin, parce que le calcul est très souvent utilisé dans le processus de raisonnement, si bien que l'on peut simplifier des expressions par calcul, plutôt que par des *manipulations* logiques complexes.

Du point de vue de l'automatisation, ACL2 est très performant. Les preuves données automatiquement sont très verbeuses et, de ce fait, très grandes. Ceci est dû à la combinaison sophistiquée de procédures de décision, de réécritures (conditionnelles), d'inductions (mathématiques et structurelles), de simplifications propositionnelles et d'heuristiques gérant les interactions entre ces fonctionnalités. Toutefois, ACL2 ne produit pas de preuves formelles, mais un ensemble de *faits* permettant de construire une éventuelle preuve formelle. Ainsi, si une preuve est certifiée en ACL2, alors on peut dire qu'il existe une preuve formelle et donc, que le théorème est valide.

3.2.2 Proposition de preuve

Les spécifications ACL2 sont regroupées en livres (*books*), qui sont des fichiers contenant des évènements (*events*). Les évènements sont des fonctions qui étendent la logique de base. Dans le cas de notre preuve sur la décidabilité de l'égalité sur les entiers naturels, le prouveur étant très automatisé, il réussit à prouver le lemme très vite, mais en utilisant le tiers exclu (d'après la trace d'exécution). Même en utilisant les différentes commandes *manuelles* du vérificateur, les heuristiques reconduisent systématiquement vers une preuve automatique classique. Pour contourner ce problème, nous avons donc défini une fonction décrivant l'algorithme de l'égalité, puis nous avons montré son équivalence avec l'égalité d'ACL2. Les évènements correspondants sont les suivants :

```
1   (defmacro naturalp (x)
2     (list 'and (list 'integerp x) (list '<= 0 x)))
3
4   (defun eqdec (n m)
5     (if (and (naturalp n) (naturalp m))
6       (if (= n 0) (= m 0)
7         (if (= m 0) nil (eqdec (- n 1) (- m 1)))) nil))
8
9   (defthm eq_nat_dec
10    (implies (and (naturalp n) (naturalp m))
11      (and (implies (eqdec n m) (= n m))
12           (implies (= n m) (eqdec n m))))
13    :rule-classes NIL :instructions (prove))
```

En lignes 1 et 2, on définit la macro `naturalp`, qui contraint un entier relatif (`integerp`) à être un entier naturel. Des lignes 4 à 7, on définit notre relation d'égalité `eqdec`, de manière usuelle, où `nil` correspond au $\perp$ propositionnel, et avec un appel récursif pour le cas successeur-successeur. Enfin, en lignes 9 à 13, il s'agit du théorème montrant l'équivalence de notre relation `eqdec` avec la relation d'égalité du système `=`. La preuve se situe en ligne 13, après le mot-clé `:instructions`, et il suffit, en effet, de déclencher le prouveur automatique avec `prove`, afin que les récurrences s'effectuent par rapport à la définition de notre algorithme d'égalité `eqdec`.

Cette suite d'évènements est évaluée par le toplevel interactif d'ACL2 et, en ce qui concerne la preuve, une trace très détaillée est disponible, nous indiquant les déductions effectuées. Le résultat des évaluations peut être consulté en annexe A, section A.4.1.

Une alternative à cette évaluation directe est de mettre ces évènements dans un livre et de le charger, par la suite, dans le toplevel. Dans notre cas, on aurait le livre suivant :

```
(in-package "ACL2")

(defmacro naturalp (x)
  (list 'and (list 'integerp x) (list '<= 0 x)))

(defun eqdec (n m)
  (if (and (naturalp n) (naturalp m))
    (if (= n 0) (= m 0)
      (if (= m 0) nil (eqdec (- n 1) (- m 1)))) nil))

(defthm eq_nat_dec
  (implies (and (naturalp n) (naturalp m))
    (and (implies (eqdec n m) (= n m))
         (implies (= n m) (eqdec n m))))
  :rule-classes NIL :instructions (prove))
```

```
16
17   (verify-guards eqdec)
```

Des lignes 3 à 15, ce sont exactement les mêmes évènements que dans la session à toplevel. Ce qui change, c'est la ligne 1, où l'on doit importer le *package* de base, à savoir `ACL2`, ainsi que la ligne 17, qui permet de vérifier les gardes de notre relation d'égalité `eqdec` (ce que nous n'avons pas fait précédemment). Les gardes sont des conditions que doivent vérifier les paramètres formels des fonctions. Une vérification des gardes consiste à regarder, dans la définition d'une fonction, si les gardes impliquent bien les gardes de toutes les fonctions appelées dans le corps de la fonction.

Ce livre peut être ensuite chargé dans le toplevel d'ACL2 par la commande `certify-book`. On trouvera le résultat de cette commande sur ce livre, en annexe A, section A.4.2.

3.2.3 Observations

Une première observation à propos d'ACL2 est que d'une certaine manière, les preuves sont lisibles, non pas tant dans le script en lui-même (il est, en effet, plutôt court du fait de l'automatisation importante) comme c'était le cas pour Mizar, mais dans les indications affichées par le toplevel lorsqu'il cherche un cheminement de preuve. Ces informations sont exprimées dans le langage naturel et sont donc, *a priori*, compréhensibles. On peut considérer que c'est un format de preuves, certes pas rigoureux, mais plus satisfaisant que dans certains prouveurs, où l'on se contente de dire si la preuve est correcte. Toutefois, comme on a pu le constater, ces traces sont très verbeuses et peuvent être aussi difficilement appréhendables, voire même, du point de vue du contenu, peu claires (pour un mathématicien, par exemple). L'utilisateur peut donc être amené à perdre du temps dans l'examen de ces informations, sachant qu'elles doivent être étudiées lorsque la procédure automatique de recherche de preuves échoue et que certaines parties sont très informelles.

Concernant la direction de preuve, on peut remarquer, dans les traces, que le prouveur raisonnne en backward, ce qui semble plutôt logique dans la mesure où c'est une procédure automatique. Par ailleurs, si le prouveur automatique échoue, il y a possibilité d'utiliser des commandes manuelles, qui ne sont pas sans rappeler les langages procéduraux, que nous avons étudiés dans le chapitre 2, et de fait, on retrouve, à nouveau, un raisonnement backward. Cependant, la philosophie des prouveurs à la Boyer-Moore consiste à utiliser le prouveur manuel non pas pour compléter la preuve, mais pour trouver des moyens d'aider le prouveur automatique à conclure. Ceci est fait par un systèmes d'indices (*hints*), qui sont des lemmes ou des stratégies permettant de *renforcer* la procédure automatique. De ce point de vue là, on peut considérer que la preuve est faite de manière non seulement déclarative, mais aussi en mode forward. La mixité entre modes forward et backward que nous avions pu observée avec Mizar est probablement plus exacerbée en ACL2.

Du fait de la grande automatisation et donc contrairement à Mizar, les preuves ont tendance à être faciles à écrire, du moment que le prouveur automatique est suffisamment puissant pour résoudre. Sinon, il faut utiliser les commandes du prouveur manuel, et ce, obligatoirement à toplevel (car procédural), ce qui n'est pas toujours aisé, puisqu'il faut comprendre, dans les traces, où a lieu l'échec.

Enfin, les exécutions, données en annexe, montrent que, même si notre preuve est petite, les performances sont très raisonnables, surtout si l'on considère tout ce qui est effectivement réalisé au regard des traces. Aucune différence notable n'a donc été observée par rapport à Mizar.

Chapitre 4

Langages de termes

Une alternative aux styles *traditionnels*, vus dans les chapitres 2 et 3, est d'utiliser des λ-termes comme preuves. Il s'agit de systèmes basés sur l'isomorphisme de Curry-Howard et plus généralement sur la sémantique de Heyting-Kolmogorov, dont nous ferons un bref rappel préliminaire. Il existe plusieurs prouveurs de ce type, comme par exemple, Coq ou Lego, mais ces derniers n'utilisent pas, comme on a pu le voir précédemment, exclusivement un langage de λ-termes pour exprimer les preuves, mais plutôt un langage de tactiques (procédural) qui donne une indication au système pour construire les preuves (donc les λ-termes). Le seul outil ayant un langage purement de λ-termes, est Alfa (basé sur ALF), que nous nous proposons d'étudier.

De manière annexe, on a pu voir, dans le chapitre 1, qu'AUTOMATH possédait aussi un langage de termes. Cependant, il ne s'agit pas de λ-termes, mais de symboles de fonctions correspondant à des règles de la logique, dont le nombre de prémisses fixe l'arité des fonctions. Ce langage est donc complètement isomorphe au système formel, et n'apporte pas de pouvoir d'expression supplémentaire. C'est pourquoi nous avons choisi de ne pas l'étudier. Toutefois, le lecteur intéressé pourra se référer à la réimplantation d'AUTOMATH, par Freek Wiedijk, qui permet de revérifier le codage du *Grundlagen der Analysis* d'Edmund Landau [50], par L. S. Jutting, ou le traitement de l'analyse réelle, par J. Sucker.

4.1 Heyting-Kolmogorov et Curry-Howard

4.1.1 Sémantique de Heyting-Kolmogorov

En 1931-32, Heyting et Kolmogorov proposent de donner une sémantique fonctionnelle aux démonstrations. Par exemple :

- une démonstration canonique de $A \Rightarrow B$ est une fonction qui, à toute démonstration de A, associe une démonstration de B,
- une démonstration canonique de $\forall x.A$ est une fonction qui, à tout objet t, associe une démonstration de $A[x \leftarrow t]$,
- une démonstration canonique de $A \wedge B$ est un couple formé d'une démonstration de A et d'une démonstration de B, ...

Les démonstrations des axiomes sont des objets donnés *a priori*. Ainsi, si on a un axiome $A \Rightarrow B$, la démonstration correspondante est une fonction f, qui associe une démonstration de B à toute démonstration de A. La nature des démonstrations des propositions atomiques importent peu, seul importe le fait que si a est une démonstration de A, alors $(f\ a)$ est une démonstration de B. De manière générale, la nature des démonstrations importe moins que la relation qui existe entre les démonstrations des différentes propositions.

4.1.2 Isomorphisme de Curry-Howard

Si d'après la sémantique de Heyting-Kolmogorov, on peut considérer les démonstrations comme des fonctions, une idée est donc d'utiliser les λ-termes pour les représenter. On peut aller plus loin en prenant un λ-calcul typé, dont l'ensemble des types est isomorphe à l'ensemble des propositions. De là, cela induit un deuxième isomorphisme entre les démonstrations et les λ-termes. C'est le principe de l'isomorphisme de Curry-Howard. En particulier, si Φ représente la bijection de l'ensemble des propositions vers l'ensemble des types, et φ, la bijection de l'ensemble des démonstrations vers l'ensemble des termes typables, alors si π est une preuve de $A_1, \ldots, A_n \vdash A$, alors $(\varphi\ \pi)$ est un terme de type $(\Phi\ A)$ dans le contexte $x_1 : (\Phi\ A_1), \ldots, x_n : (\Phi\ A_n)$.

Ainsi, plus le système de types est complexe, plus la logique dont on peut exprimer les démonstrations est complexe. Par exemple, avec le λ-calcul simplement typé, on peut exprimer les démonstrations de la logique propositionnelle minimale (propositions atomiques et l'implication). Si l'on souhaite passer à la logique minimale (logique du premier ordre avec quantificateur universel), on doit alors utiliser le λ-calcul avec types dépendants ($\lambda\Pi$-calcul), ...

4.1.3 Isomorphisme de Curry-Howard et logique intuitionniste

La sémantique de Heyting-Kolmogorov propose aussi d'exprimer les démonstrations de la disjonction et du quantificateur existentiel comme suit :

- une démonstration canonique de $A \vee B$ est une démonstration de A ou une démonstration de B
- une démonstration canonique de $\exists x.A$ est un couple formé d'un terme t et d'une démonstration de $A[x \leftarrow t]$

Cette sémantique est clairement intuitionniste, puisque l'on sait qu'en logique classique, du fait du tiers exclu, prouver $A \vee B$ ne revient pas *forcément* à prouver soit A soit B, et que pour prouver $\exists x.A$, on n'est pas *obligé* de trouver un témoin qui soit un objet du langage (théorème de Herbrand). Si l'on étend, par exemple, le $\lambda\Pi$-calcul pour prendre en compte ces nouvelles propositions ($\lambda 1$-calcul, avec en plus, la conjonction et l'absurde), l'isomorphisme de Curry-Howard ne sera valable que si le tiers exclu n'est pas introduit (sinon on n'a plus que de simples injections des propositions dans les types, et des preuves dans les λ-termes).

4.1.4 Les preuves en tant qu'objets du langage

Si on utilise les λ-termes comme termes de la logique (logiques d'ordre supérieur), alors on a une dualité du fait qu'un λ-terme est à la fois une preuve et un objet du langage. Ceci a pour conséquence directe que l'on peut raisonner sur les preuves. En réalité, il existe généralement plusieurs sortes (objets représentant intuitivement les types des types) dans les logiques d'ordre supérieur, et si, par exemple, on utilise le λ-calcul pour représenter les termes de la théorie des types simples, seuls les λ-termes ayant un type de type la sorte des propositions seront effectivement des preuves.

4.2 Alfa

4.2.1 Contexte

Alfa [18] est un outil d'aide à la preuve plutôt récent, puisqu'il date de 1996, et se définit comme le successeur d'ALF [66], dont la première implantation a été réalisée en 1993. Il est développé par le *Programming Logic Group* de l'Université de Technologie de Chalmers, à Göteborg. Alfa consiste essentiellement en une interface graphique puissante au vérificateur Agda, développé par Catarina Coquand et basé sur V3 de Thierry Coquand. Récemment, le système a été enrichi avec le *Grammatical Framework* (GF) [75] d'Aarne Ranta, qui permet de traduire les preuves en langage naturel. L'interface graphique en elle-même a été implantée, au moyen de *fudgets* [14], par Thomas Hallgren. De manière annexe, l'intégralité du code est en Haskell [45].

Alfa implante la théorie des types (monomorphes) de Martin-Löf [56] et utilise l'isomorphisme de Curry-Howard. Le λ-calcul correspondant est le λ1-calcul, vu précédemment, étendu avec les axiomes d'égalité, les axiomes de Peano et l'élimination des coupures d'égalité et de récurrence. Le mode d'édition de preuves consiste à *raffiner* un λ-terme incomplet, les *trous* étant représentés par des métavariables numérotées (de la forme `?n`, où `n` est un entier naturel) possédant un contexte (leur type). Une preuve est valide lorsqu'il n'y a plus de métavariables et si le type du λ-terme *coïncide* bien avec la proposition à montrer.

Il n'y a pas d'automatisation en Alfa, si bien que le terme doit être complété intégralement, *à la main*, par l'utilisateur, et l'interface graphique est là pour aider dans cette tâche. Du point de vue bibliothèque, il n'y en a pas encore de standard (un projet est en cours à ce sujet), et la version distribuée est très succincte.

4.2.2 Raffinement de notre preuve

On peut, tout de suite, remarquer le pouvoir calculatoire d'un système comme Alfa. En effet, contrairement à ACL2, où l'on devait écrire une fonction (relation) testant l'égalité entre deux entiers naturels, puis montrer que cette fonction était équivalente à l'égalité sur les entiers naturels, ici, il suffit de montrer le lemme, c'est-à-dire, du fait de l'isomorphisme de Curry-Howard, de donner directement la fonction correspondante d'ACL2. Les spécifications peuvent donc être exécutées, dès lors que les preuves peuvent être exécutées.

En Alfa, le lemme de décidabilité de l'égalité sur les entiers naturels, que nous appellerons *eq_nat_dec*, s'exprime comme suit :

$$eq\_nat\_dec \in \forall a \in Nat.\forall a' \in Nat.a == a' \vee \neg(a == a')$$

où Nat représente le type (inductif) des entiers naturels, et où le symbole $\in$ signifie "est de type"[1].

Le terme, que nous avons construit et qui correspond à la preuve de ce lemme, est le suivant :

$$\begin{array}{l}
\forall I(\lambda a \to natrec \\
\quad\quad (\lambda h \to \forall a' \in Nat.h == a' \vee \neg(h == a')) \\
\quad\quad (\forall I(\lambda a' \to natrec \\
\quad\quad\quad\quad (\lambda h \to 0 == h \vee \neg(0 == h)) \\
\quad\quad\quad\quad (\vee I1\ eqZero)
\end{array}$$

[1] Ce symbole a probablement été choisi pour rappeler le "appartient" de la théorie des ensembles, bien qu'il s'agisse ici d'une théorie des types.

6 $\quad\quad (\lambda n\ h \rightarrow\ \vee I2\ (\forall E\ nat\_discr1))$
7 $\quad\quad a'))$
8 $\quad (\lambda n\ h \rightarrow \forall I(\lambda a' \rightarrow natrec$
9 $\quad\quad (\lambda h' \rightarrow (n+1) == h' \vee \neg((n+1) == h'))$
10 $\quad\quad (\vee I2\ (\forall E\ nat\_discr2))$
11 $\quad\quad (\lambda n'\ h' \rightarrow\ \vee E$
12 $\quad\quad\quad (\forall E\ h)$
13 $\quad\quad\quad (\lambda a0 \rightarrow\ \vee I1\ (eqSucc\ a0))$
14 $\quad\quad\quad (\lambda b \rightarrow\ \vee I2\ (not\_eqSucc\ n\ n'\ b)))$
15 $\quad\quad a'))$
16 $\quad a)$

où $\forall I$ et $\forall E$ sont respectivement le constructeur et le destructeur du type (inductif[2]) produit $\forall$. $\vee I1$ et $\vee I2$ sont les deux contructeurs du type disjonction $\vee$ (cas gauche et cas droite). $natrec$ est le récurseur associé à Nat (prédicat à valeurs dans $Prop$), dont on peut donner, à titre indicatif, le type et les règles de réduction :

$$\begin{array}{l} natrec(C \in Nat \rightarrow Prop, bc \in C\ 0, \\ \quad ic \in (n \in Nat) \rightarrow C\ n \rightarrow C\ (succ\ n), m \in Nat) \in C\ m \end{array}$$

$$\begin{array}{l} natrec\ C\ bc\ ic\ 0 \equiv bc \\ natrec\ C\ bc\ ic\ (n+1) \equiv ic\ n\ (natrec\ C\ bc\ ic\ n) \end{array}$$

nat_discr1, nat_discr2 et not_eqSucc sont des lemmes triviaux sur les entiers naturels, que nous avons posés en axiomes (pour aller plus vite[3]), et qui ont les types suivants :

$$\begin{array}{l} nat\_discr1 \in \forall a \in Nat.\neg(0 == (a+1)) \\ nat\_discr2 \in \forall a \in Nat.\neg((a+1) = 0) \\ not\_eqSucc(a, b \in Nat, aeqb \in \neg(a == b)) \in \neg((a+1) == (b+1)) \end{array}$$

Concernant $eqZero$ et $eqSucc$, ce sont aussi des lemmes triviaux, déjà prouvés dans Alfa, et qui sont de la forme :

$$\begin{array}{l} eqZero \in 0 == 0 \\ eqSucc(a, b \in Nat, eqb \in (a == b)) \in (a+1) == (b+1) \end{array}$$

Nous pouvons maintenant décrire plus précisément la preuve. En ligne 1, on introduit le premier produit sur a, avec $\forall I$ en lui donnant un λ-terme qui abstrait a (on peut d'ailleurs donner un autre nom de variable). Ensuite, on effectue le raisonnement par récurrence sur a, ce qui revient à définir une fonction par récurrence avec $natrec$, qui, comme on l'a vu, prend quatre paramètres (le prédicat de récurrence, le cas de base, le cas récurrent et l'argument sur lequel la récurrence s'effectue). En ligne 2, il s'agit du prédicat de récurrence, c'est-à-dire, ici, le type du lemme initial, où a est abstrait par un λ. Des lignes 3 à 7, c'est le cas de base, où $a = 0$. Plus particulièrement, en ligne 3, on introduit le deuxième produit sur

[2] Ici, nous n'avons pas utilisé le produit dépendant primitif d'Alfa, mais plutôt un type inductif qui a l'avantage de posséder la syntaxe naturelle de la quantification universelle. En utilisant le produit dépendant primitif d'Alfa, le type du lemme précédent devient $(a,\ a' \in Nat) \rightarrow a == a' \vee \neg(a == a')$, et le terme preuve peut donc directement commencer par $\lambda a \rightarrow natrec\ \ldots$

[3] Dans notre schéma de preuve (voir la figure 1.2, dans le chapitre 1), ces lemmes sont triviaux et ne nécessitent pas d'être détaillés. D'ailleurs, dans tous les autres outils d'aide à la preuve, que nous avons vus jusqu'à maintenant, il n'a jamais fallu expliciter leurs preuves, qui étaient automatiques. On peut donc, ici, les poser en axiomes, sans pour autant fausser l'étude comparée de ces systèmes.

a', puis on effectue un raisonnment par cas sur a' (on utilise encore *natrec* et on ignorera l'hypothèse de récurrence). En ligne 4, c'est le prédicat de récurrence du raisonnement par cas (on abstrait a'). En ligne 5, c'est le cas où $a = 0$ et $a' = 0$, qui est résolu en choisissant le cas gauche avec $\vee I1$ et en utilisant $eqZero$. En ligne 6, on a $a' = (n+1)$ et on introduit le n, l'hypothèse de récurrence (inutilisée par la suite), pour choisir le cas droit avec $\vee I2$, puis appliquer nat_discr1 avec $\forall E$ (instantiation). En ligne 7, c'est l'argument du raisonnement par cas, à savoir a'. Des lignes 8 à 15, c'est le cas récurrent, où $a = (n + 1)$. En détails, en ligne 8, on introduit le n, l'hypothèse de récurrence sur a, ainsi que le second produit a', sur lequel on effectue le deuxième raisonnement par cas (toujours avec *natrec*). En ligne 9, c'est le prédicat de récurrence du raisonnement par cas de a' (il s'agit toujours d'abstraire a'). En ligne 10, on a $a = (n + 1)$ et $a' = 0$, qui est dual de la ligne 6 et qui se prouve en choisissant le cas droit ($\vee I2$), puis en appliquant (avec $\forall E$) nat_discr2 (symétrique de nat_discr1). Il reste le cas où $a = (n + 1)$ et $a' = (n' + 1)$. En ligne 11, on introduit n' et l'hypothèse de récurrence du deuxième raisonnement par cas (inutilisée, de même que précédemment). On raisonne alors par cas, avec $\vee E$, sur le $\vee$ de l'hypothèse de récurrence sur a (h), appliquée en ligne 12 à n' (avec $\forall E$). En ligne 13, si $n = n'$, alors on introduit l'égalité, on choisit le cas gauche (avec $\vee I1$) et on résoud trivialement en appliquant $eqSucc$. En ligne 13, si n et n' ne sont pas égaux, on introduit l'inégalité, on choisit la partie droite et on applique not_eqSucc pour conclure. En ligne 15, c'est l'argument du deuxième raisonnement par cas, c'est-à-dire a'. Enfin, en ligne 16, c'est l'argument du (vrai) raisonnement par récurrence, à savoir a.

Ce terme représentant la preuve de notre lemme a été construit progressivement au moyen de l'environnement interactif et, notamment grâce à une interface graphique (très ergonomique). On se reportera à la section A.5, dans l'annexe A, pour se donner une idée sur l'aspect de l'outil. Comme on l'a dit précédemment, on part donc d'un terme partiel, où les "trous" sont représentés par des métavariables, pour arriver à un terme clos (sans métavariables), dont le type correspond à celui du lemme à montrer. À tout moment, le terme partiel est typé et le système gère les dépendances entre métavariables. L'utilisateur peut réaliser sa preuve librement en cliquant simplement sur la métavariable qu'il souhaite instancier et pour laquelle le type est affiché (de manière générale, il suffit de cliquer sur n'importe quelle expression pour afficher le type). Dans certains cas, le typage systématique peut aider dans cette tâche, en inférant des instantiations. De plus, lorsque l'on cherche à instantier une métavariable, une fenêtre séparée fournit une liste (non exhaustive) d'expressions, avec un point vert pour celles qui sont compatibles avec le type attendu et un point rouge pour celles qui ne le sont pas. Par ailleurs, il y a plusieurs degrés de visibilité du terme, duquel on peut *cacher* certaines parties (typiquement, déjà prouvées et donc inintéressantes, puisque l'on cherche à se *concentrer* sur d'autres métavariables). Dans la figure de la section A.5, dans l'annexe A, on peut remarquer qu'ici, pour des raisons de place à l'affichage, nous avons caché le cas récurrent dans la récurrence sur n, qui est remplacé par des "...". Ces parties cachées peuvent être *révélées* et réapparaître à nouveau dans le terme. De manière similaire, il est possible d'afficher explicitement (resp. d'enlever) des signatures de types pour certaines expressions (par défaut, les signatures de types ne sont pas affichées) afin d'avoir un maximum d'informations (resp. un terme plus compact). Par exemple, dans la preuve précédente, on peut décider d'afficher le type de l'expression commençant par le premier *natrec*, et on aurait le terme suivant :

$$
\begin{aligned}
&\forall I(\lambda a \to [\forall a' \in Nat.a == a' \vee \neg(a == a')]\\
&\qquad natrec \ \ldots
\end{aligned}
$$

où l'expression entre [...] représente la signature de type.

4.2.3 Remarques

Une première remarque concerne la lisibilité de la preuve, c'est-à-dire du λ-terme. D'une certaine manière, elle est clairement meilleure que celle qu'on peut avoir avec des scripts procéduraux, puisque le λ-terme constitue la preuve proprement dite, qui, de fait, ne peut pas apporter plus de précisions. Possédant ainsi toutes les informations, on peut donc, *a priori*, lire la preuve en connaissant son état à chaque endroit, et c'est d'autant plus intéressant que le terme est compact. Toutefois, même si ce n'est pas réellement le cas avec notre preuve très courte, au fur et à mesure de la lecture, le contexte est enrichi et il devient difficile de le mémoriser. Pour pallier cela, l'interface graphique permet de cliquer sur des expressions du λ-terme, afin de connaître son type, et même, comme on a pu le voir, de rajouter, directement dans le terme, des signatures de types. Cependant, sans l'aide d'Alfa, le λ-terme n'est pas, en pratique, lisible, sachant que l'utilisateur n'est pas un vérificateur de types, et si l'on rajoute des signatures de types, la preuve perd fortement de sa compacité, puisque d'un seul bloc, ce qui nécessite d'avoir plusieurs niveaux de visibilité que seul le système est capable d'offrir.

Quant à la construction du terme, l'interface graphique fournit, à nouveau, une aide significative. En effet, le système est pourvu d'une assez puissante inférence de types des métavariables (qui permet certaines instantiations), d'une bonne gestion des contraintes entre métavariables, ainsi que d'une liste d'expressions, où celles, qui sont compatibles avec le type attendu d'une métavariable sélectionnée, sont mises en évidence. De ce point de vue là, les preuves sont faciles à écrire, à condition d'appréhender correctement l'isomorphisme de Curry-Howard, et, en particulier, la fonction d'injection (φ) des propositions vers les λ-termes[4]. Toutefois, on peut regretter l'absence d'automatisation, notamment lorsqu'il s'agit de propositions *triviales*, pour lesquelles on aimerait ne pas détailler la preuve. De ce fait, il devient difficile d'écrire de grandes preuves, car on doit aussi compléter les preuves de ces parties en question, ce qui rend la tâche très fastidieuse. Dans ces cas-là, on pourrait désirer un système de procédures de recherche de preuves, à la manière des stratégies ou tactiques des langages procéduraux.

Du point de vue de la maintenance, les preuves en Alfa ne sont pas du tout sensibles aux changements du système, s'il ne s'agit pas de modifier le vérificateur de types de manière non conservative (ce qui semble plutôt raisonnable). En effet, une fois le terme construit, c'est une donnée *immuable*, dont on doit uniquement vérifier le type pour être validée. Les évolutions des méthodes de construction (inférence de types des métavariables, gestion des contraintes, ...) n'ont donc aucune influence sur la validité des preuves. Par contre, les changements de spécification modifient obligatoirement les preuves en profondeur, et, contrairement à Coq, où l'on donne des *indications* pour construire le λ-terme, les preuves Alfa ne peuvent pas être validées sans effectuer les mises-à-jour correspondantes (d'autant plus importantes qu'il y a d'ajouts de signatures de types, censées améliorer la lisibilité).

Concernant le format de preuves, comme pour Coq (ainsi que Lego), et, de manière générale, comme pour tout outil d'aide à la preuve basé sur l'isomorhisme de Curry-Howard, la situation est complètement satisfaisante. En effet, le formalisme étant connu (théorie des types de Martin-Löf), les preuves peuvent être vérifiées de manière externe à Alfa, soit *à la main*, soit automatiquement en implantant un vérificateur de types pour la méta-théorie en question. On a donc un système ouvert, où l'utilisateur n'est pas tenu de faire confiance aux vérifications qui y sont faites.

Enfin, s'agissant des performances, l'interface graphique s'avère très gourmande en mé-

[4]En effet, si on veut introduire un $\forall$, il faut penser λ-abstraction, si on cherche à effectuer une coupure, il faut faire apparaître une application, si on souhaite faire un raisonnement par récurrence, il faut définir une fonction par récurrence, ...

moire, et la situation empire étrangement, non pas en fonction de la taille de la preuve, mais en fonction du temps d'utilisation[5].

[5]Cela pourrait laisser présager d'un problème de *garbage-collecting*.

Chapitre 5

Fusionner les trois mondes ?

Nous sommes maintenant en mesure de comparer, plus en avant, les trois styles de preuves, que nous avons répertoriés, à savoir procédural, déclaratif et de termes. Dans cette étude, nous verrons notamment si ces langages méritent d'être opposés et quels peuvent être leurs champs d'application les plus adéquates. De là, nous proposerons un nouveau langage, qui pourrait être amené à répondre aux exigences, ou plus exactement, aux attentes des trois communautés.

5.1 Étude comparée

Pour juger les trois styles de preuves que nous avons étudiés dans les chapitres précédents, rappelons, avant de faire un récapitulatif de ce qui a été vu précédemment, que nous avons essentiellement utilisé les critères suivants :

- Lisibilité
- Facilité d'écriture
- Orientation de la preuve (backward/forward)
- Maintenance (système et des spécifications)
- Mode de preuve (toplevel/batch)
- Performances
- Difficulté d'implantation

Dans le style procédural, de l'étude de PVS, HOL et Coq, on peut dégager des caractéristiques très générales. En effet, les scripts procéduraux ne sont pas raisonnablement lisibles, en ce sens que l'utilisateur ne peut pas systématiquement connaître l'état de la preuve. Ils sont, par contre, plutôt faciles à écrire, puisqu'il s'agit d'utiliser un système de règles (PVS) ou de tactiques (HOL, Coq), auxquelles on peut donner un minimum d'informations. Le reste est extrait du but à résoudre, car l'orientation de la preuve est complètement backward. Concernant la maintenance, les scripts sont peu robustes aux évolutions du système (changements dans des conventions de nommage, par exemple), mais beaucoup plus à des modifications dans les spécifications (les règles ou les tactiques, ne nécessitant que peu d'informations). Le mode d'édition de preuves se fait clairement à l'aide d'un toplevel (puisque l'utilisateur n'est pas toujours capable de prévoir l'état de preuve après l'application d'une règle ou d'une tactique), même si cela n'exclut pas de pouvoir utiliser un mode batch, une fois la preuve complétée (comme en Coq). Au niveau des performances, elles sont satisfaisantes (même pour les systèmes à forte automatisation comme PVS), puisque l'implantation est plutôt simple à réaliser (machine impérative à instructions et dirigée par le but).

Dans le style déclaratif, on a pu s'apercevoir que Mizar et ACL2 étaient fortement différents, dans la mesure où ACL2 est beaucoup plus automatisé que Mizar, si bien qu'on a été obligé de coder la relation testant l'égalité entre deux entiers naturels, pour ensuite montrer automatiquement qu'elle était équivalente à l'égalité. Les preuves automatiques étant la philosophie d'ACL2, il est clair que si l'on s'intéresse à l'expressivité des scripts, Mizar est bien plus représentatif du style déclaratif. Ainsi, on a pu remarquer que les preuves déclaratives étaient très lisibles et, de ce fait, très compréhensibles (plus proche d'une preuve mathématique exprimée en langage naturel). La succession de lemmes intermédiaires (c'est aussi valable pour ACL2, même si on ne l'a pas mis en œuvre) permet à l'utilisateur de savoir précisément ce qui est prouvé et comment conclure. Un inconvénient direct à cette bonne lisibilité est que les preuves sont un peu verbeuses (voir la preuve en Mizar) et l'utilisateur peut trouver fastidieux de devoir répéter beaucoup de propositions (à montrer). Concernant l'orientation de preuve, on est plutôt dans un style forward, même si ce n'est pas réellement figé, puisque certaines parties de preuves peuvent être *interprétées* comme étant dirigées de manière backward. Par rapport à la maintenance, les scripts sont peu sensibles aux évolutions du système, tant que l'on fait évoluer l'automatisation de manière conservative (voir le by de Mizar). Par contre, les changements dans les spécifications sont problématiques, car cela peut occasionner de nombreuses modifications dans les scripts, et il est difficile d'imaginer des outils d'édition spécifiques pouvant traiter tous les cas. Quant au mode de preuve, les scripts déclaratifs auront tendance à être écrits en mode batch, à condition de bien connaître l'automatisation du système. Pour les performances, cela dépend encore de l'automatisation. En effet, il semble assez clair qu'un langage déclaratif se doit de posséder une forte automatisation (ce n'est pas le cas de Mizar, où les preuves sont pleinement détaillées), afin de pouvoir combiner *naturellement* les lemmes intermédiaires posés. Dès lors, les performances peuvent fortement être entamées si cette procédure de recherche de preuves est puissante et donc coûteuse en temps (cela peut être éventuellement le cas en ACL2, dans certaines preuves, plus élaborées que notre exemple). Ceci est à relier directement à l'implantation, qui peut se révéler d'autant plus difficile que l'automatisation est complexe.

Enfin, pour les langages de termes, nous n'avons pas eu trop le choix, puisque seul Alfa utilise purement un langage de ce style. Nous avons pu remarquer qu'un λ-terme est finalement assez lisible (si aidé par des signatures de types placées de manière appropriées), même si cela demande un peu de pratique de l'isomorphisme de Curry-Howard, et même si cela peut devenir difficile dans le cas d'un terme de taille importante (surtout, parce que le terme est donné de manière monolithique). De même, on peut dire qu'écrire une preuve sous la forme d'un λ-terme est plutôt facile, à condition toujours d'être familier avec l'isomorphisme de Curry-Howard (de plus, en Alfa, en particulier, l'interface graphique fournit une aide supplémentaire non négligeable). Comme il s'agit de raffiner un terme (au départ réduit à une simple métavariable) dont on connaît le type, la preuve est donc essentiellement backward. À propos de la maintenance, les changements du système n'interfèrent pas avec le terme, puisqu'il ne s'agit pas d'instructions pour construire le terme (comme dans des systèmes comme Coq ou Lego), mais du terme en lui-même (ceci est valable si le vérificateur de types n'évolue pas de manière non conservative, ce qui semble raisonnable). Toutefois, les modifications de spécifications doivent presque systématiquement être répercutées dans les termes, même si ces derniers n'ont pas de signatures de types. De telles preuves peuvent être, *a priori*, écrites aussi bien à toplevel qu'en mode batch, même si l'expérience d'Alfa montre combien l'aide d'un toplevel peut être significative (affichage des types, inférences d'instantiations de métavariables, gestion des dépendances entre métavariables, ...). Quant aux performances, elles sont *théoriquement* bonnes (Alfa étant un peu particulier, car c'est l'interface graphique qui tend à affaiblir le système de ce point de vue là), puisque, sans automatisation, il ne s'agit que d'un vérificateur de types. Enfin, l'implantation peut se révéler complexe, suivant le niveau de conversion, que l'on souhaite, entre deux termes partiels.

On peut synthétiser la comparaison entre ces trois styles de preuves dans le tableau suivant :

	Procédural	Déclaratif	Terme
Lisibilité	Difficile	Naturelle	Possible
Écriture	Très bonne	Verbeux	Plutôt bonne
Orientation	Backward	Plutôt forward	Backward
Maintenance	Système	Théories	Théories
Mode	Toplevel	Plutôt batch	Toplevel/Batch
Performances	Bonnes	Moyennes	Bonnes
Implantation	Assez simple	Plutôt complexe	Peut être complexe

D'après les observations précédentes, nous sommes en mesure de dire où ces styles sont utiles et aussi quand il est intéressant de les utiliser. On utilisera les preuves procédurales pour de petites preuves simples et réalisées interactivement en backward, pour lesquelles on n'est pas intéressé par les détails formels. Ces preuves devront être vues comme des *boîtes noires*. Le style déclaratif sera utilisé pour des preuves plus complexes, que l'on veut construire dans un mode forward (comme des preuves mathématiques exprimées en langage naturel), plutôt en mode batch et très précisément[1], c'est-à-dire, avec beaucoup d'informations pour le lecteur. Enfin, les λ-termes peuvent aussi être utilisés pour des preuves complexes, mais en backward, construites, soit interactivement, soit en mode batch, et pour lesquelles on peut choisir le niveau de détails (mettre toutes les signatures de types, quelques-unes ou aucune). Ainsi, on peut remarquer que ces trois styles correspondent à des besoins spécifiques, et qu'il pourrait être intéressant de les fusionner pour profiter de leurs avantages dans tous les types de preuves. Dans cette optique, nous allons présenter un nouveau langage, que nous avons appelé $\mathcal{L}_{pdt}$ ("pdt" correspond aux initiales dénotant la fusion entre les trois mondes : "p" pour procédural, "d" pour déclaratif et "t" pour terme), qui tend à unifier ces trois types de langages de preuves.

5.2 Définition de $\mathcal{L}_{pdt}$

La syntaxe de $\mathcal{L}_{pdt}$ est donnée par la figure 5.1, présentée dans un style à la BNF et où <*proof*> est le point d'entrée. <*ident*> et <*int*> sont respectivement les entrées pour les *identificateurs* et les *entiers* (naturels). <*proc*> inclut une partie du langage procédural de Coq. Ici, nous avons simplifié l'entrée <*term*> des termes Coq. En particulier, la construction `Cases` a été réduite par rapport à la version du système et les formes `Fix`/`CoFix` ont été supprimées (en effet, nous ne les utiliserons pas directement dans les exemples et il sera suffisant de considérer les récurseurs uniquement comme des constantes, à la manière de la théorie des types de Martin-Löf).

5.3 Retour sur l'exemple

Avant de donner la sémantique formelle de $\mathcal{L}_{pdt}$, nous allons voir comment ce nouveau langage peut être utilisé et, en particulier, comment les trois styles de preuves se mélangent.

[1]Cela dépend du niveau d'automatisation. Une automatisation trop forte peut *casser* la lisibilité, même si on demande aux procédures de décision de donner des détails sur preuves générées, parce que les preuves automatiques ne sont, en général, pas minimales, et, de ce fait, pas aussi claires que des preuves construites manuellement.

```
<proof>        ::= (<proof-sen>)+

<proof-sen>    ::= <proof-top>.

<proof-top>    ::= Let [[?<int>]] <ident> : <term>
               |   Let <let-clauses>
               |   ?<int> [ : <term>] := <term>
               |   Proof
               |   Qed | Save

<let-clauses>  ::= [[?<int>]] <ident> [ : <term>] := <term>
                   (And [[?<int>]] <ident> [ : <term>] := <term>)*

<term>         ::= [<binders>]<term>
               |   <term> -> <term>
               |   (<ident>(, <ident>)* : <term>)<term>
               |   ((<term>)*)
               |   [<<term>>] Cases <term> of (<rules>)* end
               |   Set | Prop | Type
               |   <ident>
               |   ? | ?<int>
               |   <proc>
               |   <decl>

<binders>      ::= <ident> (,<ident>)* [ : <term>](; <binders>)*

<rules>        ::= [|] <pattern> => <term> (| <pattern> => <term>)*

<pattern>      ::= <ident> | ((<ident>)+)

<proc>         ::= <by <tac>>

<tac>          ::= Intro | Intro <ident>
               |   Intros (<ident>)*
               |   Clear (<ident>)+
               |   Cut <term>
               |   Apply <term>
               |   Pattern (<int>)* <term>
               |   <tactical>

<tactical>     ::= Idtac| Fail
               |   Try <tac>
               |   <tac> Orelse <tac>
               |   Progress <tac>
               |   <tac> ; <tac>
               |   <tac> ;[ <tac-seq> ]
               |   Repeat <tac>
               |   Do <int> <tac>
               |   First [ <tac-seq> ]
               |   Solve [ <tac-seq> ]

<tac-seq>      ::=
               |   <tac> (| <tac>)*

<decl>         ::= Let <let-clauses> In <term>
```

FIG. 5.1 – Syntaxe de $\mathcal{L}_{pdt}$.

Comme dans les chapitres précédents décrivant les langages de preuves, nous allons détailler la preuve de notre exemple récurrent, à savoir la décidabilité de l'égalité sur les entiers naturels :

```
1   Lemma eq_nat:(n,m:nat)n=m\/~n=m.
2   Proof.
3     Let b_n:(m:nat)(0)=m\/~(0)=m.
4     Proof.
5       ?1 := [m:nat]
6             <[m:nat](0)=m\/~(0)=m>
7             Cases m of
8             | O => <by Left;Auto>
9             | (S n) => <by Right;Auto>
10            end.
11    Qed.
12    Let i_n:(n:nat)((m:nat)n=m\/~n=m)->(m:nat)(S n)=m\/~(S n)=m.
13    Proof.
14      ?1 := [n:nat;Hrec;m:nat]
15            <[m:nat](S n)=m\/~(S n)=m>
16            Cases m of
17            | O => <by Right;Auto>
18            | (S n0) => ?2
19            end.
20      ?2 := <[_:n=n0\/~n=n0](S n)=(S n0)\/~(S n)=(S n0)>
21            Cases (Hrec n0) of
22            | (or_introl _) => <by Left;Auto>
23            | (or_intror _) => <by Right;Auto>
24            end.
25    Qed.
26    ?1 := (nat_ind ([n:nat](m:nat)n=m\/~n=m) b_n i_n).
27  Save.
```

Le principe du mode de preuves est similaire aux versions existantes de `Coq`, avec un ensemble de buts (à résoudre), constitués d'un contexte local (hypothèses) et d'une conclusion. La différence est qu'ici, tout but sera systématiquement lié à une métavariable numérotée, qui devra être instantiée. Ainsi, un but ne pourra être résolu que par instantiation de la métavariable qui lui est associée (voir, par exemple, les lignes 5, 14, 20 et 26). Lorsque l'on entre dans une nouvelle session de preuves, avec `Lemma` par exemple (ligne 1), le but à résoudre est initialement lié à la métavariable numéro 1.

Dans ce script, les parties procédurales sont clairement identifiées et parenthésées par `<by ...>` (lignes 8, 9, 17, 22 et 23). Dans ces parties, on peut utiliser tout le langage de tactiques habituel de `Coq` (voir [84]). Les parties déclaratives sont représentées par les deux constructions `Let` (lignes 3 et 12). Au même titre que `Lemma`, elles permettent d'ouvrir une nouvelle session de preuves (le but introduit est alors lié à la métavariable 1). La différence avec `Lemma` est qu'elles introduisent des objets non persistants, uniquement utilisables dans l'environnement où elles ont été déclarées. Pour faire apparaître cette distinction syntaxiquement, les preuves suivant `Let` sont parenthésées par `Proof ... Qed` (lignes 4 et 11, puis 13 et 25), tandis que les preuves suivant `Lemma` sont parenthésées par `Proof ... Save` (lignes 2 et 27). Enfin, les parties termes (preuves) sont utilisés exclusivement en partie droite des instantiations de métavariables (lignes 5, 14, 20 et 26). Dans ces termes, on peut directement utiliser d'autres traits déclaratifs avec des `Let ... In` (ce n'est pas le cas ici), ou des expressions procédurales (lignes 8, 9, 17, 22 et 23).

Pour faire cette preuve, on décide de faire la première récurrence (sur `n`) en mode plutôt forward. En effet, on commence par prouver le cas de base (`b_n`, ligne 3) et le cas récurrent (`i_n`, ligne 12) avec deux `Let`, puis on utilise ces deux résultats pour résoudre le but en instantiant avec le schéma d'élimination sur les entiers naturels `nat_ind` (ligne 26)[2].

Pour résoudre le cas de base (`b_n`), on réalise directement une instantiation (ligne 5). On introduit la variable `m` (avec `[...]`, ligne 5), puis on réalise une preuve par cas sur `m` (avec `Cases`, lignes 6 et 7). En ligne 6, entre les `<...>`, il s'agit du type du prédicat de `Cases` (qui permet de savoir le type de l'objet sur lequel on fait le filtrage, ainsi que le type rendu par le `Cases`)[3]. Si `m` est égal à `O` (ligne 8), alors la preuve est triviale et on décide de la faire de manière procédurale. On choisit d'abord le cas gauche avec `Left` pour obtenir `O=O` à montrer, puis on conclut avec `Auto`. Si `m` est de la forme `(S n)` (ligne 9), c'est complètement similaire, si ce n'est qu'il faut choisir la partie droite avec `Right` pour avoir `~(S n)=O` à prouver.

Dans le cas récurrent (`i_n`), on instantie aussi directement (ligne 14). On introduit la variable `n` (provenant du raisonnement par cas effectué sur `m`), l'hypothèse de récurrence `Hrec` (issue de la récurrence initiale réalisée sur `n`) et la variable `m` (avec `[...]`, ligne 14). On fait alors un raisonnement par cas sur `m` (avec `Cases`, lignes 15 et 16). Si `m` est égal à `O` (ligne 17), alors on considère que la preuve est triviale et qu'elle peut être résolue de manière procédurale. On choisit donc le cas de droite avec `Right` pour obtenir `~(S n)=O` et on conclut avec `Auto`. Si `m` est égal à `(S n0)` (ligne 18), on décide de retarder la preuve en mettant la métavariable `?2`. Cette métavariable est, de suite, instantiée (ligne 20) et on effectue un raisonnement par cas sur l'hypothèse de récurrence `Hrec` appliquée à `n0` (lignes 20 et 21). Le premier motif (ligne 22) correspond à `n=n0`, et on utilise un script procédural pour le prouver. On choisit le cas gauche avec `Left`, pour obtenir `(S n)=(S n0)` et on termine avec `Auto`. Dans le deuxième motif (ligne 23), on a `~n=n0`, et, le traitement est identique, excepté que l'on prend la partie droite pour montrer `~(S n)=(S n0)`.

La dernière partie du script (ligne 26) consiste, comme nous l'avons dit précédemment, à instantier la métavariable (numéro 1), correspondant au lemme que nous avons déclaré. L'instantiation consiste à effectuer la récurrence sur `n` au moyen du schéma d'élimination `nat_ind`. Ce schéma prend trois arguments : le prédicat de récurrence, ainsi que les preuves pour les cas de base et récurrent. Ces preuves ont été introduites par les `Let` (`i_n` et `b_n`, lignes 3 et 12) et sont donc directement référencées.

Même si on ne l'a pas utilisée dans cette preuve, il existe une deuxième sorte de métavariable, représentée par un `?` non numéroté explicitement (voir figure 5.1) et appelée argument implicite. Les arguments implicites permettent de ne pas expliciter certaines parties d'un terme, qui le seront ultérieurement lors de la phase de typage par unification. Ainsi, contrairement aux métavariables numérotées, les arguments implicites sont instantiés par le système et non par l'utilisateur. Les arguments implicites se révèlent très utiles pour définir partiellement des termes, comme des fonctions par exemple, et, nous le verrons, leur intérêt est d'autant plus renforcé que l'aspect terme, en tant que preuve, a été mis en valeur dans $\mathcal{L}_{pdt}$.

---

[2] Ces `Let` permettent d'introduire des lemmes qui n'ont, *a priori*, rien à voir avec le but à montrer. Toutefois, dans ce cas précis, le pas qui les sépare du but est trivial (on voit clairement que l'on fait une induction), si bien que, même si le sens de lecture de la preuve impose, sans aucun doute, un style forward, on peut, d'une certaine manière, considérer aussi cette partie de preuve comme étant backward.

[3] Ce type peut quelquefois être inféré, mais, pour cela, le système doit être capable de typer toutes les branches du `Cases`. Ici, ce n'est pas le cas, car, dans les branches, on a des scripts procéduraux qui n'apportent aucune information sur le type du terme construit (dont ils ont d'ailleurs besoin pour s'évaluer).

5.4 Sémantique

5.4.1 Préliminaires

Par rapport à la figure 5.1, nous appellerons *script de preuve* ou simplement *script*, toute expression de l'entrée <*proof*>. Une *phrase* sera une expression de l'entrée <*proof-sen*>. Nous nommerons *terme*, toute expression de l'entrée <*term*> et *terme pur*, toute expression de l'entrée <*term*>, privée des entrées <*proc*> et <*decl*>. On appellera *partie procédurale*, toute expression de l'entrée <*proc*> et *partie déclarative*, toute expression de l'entrée <*decl*>. Enfin, une *tactique* sera toute expression de l'entrée <*tac*> et un *tactical* sera toute expression de l'entrée <*tactical*>.

Nous opterons pour une sémantique naturelle (à grands pas). L'intérêt d'une telle sémantique est d'être très concrète et donc très proche d'une éventuelle implantation. Toutefois, ce type de sémantique ne peut traiter les scripts de preuves qui bouclent[4], mais ces scripts correspondent évidemment à des cas pathologiques et ne sont pas à considérer comme corrects. Cette sémantique est donc complètement adaptée et permet de rendre compte du côté calculatoire des scripts de preuves. De manière annexe, nous devrons expliciter les règles d'erreurs de cette sémantique car certaines règles réaliseront un traitement particulier des erreurs (comme les tacticals `Try` ou `Orelse`, par exemple). Cependant, afin de ne pas surcharger la présentation de la sémantique, les règles d'erreurs seront regroupées en annexe B.

Tout d'abord, donnons quelques définitions que nous utiliserons pour construire la sémantique.

Définition 5.4.1 (Environnement local, contexte) *Un environnement local ou contexte est une liste ordonnée d'hypothèses $(x : T)$, où x est un identificateur et T, un terme pur appelé type de x.*

On a $x \in \Gamma$, où x est un identificateur et Γ un contexte, si et seulement s'il existe une hypothèse $(x : T)$ dans Γ. On a $t \in \Gamma$, où t est un terme et Γ un contexte, si et seulement s'il existe une hypothèse $(x : T)$ dans Γ, telle que t soit un sous-terme de T.

Définition 5.4.2 (Environnement global) *Un environnement global est une liste ordonnée d'hypothèses et de définitions. Une hypothèse est de la forme $(x : T)$, où x est un identificateur et T, un terme pur appelé type de x. Une définition peut être inductive ou non. Une définition non inductive est de la forme $c := t : T$, où c est un identificateur appelé constante, t, un terme pur appelé corps de c et T, un terme pur appelé type de c ou de t. Une définition inductive est de la forme $\mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, où Γ_p, Γ_d et Γ_c sont des contextes représentant respectivement les paramètres, les types inductifs définis et les constructeurs.*

On a $x \in \Delta$, où x est un identificateur et Δ un environnement global, si et seulement s'il existe une hypothèse $(x : T)$ dans Δ ou une définition non inductive $x := t : T$ dans Δ ou une définition inductive $\mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$ dans Δ telle qu'il existe une hypothèse $(x : T)$ dans Γ_d ou Γ_c.

Définition 5.4.3 (But) *Un but est constitué d'un environnement global Δ, d'un environnement local Γ et d'un terme pur T (type). Il est noté $\Delta[\Gamma \Vdash T]$.*

On a $t \in \Delta[\Gamma \Vdash T]$, si et seulement si $t \in \Gamma$ ou t est un sous-terme de T.

Définition 5.4.4 (But indéfini) *Un but indéfini est constitué d'un environnement global Δ et d'un environnement local Γ. Il est noté $\Delta[\Gamma]$.*

[4]Nous verrons plus tard, notamment dans la sémantique des parties procédurales, que c'est possible.

On définit les *termes évalués* par l'entrée <*term-eval*> de la figure 5.2. L'ensemble des termes évalués sera noté $\mathcal{T}_\mathcal{E}$. On appelle *argument implicite*, tout terme de la forme `?`, ou tout terme évalué de la forme `?`$_{\mathtt{i}n}$, où n est un entier. L'ensemble des termes évalués qui sont arguments implicites sera noté $\mathcal{I}_\mathcal{E}$. On appelle *métavariable*, tout terme ou tout terme évalué de la forme `?`n, où n est un entier. Enfin, tout terme ou tout terme évalué de la forme `Set`, `Prop` ou `Type` sera appelé *sorte* et l'ensemble des sortes sera noté S.

```
<term-eval>  ::= [<binders>]<term-eval>
              |  <term-eval> -> <term-eval>
              |  (<ident>(, <ident>)* : <term-eval>)<term-eval>
              |  ((<term-eval>)*)
              |  [<<term-eval>>] Cases <term-eval> of (<rules>)* end
              |  Set | Prop | Type
              |  <ident>
              |  ?i<int> | ?<int>
```

FIG. 5.2 – Définition des termes évalués.

5.4.2 Sémantique des termes

Il s'agit de typer partiellement certains termes du CCI (termes purs) et d'interpréter certaines autres parties (procédurales ou déclaratives) qui indiquent comment construire un terme pur. En effet, le typage n'est potentiellement que partiel puisque les termes purs pourront contenir des métavariables.

Valeurs

Les valeurs de la sémantique des termes sont les suivantes :
- $(t, \Delta[\Gamma \Vdash T], m, \sigma)$, où $t \in \mathcal{T}_\mathcal{E}$, $\Delta[\Gamma \Vdash T]$ est un but, m est un ensemble de couples numéros de métavariables-buts $(i, \Delta[\Gamma_i \Vdash T_i])$, σ est une substitution de $\mathcal{I}_\mathcal{E}$ dans $\mathcal{T}_\mathcal{E}$, tel que $\nexists j$, $?j \in \Delta[\Gamma \Vdash T]$ et $?j \in \Delta[\Gamma_i \Vdash T_i]$.
- Erreur

Dans la suite, on désignera par $\mathcal{V}_\mathcal{T}$, l'ensemble des valeurs de la sémantique des termes.

Évaluation

Deux modes d'évaluation sont possibles suivant que l'on possède le type du terme ou non : le mode vérification ou le mode inférence de type. Initialement, on possédera le type du terme à évaluer (venant du but) et l'évaluation se fera alors en mode vérification, mais nous verrons que certains termes, comme l'application, ne peuvent s'évaluer (au moins partiellement) qu'en mode inférence.

Termes purs

Définition 5.4.5 (Évaluation des termes purs) *On dira que le terme pur t s'évalue* en mode inférence *en v, avec $v \in \mathcal{V}_\mathcal{T}$, dans le but indéfini $\Delta[\Gamma]$, que l'on notera $(t, \Delta[\Gamma]) \triangleright v$, si et seulement si $(t, \Delta[\Gamma]) \triangleright v$ est dérivable en utilisant les règles des figures 5.3, 5.4 et 5.5.*

On dira que le terme pur t s'évalue en mode vérification *en v, avec $v \in \mathcal{V}_{\mathcal{T}}$, dans le but $\Delta[\Gamma \Vdash T]$, que l'on notera $(t, \Delta[\Gamma \Vdash T]) \triangleright v$, si et seulement si $(t, \Delta[\Gamma \Vdash T]) \triangleright v$ est dérivable en utilisant les règles des figures 5.3, 5.4, et 5.5.*

Les fonctions utilisées dans les figures 5.3, 5.4 et 5.5, se définissent comme suit :

- $\mathsf{access}(x, \Delta[\Gamma])$: si $x \in \Gamma$, renvoie le type T de l'hypothèse $(x : T)$ le plus à droite dans Γ, sinon si $x \in \Delta$, renvoie le type T de l'hypothèse $(x : T)$, ou de la définition non inductive $c := t : T$, ou de l'hypothèse $(x : T)$ la plus à droite dans Γ_c ou Γ_d de la définition inductive $\mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, la plus à droite dans Δ, sinon renvoie Erreur.
- $\mathsf{new\_impl}(n)$: assure que l'entier n, numérotant un argument implicite, est unique.
- $\mathsf{unify}(T_1, T_2)$: renvoie un unificateur[5] entre T_1 et T_2 par rapport aux arguments implicites $?_{in}$.
- $\mathsf{induct\_info}(\Delta, T)$: vérifie que T est de la forme $(I\ p_1 \dots p_k\ a_1 \dots a_l)$, avec $I = \mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, $k = card(\Gamma_p)$, $l = card(\Gamma_d)$, et renvoie $(I, \{p_1, \dots, p_k\}, \{a_1, \dots, a_l\}, \{ta_1, \dots, ta_l\})$, avec $\{ta_1, \dots, ta_l\} = \{T_i | (x_i, T_i) \in \Gamma_d\}$.
- $\mathsf{elim}(s, I)$: si $I = \mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, $\Gamma_d = (x_1 : T_1) \dots (x_n : T_n) s_1$, avec $s_1 \in S$, vérifie si $s \in S$, si $(s_1, s) \in \{(Prop, Prop), (Set, Set|Type), (Type, Prop|Set|Type)\}$, et renvoie s.
- $\mathsf{abstract}(I, c, t)$: vérifie que c est de la forme $(C\ a_1 \dots a_n)$, où C est un constructeur de $I = \mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, et renvoie $[x_1 : T_1] \dots [x_n : T_n] t$, avec $(C, (x_1 : T_1) \dots (x_n : T_n)(I\ e_1 \dots e_m)) \in \Gamma_c$.
- $\mathsf{branch\_type}(\Delta, I, c, \{p_1, \dots, p_k\}, P)$: si $I = \mathsf{Ind}[\Gamma_p](\Gamma_d := \Gamma_c)$, $k = card(\Gamma_p)$, $l = card(\Gamma_d)$, $c = (C\ a_1 \dots a_n)$, $(C : (c_1 : C_1) \dots (c_n : C_n)(I\ p_1 \dots p_k\ e_1 \dots e_l)) \in \Gamma_c$, $P = [x_1 : T_1] \dots [x_l : T_l][i : (I\ g_1 \dots g_k\ x_1 \dots x_l)]Q$, renvoie $(c_1 : C_1) \dots (c_n : C_n) P[x_1 \backslash e_1; \dots; x_l \backslash e_l; i \backslash (C\ g_1 \dots g_k\ a_1 \dots a_n)]$.

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans les figures B.1, B.2, B.3 et B.4.

Exemple 5.4.6 (Évaluation d'un terme pur) *Comme exemple d'évaluation de terme pur, nous allons raffiner un terme de preuve partielle du lemme établissant qu'il existe un entier naturel n tel que $n = n$. Ce lemme s'exprime comme suit : $(ex\ nat\ [x : nat]x = x)$, où ex est le type de l'existentielle sur Set et dans Prop. Nous proposons de donner O comme témoin, mais pas la preuve que $O = O$, ce qui peut s'exprimer par le terme suivant à raffiner : $(ex\_intro\ ?\ [x]x = x\ O\ ?1)$. On suppose que l'on est dans l'environnement global Δ et dans le contexte vide. L'évaluation s'effectue comme suit :*

- *On évalue : $((ex\_intro\ ?\ [x]x = x\ O\ ?1), \Delta[\vdash (ex\ nat\ [x : nat]x = x)])$. On utilise la règle* $\mathsf{TAppSynt}$ *:*
- $((ex\ nat\ [x : nat]x = x), \Delta[]) \triangleright ((ex\ nat\ [x : nat]x = x), \Delta[Prop], \emptyset, \sigma_{id})$
- *On évalue $((ex\_intro\ ?\ [x]x = x\ O), \Delta[])$. On utilise la règle* UApp *:*
- *On évalue $((ex\_intro\ ?\ [x]x = x), \Delta[])$. On utilise la règle* UApp *:*
- *On évalue $((ex\_intro\ ?), \Delta[])$. On utilise la règle* $\mathsf{UApplmpl}$ *:*
- $(ex\_intro, \Delta[]) \triangleright$ $(ex\_intro, \Delta[\vdash (A : Set)(P : A\text{->}Prop)(x : A)\text{->}(P\ x)\text{->}(ex\ A\ P)], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- $(?, d[\vdash Set]) \triangleright (?_{i1}, d[\vdash Set], \emptyset, \sigma_{id})$ *(règle* Impl*).*
- *On obtient donc : $((ex\_intro\ ?), \Delta[]) \triangleright$ $((ex\_intro\ ?_{i1}), \Delta[\vdash (?i1 : Set)(P : ?i1\text{->}Prop)(x : ?_{i1})\text{->}(P\ x)\text{->}(ex\ ?i1\ P)], \emptyset, \sigma_{id})$.*

[5] Remarquons ici que l'évaluation est paramétrée par l'unification des termes purs, ce qui la rend plus générale. Lorsque nous décrirons l'implantation que nous avons réalisée, nous verrons quel algorithme a été choisi pour unifier.

- *On évalue* $([x]x = x, \Delta[])$. *On utilise la règle* $\lambda_{\mathsf{UCurry}}$ *:*
- *On évalue* $(x = x, \Delta[(x :?_{i2})])$*, c'est-à-dire* $((eq\ ?\ x\ x), \Delta[x :?_{i2})])$*. On utilise la règle* UApp *:*
- *On évalue* $((eq\ ?\ x), \Delta[x :?_{i2})])$*. On utilise la règle* UApp *:*
- *On évalue* $((eq\ ?), \Delta[(x :?_{i2})])$*. On utilise la règle* UAppSynt *:*
- $(eq, \Delta[(x :?_{i2})]) \triangleright (eq, d[(x :?_{i2}) \vdash (A : Set)A\text{-}{>}A\text{-}{>}Prop], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- $(?, \Delta[(x :?_{i2}) \vdash Set]) \triangleright (?_{i3}, d[(x :?_{i2}) \vdash Set], \emptyset, \sigma_{id})$ *(règle* Impl*).*
- *On obtient alors :* $((eq\ ?), \Delta[(x :?_{i2})]) \triangleright ((eq\ ?_{i3}), \Delta[(x :?_{i2}) \vdash ?_{i3}\text{-}{>}?_{i3}\text{-}{>}Prop], \emptyset, \sigma_{id})$
- $(x, \Delta[(x :?_{i2})]) \triangleright (x, d[(x :?_{i2}) \vdash ?_{i2}], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- *On unifie :* $\mathsf{unify}(?_{i3}, ?_{i2}) = \{?_{i3} \backslash ?_{i2}\} = \sigma_1$.
- *On obtient alors :* $((eq\ ?\ x), \Delta[(x :?_{i2})]) \triangleright ((eq\ ?_{i2}\ x), \Delta[(x :?_{i2}) \Vdash ?_{i2}\text{-}{>}Prop, \emptyset, \sigma_1)$
- $(x, \Delta[(x :?_{i2})]) \triangleright (x, \Delta[(x :?_{i2}) \Vdash ?_{i2}], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- *On unifie :* $\mathsf{unify}(?_{i2}, ?_{i2}) = \sigma_{id}$.
- *On obtient alors :* $((eq\ ?\ x\ x), \Delta[(x :?_{i2})]) \triangleright ((eq\ ?_{i2}\ xx), \Delta[(x :?i2) \Vdash Prop], \emptyset, \sigma_1)$.
- *On obtient alors :* $([x]x = x, \Delta[]) \triangleright ([x :?_{i2}]x = x, \Delta[\Vdash ?_{i2}\text{-}{>}Prop], \emptyset, \sigma_1)$.
- *On unifie :* $\mathsf{unify}(?_{i1}\text{-}{>}Prop, ?_{i2}\text{-}{>}Prop) = \{?_{i2} \backslash ?_{i1}\} = \sigma_2$*. On pose* $\sigma_3 = \sigma_1\sigma_2$.
- *On obtient alors :* $((ex\_intro\ ?\ [x]x = x), \Delta[]) \triangleright$
  $((ex\_intro\ ?_{i1}\ [x :?_{i1}]x = x),$
  $\Delta[\Vdash (x :?_{i1})\text{-}{>}([x :?_{i1}]x = x\ x)\text{-}{>}(ex\ ?_{i1}\ [x :?_{i1}]x = x)], \emptyset, \sigma_3)$.
- $(O, \Delta[]) \triangleright (O, d[\Vdash nat], , \sigma_3)$ *(règle* UVar*).*
- *On unifie :* $\mathsf{unify}(?_{i1}, nat) = \{?_{i1} \backslash nat\} = \sigma_4$*. On pose* $\sigma_5 = \sigma_3\sigma_4$.
- *On obtient alors :* $((ex\_intro\ ?\ [x]x = xO), \Delta[]) \triangleright$
  $((ex\_intro\ nat\ [x : nat]x = x\ O),$
  $\Delta[([x : nat]x = x\ O)\text{-}{>}(ex\ nat\ [x : nat]x = x)], \emptyset, \sigma_5)$.
- $(?1, \Delta[\Vdash ([x : nat]x = x\ O)]) \triangleright$
  $(?1, \Delta[\Vdash ([x : nat]x = x\ O)], \{(n, d[\Vdash ([x : nat]x = x\ O)])\}, \sigma_{id})$
- *On unifie :* $\mathsf{unify}((ex\ nat\ [x : nat]x = x), (ex\ nat\ [x : nat]x = x)) = \sigma_{id}$
- *On obtient le résultat final :*
  $((ex\_intro\ ?\ [x]x = x\ O\ ?1), \Delta[\Vdash (ex\ nat\ [x : nat]x = x)]) \triangleright$
  $((ex\_intro\ nat\ [x : nat]x = x\ O\ ?1),$
  $\Delta[\Vdash (ex\ nat\ [x : nat]x = x)], \{(n, \Delta[\Vdash ([x : nat]x = x\ O)])\}, \sigma_5)$

Cet exemple est intéressant car il montre comment sont résolus les arguments implicites (apparaissant dans les λ et dans les applications) et comment sont traitées les métavariables. On peut remarquer que la substitution du prédicat lié à la variable P a introduit un β-redex dans le type de la métavariable (nouveau but à montrer). Il n'y a pas de raisons d'effectuer une réduction à ce niveau-là et ce β-redex ne sera pas problématique puisque, lors d'une ultérieure unification, ce type sera réduit pour vérifier la compatibilité.

Parties déclaratives

Nous allons étendre la relation $\triangleright$ pour prendre en compte les parties déclaratives :

Définition 5.4.7 (Évaluation des termes purs et des parties déclaratives) *On dira que le terme pur ou la partie déclarative t s'évalue* en mode inférence *en v, avec $v \in \mathcal{V}_\mathcal{T}$, dans le but indéfini $\Delta[\Gamma]$, que l'on notera $(t, \Delta[\Gamma]) \triangleright v$, si et seulement si $(t, \Delta[\Gamma]) \triangleright v$ est dérivable en utilisant les règles des figures 5.3, 5.4, 5.5, 5.6 et 5.7.*

On dira que le terme pur ou la partie déclarative t s'évalue en mode vérification *en v, avec $v \in \mathcal{V}_\mathcal{T}$, dans le but $\Delta[\Gamma \Vdash T]$, que l'on notera $(t, \Delta[\Gamma \Vdash T]) \triangleright v$, si et seulement si $(t, \Delta[\Gamma \Vdash T]) \triangleright v$ est dérivable en utilisant les règles des figures 5.3, 5.4, 5.5, 5.6 et 5.7.*

$$\frac{s \in S}{(s, \Delta[\Gamma]) \triangleright (s, \Delta[\Gamma \Vdash Type], \emptyset, \sigma_{id})} \ (\mathsf{USort})$$

$$\frac{s \in S}{(s, \Delta[\Gamma \Vdash Type]) \triangleright (s, \Delta[\Gamma \Vdash Type], \emptyset, \sigma_{id})} \ (\mathsf{TSort})$$

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1) \qquad s_1 \in S \\ (T_2, \Delta[\Gamma_1, (x : T_3)]) \triangleright (T_4, \Delta[\Gamma_2, (x : T_5) \Vdash s_2], m_2, \sigma_2) \qquad s_2 \in S\end{array}}{((x : T_1)T_2, \Delta[\Gamma]) \triangleright ((x : T_5)T_4, \Delta[\Gamma_2 \Vdash s_2], m_1\sigma_2 \cup m_2, \sigma_1\sigma_2)} \ (\mathsf{UProd})$$

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_2], m_1, \sigma_1) \qquad s_2 \in S \\ (T_2, \Delta[\Gamma_1, (x : T_3) \Vdash s_1]) \triangleright (T_4, \Delta[\Gamma_2, (x : T_5) \Vdash s_1], m_2, \sigma_2) \qquad s_1 \in S\end{array}}{((x : T_1)T_2, \Delta[\Gamma \Vdash s_1]) \triangleright ((x : T_5)T_4, \Delta[\Gamma_2 \Vdash s_1], m_1\sigma_2 \cup m_2, \sigma_1\sigma_2)} \ (\mathsf{TProd})$$

$$\frac{T = \mathsf{access}(x, \Delta[\Gamma])}{(x, \Delta[\Gamma]) \triangleright (x, \Delta[\Gamma \Vdash T], \emptyset, \sigma_{id})} \ (\mathsf{UVar})$$

$$\frac{\begin{array}{c}(T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma_1) \qquad s \in S \\ T_2 = \mathsf{access}(x, \Delta[\Gamma]) \qquad \sigma = \mathsf{unify}(T_1, T_2\sigma_1)\end{array}}{(x, \Delta[\Gamma \Vdash T]) \triangleright (x, \Delta[\Gamma_1 \Vdash T_1]\sigma, m\sigma, \sigma)} \ (\mathsf{TVar})$$

$$\frac{\begin{array}{c}(T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma) \qquad s \in S \\ \mathsf{new\_impl}(n)\end{array}}{(?, \Delta[\Gamma \Vdash T]) \triangleright (?_{in}, \Delta[\Gamma_1 \Vdash T_1], m, \sigma)} \ (\mathsf{Impl})$$

$$\frac{\begin{array}{c}(T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma) \qquad s \in S \\ \forall i. ?i \notin \Gamma, T\end{array}}{(?n, \Delta[\Gamma \Vdash T]) \triangleright (?n, \Delta[\Gamma_1 \Vdash T_1], \{(n, \Delta[\Gamma_1 \Vdash T_1])\}, \sigma)} \ (\mathsf{Meta})$$

FIG. 5.3 – Raffinement des termes purs (1/3).

$$\dfrac{\begin{array}{c}(T,\Delta[\Gamma]) \triangleright (T_1,\Delta[\Gamma_1 \Vdash s],m_1,\sigma_1) \qquad s\in S\\ (t,\Delta[\Gamma_1,(x:T_1)]) \triangleright (t_1,\Delta[\Gamma_2,(x:T_2)\Vdash T_3],m_2,\sigma_2) \qquad x\notin\Delta[\Gamma]\end{array}}{([x:T]t,\Delta[\Gamma]) \triangleright ([x:T_2]t_1,\Delta[\Gamma_2 \Vdash (x:T_2)T_3],m_1\sigma_2\cup m_2,\sigma_1\sigma_2)}\,(\lambda_{\mathsf{UChurch}})$$

$$\dfrac{\begin{array}{c}((x:T_2)T_3,\Delta[\Gamma]) \triangleright ((x:T_4)T_5,\Delta[\Gamma_1 \Vdash s_1],m_1,\sigma_1) \qquad s_1\in S\\ (T_1,\Delta[\Gamma_1]) \triangleright (T_6,\Delta[\Gamma_2 \Vdash s_2],m_2,\sigma_2) \qquad s_2\in S\\ \sigma_3=\mathsf{unify}(T_2\sigma_2,T_6)\\ (t,\Delta[\Gamma_2,(x:T_6)\Vdash T_5\sigma_2]\sigma_3) \triangleright (t_1,\Delta[\Gamma_3,(x:T_7)\Vdash T_8],m_3,\sigma_4) \qquad x\notin\Delta[\Gamma]\end{array}}{\begin{array}{c}([x:T_1]t,\Delta[\Gamma \Vdash (x:T_2)T_3]) \triangleright\\ ([x:T_7]t_1,\Delta[\Gamma_3 \Vdash (x:T_7)T_8],(m_1\cup m_2)\sigma_3\sigma_4\cup m_3,\sigma_1\sigma_2\sigma_3\sigma_4)\end{array}}\,(\lambda_{\mathsf{TChurch}})$$

$$\dfrac{(t,\Delta[\Gamma,(x:?_{in})]) \triangleright (t_1,\Delta[\Gamma_1,(x:T_1)\Vdash T_2],m,\sigma) \qquad x\notin\Delta[\Gamma] \qquad \mathsf{new\_impl}(n)}{([x:?]t,\Delta[\Gamma]) \triangleright ([x:T_1]t_1,\Delta[\Gamma_1 \Vdash (x:T_1)T_2],m,\sigma)}\,(\lambda_{\mathsf{UCurry}})$$

$$\dfrac{\begin{array}{c}((x:T_1)T_2,\Delta[\Gamma]) \triangleright ((x:T_3)T_4,\Delta[\Gamma_1 \Vdash s],m_1,\sigma_1) \qquad s\in S\\ (t,\Delta[\Gamma_1,(x:T_3)\Vdash T_4]) \triangleright (t_1,\Delta[\Gamma_2,(x:T_5)\Vdash T_6],m_2,\sigma_2) \qquad x\notin\Delta[\Gamma]\end{array}}{([x:?]t,\Delta[\Gamma \Vdash (x:T_1)T_2]) \triangleright ([x:T_5]t_1,\Delta[\Gamma_2 \Vdash (x:T_5)T_6],m_1\sigma_2\cup m_2,\sigma_1\sigma_2)}\,(\lambda_{\mathsf{TCurry}})$$

$$\dfrac{\begin{array}{c}t_2\neq ?\ \text{et}\ \forall i.t_2\neq ?i\\ (t_1,\Delta[\Gamma]) \triangleright (t_3,\Delta[\Gamma_1 \Vdash (x:T_1)T_2],m_1,\sigma_1)\\ (t_2,\Delta[\Gamma_1]) \triangleright (t_4,\Delta[\Gamma_2 \Vdash T_3],m_2,\sigma_2)\\ \sigma_3=\mathsf{unify}(T_1\sigma_2,T_3)\end{array}}{((t_1\ t_2),\Delta[\Gamma]) \triangleright ((t_3\sigma_2\ t_4)\sigma_3,\Delta[\Gamma_2 \Vdash T_2\sigma_2[x\backslash t_4]\sigma_3],(m_1\sigma_2\cup m_2)\sigma_3,\sigma_1\sigma_2)\sigma_3}\,(\mathsf{UApp})$$

$$\dfrac{\begin{array}{c}t_2\neq ?\ \text{et}\ \forall i.t_2\neq ?i\\ (T_1,\Delta[\Gamma]) \triangleright (T_2,\Delta[\Gamma_1 \Vdash s],m_1,\sigma_1) \qquad s\in S\\ (t_1,\Delta[\Gamma_1]) \triangleright (t_3,\Delta[\Gamma_2 \Vdash (x:T_3)T_4],m_2,\sigma_2)\\ (t_2,\Delta[\Gamma_2]) \triangleright (t_4,\Delta[\Gamma_3 \Vdash T_5],m_3,\sigma_3)\\ \sigma_4=\mathsf{unify}(T_3\sigma_3,T_5)\\ \sigma_5=\mathsf{unify}(T_2\sigma_2\sigma_3\sigma_4,T_4\sigma_3[x\backslash t_4]\sigma_4)\end{array}}{\begin{array}{c}((t_1\ t_2),\Delta[\Gamma \Vdash T_1]) \triangleright\\ ((t_3\sigma_3\ t_4)\sigma_4\sigma_5,\Delta[\Gamma_3 \Vdash T_2\sigma_2\sigma_3]\sigma_4\sigma_5,((m_1\sigma_2\cup m_2)\sigma_3\cup m_3)\sigma_4\sigma_5,\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5)\end{array}}\,(\mathsf{TApp})$$

$$\dfrac{\begin{array}{c}t_2=?\ \text{ou}\ \exists i.t_2=?i\\ (t_1,\Delta[\Gamma]) \triangleright (t_3,\Delta[\Gamma_1 \Vdash (x:T_1)T_2],m_1,\sigma_1)\\ (t_2,\Delta[\Gamma_1 \Vdash T_1]) \triangleright (t_4,\Delta[\Gamma_2 \Vdash T_2],m_2,\sigma_2)\end{array}}{((t_1\ t_2),\Delta[\Gamma]) \triangleright ((t_3\sigma_2\ t_4),\Delta[\Gamma_2 \Vdash T_2\sigma_2[x\backslash t_4]],m_1\sigma_2\cup m_2,\sigma_1\sigma_2)}\,(\mathsf{UAppSynt})$$

$$\dfrac{\begin{array}{c}t_2=?\ \text{ou}\ \exists i.t_2=?i\\ (T,\Delta[\Gamma]) \triangleright (T_1,\Delta[\Gamma_1 \Vdash s],m_1,\sigma_1) \qquad s\in S\\ (t_1,\Delta[\Gamma_1]) \triangleright (t_3,\Delta[\Gamma_2 \Vdash (x:T_2)T_3],m_2,\sigma_2)\\ (t_2,\Delta[\Gamma_2 \Vdash T_2]) \triangleright (t_4,\Delta[\Gamma_3 \Vdash T_4],m_3,\sigma_3)\\ \sigma_4=\mathsf{unify}(T_1\sigma_2\sigma_3,T_3\sigma_3[x\backslash t_4])\end{array}}{\begin{array}{c}((t_1\ t_2),\Delta[\Gamma \Vdash T]) \triangleright\\ ((t_3\sigma_3\ t_4)\sigma_4,\Delta[\Gamma_3 \Vdash T_1\sigma_2\sigma_3]\sigma_4,((m_1\sigma_2\cup m_2)\sigma_3\cup m_3)\sigma_4,\sigma_1\sigma_2\sigma_3\sigma_4)\end{array}}\,(\mathsf{TAppSynt})$$

FIG. 5.4 – Raffinement des termes purs (2/3).

$$\frac{\begin{array}{c}(t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l)\to s)\sigma_2],m_2,\sigma_2)\\ s=\mathsf{elim}(I,s) \qquad P=(x_1:o1)\ldots(x_l:ol)(c:oi)P_b\\ ap_1=\mathsf{abstract}(I,c_1\sigma_{f1},t_1\sigma_{f1})\\ it_1=\mathsf{branch\_type}(\Delta,I,c_1,\{p_1,\ldots,p_k\}\sigma_{f1},P\sigma_{f1})\\ (ap_1,\Delta[\Gamma_{11}\Vdash it_1]) \triangleright (ap_{11},\Delta[\Gamma_{21}\Vdash it_{11}],m_{11},\sigma_{11})\\ \vdots\\ ap_n=\mathsf{abstract}(I,c_n\sigma_{fn},t_n\sigma_{fn})\\ it_n=\mathsf{branch\_type}(\Delta,I,c_2,\{p_1,\ldots,p_k\}\sigma_{fn},P\sigma_{fn})\\ (ap_n,\Delta[\Gamma_{n1}\Vdash it_n]) \triangleright (ap_{n1},\Delta[\Gamma_{(n+1)1}\Vdash it_{n1}],m_{n1},\sigma_{n1})\end{array}}{\begin{array}{c}(<P>Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end,\Delta[\Gamma]) \triangleright\\ ((<P>Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end)\sigma_{f(n+1)},\\ \Delta[\Gamma_{n1}\Vdash (P_b[x_1\backslash a_1;\ldots;x_l\backslash a_l;c\backslash t])\sigma_{f(n+1)}],\\ m_1\sigma_2\sigma_{11}\sigma_{m1}\cup m_2\sigma_{11}\sigma_{m1}\cup m_{11}\sigma_{m1}\cup\ldots\cup m_{n1}\sigma_{mn},\\ \sigma_1\sigma_2\sigma_{11}\sigma_{m1})\end{array}}\ (\mathsf{UCases})$$

où $\Gamma_{11}=\Gamma_2,\ \sigma_{fi}=\sigma_1\sigma_2\sigma_{11}\ldots\sigma_{(i-1)1}$ et $\sigma_{mi}=\sigma_{(i+1)1}\ldots\sigma_{n1}$

$$\frac{\begin{array}{c}(t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l)\to s)\sigma_2],m_2,\sigma_2)\\ s=\mathsf{elim}(I,s) \qquad P=(x_1:o1)\ldots(x_l:ol)(c:oi)P_b\\ ap_1=\mathsf{abstract}(I,c_1\sigma_{f1},t_1\sigma_{f1})\\ it_1=\mathsf{branch\_type}(\Delta,I,c_1,\{p_1,\ldots,p_k\}\sigma_{f1},P\sigma_{f1})\\ (ap_1,\Delta[\Gamma_{11}\Vdash it_1]) \triangleright (ap_{11},\Delta[\Gamma_{21}\Vdash it_{11}],m_{11},\sigma_{11})\\ \vdots\\ ap_n=\mathsf{abstract}(I,c_n\sigma_{fn},t_n\sigma_{fn})\\ it_n=\mathsf{branch\_type}(\Delta,I,c_2,\{p_1,\ldots,p_k\}\sigma_{fn},P\sigma_{fn})\\ (ap_n,\Delta[\Gamma_{n1}\Vdash it_n]) \triangleright (ap_{n1},\Delta[\Gamma_{(n+1)1}\Vdash it_{n1}],m_{n1},\sigma_{n1})\\ \sigma_p=\mathsf{unify}(T\sigma_{f(n+1)},(P_b[x_1\backslash a_1;\ldots;x_l\backslash a_l;c\backslash t]\sigma_{f(n+1)})\end{array}}{\begin{array}{c}(<P>Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end,\Delta[\Gamma\Vdash T]) \triangleright\\ ((<P>Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end)\sigma_{f(n+1)}\sigma_p,\\ \Delta[\Gamma_{n1}\Vdash (P_b[x_1\backslash a_1;\ldots;x_l\backslash a_l;c\backslash t])\sigma_{f(n+1)}\sigma_p],\\ m_1\sigma_2\sigma_{11}\sigma_{m1}\cup m_2\sigma_{11}\sigma_{m1}\cup m_{11}\sigma_{m1}\cup\ldots\cup m_{n1}\sigma_{mn},\\ \sigma_1\sigma_2\sigma_{11}\sigma_{m1}\sigma_p)\end{array}}\ (\mathsf{TCases})$$

où $\Gamma_{11}=\Gamma_2,\ \sigma_{fi}=\sigma_1\sigma_2\sigma_{11}\ldots\sigma_{(i-1)1}$ et $\sigma_{mi}=\sigma_{(i+1)1}\ldots\sigma_{n1}\sigma_p$

FIG. 5.5 – Raffinement des termes purs (3/3).

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans les figures B.1, B.2, B.3, B.4, B.5 et B.6.

Exemple 5.4.8 (Évaluation d'une partie déclarative) *Nous nous propossons d'évaluer l'expression* $Let\ x : Set := nat\ In\ [y : x](refl\_equal\ ?\ y)$*, où* $refl\_equal$ *est l'unique constructeur de l'égalité sur* Set*, dans le but* $\Delta[\Vdash (y : nat)y = y]$*. L'évaluation s'effectue selon les étapes suivantes :*

- *Au départ, on a* $(Let\ x : Set := nat\ In\ [y : x](refl\_equal\ ?\ y), \Delta[| - (y : nat)y = y])$*. On utilise la règle* TLetIn *:*
- $([y : nat]y = y, \Delta[]) \triangleright ([y : nat]y = y, \Delta[\Vdash Prop], \emptyset, \sigma_{id})$.
- $(Set, \Delta[]) \triangleright (Set, \Delta[\Vdash Type], \emptyset, \sigma_{id})$ *(règle* USort*).*
- $(nat, \Delta[\Vdash Set]) \triangleright (nat, \Delta[\Vdash Set], \emptyset, \sigma_{id})$ *(règle* TVar*).*
- *On évalue* $([y : x](refl\_equal\ ?\ y), \Delta[(x : Set)])$*. On utilise la règle* $\lambda_{\mathsf{UChurch}}$ *:*
- $(x, \Delta[(x : Set)]) \triangleright (x, \Delta[(x : Set) \Vdash Set], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- *On évalue* $((refl\_equal\ ?\ y), \Delta[(x : Set)(y : x)])$*. On utilise la règle* UApp *:*
- *On évalue* $((refl\_equal\ ?), \Delta[(x : Set)(y : x)])$*. On utilise la règle* UAppSynt *:*
- $(refl\_equal, \Delta x : Set)(y : x)]) \triangleright$
  $(refl\_equal, \Delta[(x : Set)(y : x) \Vdash (A : Set; x : A)x = x], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- $(?, \Delta[(x : Set)(y : x) \Vdash Set]) \triangleright (?_{i1}, \Delta[(x : Set)(y : x)| - Set], \emptyset, \sigma_{id})$ *(règle* Impl*).*
- *On obtient alors :* $((refl\_equal\ ?), \Delta[(x : Set)(y : x)]) \triangleright$
  $((refl\_equal\ ?_{i1}), \Delta[(x : Set)(y : x) \Vdash (x : ?_{i1})x = x], \emptyset, \sigma_{id})$.
- $(y, \Delta[(x : Set)(y : x)]) \triangleright (y, \Delta[(x : Set)(y : x) \Vdash x], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- *On unifie :* $\mathsf{unify}(?_{i1}, x) = \{?_{i1} \backslash x\} = \sigma_1$.
- *On obtient alors :* $((refl\_equal\ ?\ y), \Delta[(x : Set)(y : x)]) \triangleright$
  $((refl\_equal\ x\ y), \Delta[(x : Set)(y : x) \Vdash y = y], \emptyset, \sigma_1)$.
- *On obtient alors :* $([y : x](refl\_equal\ ?\ y), \Delta[(x : Set)]) \triangleright$
  $([y : x](refl\_equal\ x\ y), \Delta[(x : Set) \Vdash (y : x)y = y], \emptyset, \sigma_1)$.
- *On unifie :* $\mathsf{unify}((y : nat)y = y, (y : nat)y = y) = \sigma_{id}$.
- *On obtient le résultat final :*
  $(Let\ x : Set := nat\ In\ [y : x](refl\_equal\ ?\ y), \Delta[\Vdash (y : nat)y = y]) \triangleright$
  $(Let\ x : Set := nat\ In\ [y : x](refl\_equal\ x\ y), \Delta[bv(y : nat)y = y], \emptyset, \sigma_1)$.

Dans cet exemple, on remarque que, dans la règle TLetIn*, on ne peut pas évaluer* t_2 *en mode vérification avec le résultat de l'évaluation de* T_2*, à savoir* T_3*. En effet,* x *peut avoir des occurences dans* t_2*, mais aussi dans le type de* t_2 *(comme ici). Si on ne fait aucune subsitution dans* t_2*, on aurait, dans notre exemple, à unifier* $(y : nat)y = y$ *et* $(y : x)y = y$*, qui ne sont pas compatibles dans la mesure où l'unification se fait uniquement par rapport aux arguments implicites. La solution est donc de typer* t_2 *en mode inférence en obtenant le type* T_7*, puis d'unifier avec* T_3 *en effectuant la substitution dans* T_7*. Une autre solution pourrait être de typer* t_2 *en mode vérification avec* T_3 *mais en ayant au préalable substituer* x *dans* t_2*. Cependant, cette solution n'est pas très raisonnable, car il faudrait typer autant de fois le terme* t_3 *(résultat de l'évaluation de* t_1*) qu'il y a d'occurences de* x *dans* t_2*.*

Parties procédurales

Pour traiter tous les termes, il reste à inclure les parties procédurales dans la relation $\triangleright$. Pour ce faire, nous définissons d'abord l'évaluation des tactiques. Cette évaluation ne peut se faire qu'en mode vérification dans la mesure où la grande majorité des tactiques ne fournissent pas de termes purs mais des méthodes pour construire des termes purs à partir des buts qui leur sont donnés. On aura donc la définition suivante :

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1) \qquad s \in S \\ (t_1, \Delta[\Gamma_1 \Vdash T_2]) \triangleright (t_3, \Vdash [\Gamma_2 \Vdash T_3], m_2, \sigma_2) \\ (t_2, \Delta[\Gamma_2, (x : T_3)]) \triangleright (t_4, \Delta[\Gamma_3, (x : T_4) \Vdash T_5], m_3, \sigma_3)\end{array}}{\begin{array}{c}(Let\ x : T_1 := t_1\ In\ t_2, \Delta[\Gamma]) \triangleright \\ (([x : T_4]t_4\ t_3\sigma_3), \Delta[\Gamma_3 \Vdash T_5[x \backslash t_3\sigma_3]], (m_1\sigma_2 \cup m_2)\sigma_3 \cup m_3, \sigma_1\sigma_2\sigma_3)\end{array}} \text{(ULetIn)}$$

$$\frac{\begin{array}{c}(T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1) \qquad s_1 \in S \\ (T_1, \Delta[\Gamma_1]) \triangleright (T_4, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2) \qquad s_2 \in S \\ (t_1, \Delta[\Gamma_2 \Vdash T_4]) \triangleright (t_3, \Vdash [\Gamma_3 \Vdash T_5], m_3, \sigma_3) \\ (t_2, \Delta[\Gamma_3, (x : T_5)]) \triangleright (t_4, \Delta[\Gamma_4, (x : T_6) \Vdash T_7], m_4, \sigma_4) \\ \sigma_5 = \mathsf{unify}(T_3\sigma_2\sigma_3\sigma_4, T_7[x \backslash t_3\sigma_4])\end{array}}{\begin{array}{c}(Let\ x : T_1 := t_1\ In\ t_2, \Delta[\Gamma \Vdash T_2]) \triangleright \\ (([x : T_6]t_4\ t_3\sigma_4)\sigma_5, \Delta[\Gamma_4 \Vdash T_7[x \backslash t_3\sigma_4]]\sigma_5, \\ (((m_1\sigma_2 \cup m_2)\sigma_3 \cup m_3)\sigma_4 \cup m_4)\sigma_5, \sigma_1\sigma_2\sigma_3\sigma_4\sigma_5)\end{array}} \text{(TLetIn)}$$

FIG. 5.6 – Évaluation des parties déclaratives (1/2).

Définition 5.4.9 (Évaluation des tactiques) *On dira qu'une tactique t s'évalue en v, avec $v \in \mathcal{V}_\mathcal{T}$, dans le but $\Delta[\Gamma \Vdash T]$, que l'on notera $(t, \Delta[\Gamma \Vdash T]) \triangleright v$, si et seulement si $(t, \Delta[\Gamma \Vdash T]) \triangleright v$ est dérivable en utilisant les règles des figures 5.9, 5.10, 5.11 et 5.12.*

Les fonctions utilisées dans les figures 5.9, 5.10, 5.11 et 5.12, se définissent de la manière suivante :

- $\mathsf{new\_meta}(n_1, n_2, \ldots, n_p)$: assure que les entier $n_1, n_2, \ldots, n_p$, numérotant des méta-variables, sont uniques.
- $\mathsf{apply\_unify}(T, (x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n)T_{n+1}, \Delta[\Gamma])$: unifie T et T_{n+1} pour trouver les instantiations des variables $x_1, x_2, \ldots, x_n$. Cela s'effectue en plusieurs étapes :
  1. On unifie T et T_{n+1} par rapport aux variables $x_1, x_2, \ldots, x_n$. Si les deux types ne sont pas unifiables, on renvoie Erreur, sinon on obtient la substitution σ. On pose $\sigma_{nd} = \sigma_{id}$ et p, le nombre de variables dépendantes, c'est-à-dire, les variables x_i telles que x_i appartienne à un des termes $T_{i+1}, T_{i+2}, \ldots, T_{n+1}$.
  2. Considérons les variables $x_{d1}, x_{d2}, \ldots, x_{dm}$ du domaine de σ. On évalue les instantiations. On renvoie Erreur si une des évaluations s'évalue en Erreur, sinon on a les évaluations suivantes :

     $(x_{d1}\sigma_{t1}, \Delta[\Gamma\sigma_{nd}]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_{d1}], \emptyset, \sigma_{d1})$
     $(x_{d2}\sigma_{t2}, \Delta[\Gamma_1]) \triangleright (t_2, \Delta[\Gamma_2 \Vdash T_{d2}], \emptyset, \sigma_{d2})$
     $\vdots$
     $(x_{dm}\sigma_{tm}, \Delta[\Gamma_{m-1}]) \triangleright (t_m, \Delta[\Gamma_m \Vdash T_{dm}], \emptyset, \sigma_{dm})$

     où $\sigma_{ti} = \sigma\sigma_{d1}\sigma_{d2}\ldots\sigma_{d(i-1)}$. On obtient donc la substitution σ' de même domaine que σ et d'image $\{t_1\sigma_{s1}, t_2\sigma_{s2}, \ldots, t_m\sigma_{sm}\}$ avec $\{T_{d1}\sigma_{s1}, T_{d2}\sigma_{s2}, \ldots, T_{dm}\sigma_{sm}\}$, comme types, où $\sigma_{si} = \sigma_{d(i+1)}\sigma_{d(i+2)}\ldots\sigma_{dm}$.
  3. On cherche alors à résoudre le problème d'unification suivant :
     $x_{d1} = t_1\sigma_{s1}, x_{d2} = t_2\sigma_{s2}, \ldots, x_{dm} = t_m\sigma_{sm}$,
     $T_{d1}\sigma_{s1} = T_{r1}\sigma_{nd}\sigma_{s0}, T_{d2}\sigma_{s2} = T_{r2}\sigma_{nd}\sigma_{s0}, \ldots, T_{dm}\sigma_{sm} = T_{rm}\sigma_{nd}\sigma_{s0}$

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_{11}, \Delta[\Gamma_{11} \Vdash s_1], m_{11}, \sigma_{11}) \qquad s_1 \in S \\ (t_1, \Delta[\Gamma_{11} \Vdash T_{11}]) \triangleright (t_{11}, \Vdash [\Gamma_{12} \Vdash T_{12}], m_{12}, \sigma_{12}) \\ \vdots \\ (T_n, \Delta[\Gamma_{(n-1)2}]) \triangleright (T_{n1}, \Delta[\Gamma_{n1} \Vdash s_n], m_{n1}, \sigma_{n1}) \qquad s_n \in S \\ (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}]) \triangleright (t_{n1}, \Vdash [\Gamma_{n2} \Vdash T_{n2}], m_{n2}, \sigma_{n2}) \\ (t_{n+1}, \Delta[\Gamma_{n2}, (x_1 : T_{12}\sigma_{c2}), (x_2 : T_{22}\sigma_{c3}), \ldots, (x_n : T_{n2}\sigma_{c(n+1)})]) \triangleright \\ (t_{(n+1)1}, \Delta[\Gamma_{n+1}, (x_1 : T_{13}), (x_2 : T_{23}), \ldots, (x_n : T_{n3}) \Vdash T_{n+1}], m_{n+1}, \sigma_{n+1})\end{array}}{\begin{array}{c}(\mathit{Let}\ x_1 : T_1 := t_1\ \mathit{And}\ \ldots\ \mathit{And}\ x_n : T_n := t_n\ \mathit{In}\ t_{n+1}, \Delta[\Gamma]) \triangleright \\ (([x_1 : T_{13}; \ldots; x_n : T_{n3}]t_{(n+1)1}\ t_{11}\sigma_{t2}\ \ldots\ t_{n1}\sigma_{t(n+1)}), \\ \Delta[\Gamma_{n+1} \Vdash T_{n+1}[x_1 \backslash t_{11}\sigma_{t2}; \ldots; x_n \backslash t_{n1}\sigma_{t(n+1)}]], \\ m_{11}\sigma_{m2} \cup m_{12}\sigma_{t2} \cup m_{21}\sigma_{m3} \cup \ldots \cup \\ m_{n1}\sigma_{m(n+1)} \cup m_{n2}\sigma_{t(n+1)} \cup m_{n+1}, \sigma_{t1})\end{array}} \ (\mathsf{ULetsIn})$$

où $\sigma_{ci} = \sigma_{i1} \ldots \sigma_{n1}\sigma_{n2}, \sigma_{ti} = \sigma_{ci}\sigma_{n+1}$ et $\sigma_{mi} = \sigma_{(i-1)2}\sigma_{ti}$

$$\frac{\begin{array}{c}(T_{n+1}, \Delta[\Gamma]) \triangleright (T_{n+2}, \Delta[\Gamma_1 \Vdash s_{n+1}], m_1, \sigma_1) \qquad s_{n+1} \in S \\ (T_1, \Delta[\Gamma_1]) \triangleright (T_{11}, \Delta[\Gamma_{11} \Vdash s_1], m_{11}, \sigma_{11}) \qquad s_1 \in S \\ (t_1, \Delta[\Gamma_{11} \Vdash T_{11}]) \triangleright (t_{11}, \Vdash [\Gamma_{12} \Vdash T_{12}], m_{12}, \sigma_{12}) \\ \vdots \\ (T_n, \Delta[\Gamma_{(n-1)2}]) \triangleright (T_{n1}, \Delta[\Gamma_{n1} \Vdash s_n], m_{n1}, \sigma_{n1}) \qquad s_n \in S \\ (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}]) \triangleright (t_{n1}, \Vdash [\Gamma_{n2} \Vdash T_{n2}], m_{n2}, \sigma_{n2}) \\ (t_{n+1}, \Delta[\Gamma_{n2}, (x_1 : T_{12}\sigma_{c2}), (x_2 : T_{22}\sigma_{c3}), \ldots, (x_n : T_{n2}\sigma_{c(n+1)})]) \triangleright \\ (t_{(n+1)1}, \Delta[\Gamma_{n+1}, (x_1 : T_{13}), (x_2 : T_{23}), \ldots, (x_n : T_{n3}) \Vdash T_{n+3}], m_{n+1}, \sigma_{n+1}) \\ \sigma_{n+2} = \mathsf{unify}(T_{n+2}\sigma_{t1}, T_{n+3}[x_1 \backslash t_{11}\sigma_{t2}; \ldots; x_n \backslash t_{n1}\sigma_{t(n+1)}])\end{array}}{\begin{array}{c}(\mathit{Let}\ x_1 : T_1 := t_1\ \mathit{And}\ \ldots\ \mathit{And}\ x_n : T_n := t_n\ \mathit{In}\ t_{n+1}, \Delta[\Gamma \Vdash T_{n+1}]) \triangleright \\ (([x_1 : T_{13}; \ldots; x_n : T_{n3}]t_{(n+1)1}\ t_{11}\sigma_{t2}\ \ldots\ t_{n1}\sigma_{t(n+1)})\sigma_{n+2}, \\ \Delta[\Gamma_{n+1} \Vdash T_{n+3}[x_1 \backslash t_{11}\sigma_{t2}; \ldots; x_n \backslash t_{n1}\sigma_{t(n+1)}]]\sigma_{n+2}, \\ (m_1\sigma_{t1} \cup m_{11}\sigma_{m2} \cup m_{12}\sigma_{t2} \cup m_{21}\sigma_{m3} \cup \ldots \cup \\ m_{n1}\sigma_{m(n+1)} \cup m_{n2}\sigma_{t(n+1)} \cup m_{n+1})\sigma_{n+2}, \sigma_{t1}\sigma_{n+2})\end{array}} \ (\mathsf{TLetsIn})$$

où $\sigma_{ci} = \sigma_{i1} \ldots \sigma_{n1}\sigma_{n2}, \sigma_{ti} = \sigma_{ci}\sigma_{n+1}$ et $\sigma_{mi} = \sigma_{(i-1)2}\sigma_{ti}$

FIG. 5.7 – Évaluation des parties déclaratives (1/2).

où T_{ri} est le type T_j de la variable x_j telle que $x_j = x_{di}$. Si le problème n'a pas de solution, on renvoie Erreur, sinon on obtient une substitution σ''. Si $m = p$ alors on renvoie $(\{(x_i, t'_i, T'_i)\}, \sigma_{nd}\sigma_{s0})$ tel que :
- Si x_i est une variable dépendante, on donne $(x_{dj}, t_j\sigma_{sj}, T_{dj}\sigma_{sj})$ avec $x_i = x_{dj}$.
- Sinon on donne $(x_i, ?l, T_i\sigma'')$ avec new_meta(l).

Dans le cas contraire (où $m \neq p$), si σ'' et σ' ont même domaine, on renvoie Erreur, sinon on recommence au point 2 avec $\sigma = \sigma''$ et $\sigma_{nd} = \sigma_{nd}\sigma_{s0}$.

Exemple 5.4.10 (Utilisation de apply_unify) *Pour comprendre comment* apply_unify *fonctionne, considérons l'exemple où l'on cherche à unifier les types* $(lt\ O\ (S\ O))$ *et* $(k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k)$*, où gt et lt sont deux constantes sur les entiers naturels représentant respectivement les relations* $>$ *et* $<$*, dans le but indéfini* $\Delta[]$*. On associe, dans un premier temps, un nom de variable à tous les produits (même aux non-dépendants) en s'assurant que la variable est fraîche (à savoir qu'elle n'appartient pas à* Δ*), et on associe explicitement un type à chaque variable (on expanse le sucre syntaxique qui permet d'associer un type à plusieurs variables). On obtient donc le type :* $(k : nat; l : nat; x_1 : (gt\ k\ l))(lt\ l\ k)$*. Ensuite, on a les étapes suivantes :*

1. *L'unification par rapport à n, m et* x_1 *produit la substitution* $\sigma = \{k\backslash(S\ O); l\backslash O\}$*. On a* $\sigma_{nd} = \sigma_{id}$ *et* $p = 2$ *(2 variables dépendantes k et l).*
2. *Les variables du domaine de* σ *sont :* $x_{d1} = k$ *et* $x_{d2} = l$*. On a alors* $m = 2$*. Les instantiations correspondantes s'évaluent comme suit (nous ne donnons que le résultat et ne détaillons pas les dérivations) :*

   $((S\ O), \Delta[]) \triangleright ((S\ O), \Delta[\Vdash nat], \emptyset, \sigma_{id})$
   $(O, \Delta[]) \triangleright (O, \Delta[\Vdash nat], \emptyset, \sigma_{id})$

   On a alors $\sigma' = \sigma$ *(il n'y a pas d'implicites) et l'ensemble des types correspondant est* $\{nat, nat\}$*.*
3. *Le problème d'unification à résoudre est :* $k = (S\ O)$*,* $l = O$*,* $nat = nat$ *et* $nat = nat$*. C'est déjà une forme résolue et on obtient la substitution* $\sigma'' = \{k\backslash(S\ O); l\backslash O\}$*. On a* $m = p = 2$ *et on renvoie* $(\{(k, (S\ O), nat), (l, O, nat), (x_1, ?1, (gt\ (S\ O)\ O))\}, \sigma_{id})$*.*

On peut noter que le nommage du produit non-dépendant n'est nécessaire que pour évaluer apply_unify *et, en particulier, pour référencer le produit en question. En effet, dans la règle* Apply *correspondante (figure 5.10), on n'utilise pas les variables associées aux produits non-dépendants et on ne récupère que les métavariables correspondantes ainsi que leurs types.*

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans les figures B.8 et B.9.

Exemple 5.4.11 (Évaluation d'une partie procédurale) *Considérons le but* $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))]$ *que l'on cherche à résoudre avec la partie procédurale* $<$ `by Intro; Apply H; Assumption` $>$*. L'évaluation s'effectuera de la manière suivante :*
- *On évalue :* $(<$ `by Intro; Apply` H`; Assumption` $>$, $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))])$*. On utilise la règle* By *:*
- $((gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)), \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))]) \triangleright$ $((gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)), \Delta[(H : (k,l : nat)(gt\ k\ l)) \Vdash Prop], \emptyset, \sigma_{id})$.
- *On évalue :* (`Intro; Apply` H`; Assumption`, $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))])$*. On utilise la règle* Then *:*

- *On évalue :* $(\texttt{Intro};\texttt{Apply}\ H,$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))])$. *On utilise la règle* Then *:*
- $(\texttt{Intro}, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))]) \triangleright$
  $([x : (gt\ (S\ O)\ O)]?1, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash$
  $(gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))],$
  $\{\{(1, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash (lt\ O\ (S\ O)))])\}, \sigma_{id})$
  (règle AIntro*).*
- *On évalue* $(\texttt{Apply}\ H,$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k)(x : (gt\ (S\ O)\ O)) \Vdash (lt\ O\ (S\ O)))])$. *On utilise la règle* Apply *:*
- $(H, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O))]) \triangleright$
  $(H, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash$
  $(k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k)], \emptyset, \sigma_{id})$ *(règle* UVar*).*
- *On unifie :* $(\{(k,(S\ O),nat),(l,O,nat),(x_1,?1,(gt\ (S\ O)\ O))\},\sigma_{id}) =$
  $\mathsf{apply\_unify}((lt\ O\ (S\ O)),(k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k),$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O))])$ *(voir l'exemple 5.4.10*[6]*), et on a* $m = \{(2, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash (gt\ (S\ O)\ O)])\}$.
- *On obtient alors :* $(\texttt{Apply}\ H,$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k)(x : (gt\ (S\ O)\ O)) \Vdash (lt\ O\ (S\ O)))]) \triangleright$
  $((H\ (S\ O)\ O\ ?2), \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash$
  $((lt\ O\ (S\ O))], \{(2, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash$
  $(gt\ (S\ O)\ O)])\}, \sigma_{id}$.
- $(\texttt{Assumption},$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash (gt\ (S\ O)\ O)]) \triangleright$
  $(x, \Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O)) \Vdash (gt\ (S\ O)\ O)], \emptyset, \sigma_{id})$
  (règle Assumption*).*
- *On obtient alors :* $(\texttt{Intro};\texttt{Apply}\ H;\texttt{Assumption},$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))]) \triangleright$
  $([x : (gt\ (S\ O)\ O)](H\ (S\ O)\ O\ x),$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))], \emptyset, \sigma_{id})$.
- *Le résultat final est donc :* $(<\texttt{by Intro};\texttt{Apply}\ H;\texttt{Assumption}>,$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))]) \triangleright$
  $([x : (gt\ (S\ O)\ O)](H\ (S\ O)\ O\ x),$
  $\Delta[(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k) \Vdash (gt\ (S\ O)\ O)\text{->}(lt\ O\ (S\ O)))], \emptyset, \sigma_{id})$.

Nous avons donné la sémantique de quelques tactiques fréquemment utilisées (figures 5.9 et 5.10). La liste est loin d'être exhaustive et le lecteur intéressé pourra toujours se reporter au manuel de référence de Coq ([84]) pour une description plus informelle des autres tactiques proposées par le système. Par contre, concernant les tacticals, nous en avons décrit la totalité[7] (figures 5.11 et 5.12).

On peut maintenant donner l'évaluation complète des termes avec le traitement des parties procédurales :

Définition 5.4.12 (Évaluation des termes) *On dira que le terme t s'évalue* en mode inférence *en v, avec* $v \in \mathcal{V}_\mathcal{T}$*, dans le but indéfini* $\Delta[\Gamma]$*, que l'on notera* $(t,\Delta[\Gamma]) \triangleright v$*, si et*

[6] L'exécution est, en effet, similaire et il faut juste remplacer toutest les occurences du contexte vide par le contexte $(H : (k,l : nat)(gt\ k\ l)\text{->}(lt\ l\ k))(x : (gt\ (S\ O)\ O))$.

[7] Il manque cependant `Info` et `Abstract`, mais ce ne sont pas vraiment des tacticals (voir [84] pour plus détails).

seulement si $(t, \Delta[\Gamma]) \triangleright v$ *est dérivable en utilisant les règles des figures 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 et 5.12.*

On dira que le terme t s'évalue en mode vérification *en v, avec* $v \in \mathcal{V}_{\mathcal{T}}$*, dans le but* $\Delta[\Gamma \Vdash T]$*, que l'on notera* $(t, \Delta[\Gamma \Vdash T]) \triangleright v$*, si et seulement si* $(t, \Delta[\Gamma \Vdash T]) \triangleright v$ *est dérivable en utilisant les règles des figures 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 et 5.12.*

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans les figures B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8 et B.9.

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1) \qquad s \in S \\ (tac, \Delta[\Gamma_1 \Vdash T_2]) \triangleright (t, \Delta[\Gamma_2 \Vdash T_3], \emptyset, \sigma_2)\end{array}}{(<\texttt{by}\ tac>, \Delta[\Gamma \Vdash T_1]) \triangleright (t, \Delta[\Gamma_2 \Vdash T_3], m_1\sigma_2, \sigma_1\sigma_2)}\ (\mathsf{By})$$

FIG. 5.8 – Évaluation des parties procédurales.

5.4.3 Sémantique des phrases

Une phrase permet soit d'instantier une métavariable du terme preuve, soit d'ajouter un lemme dans le contexte d'une métavariable.

Valeurs

Les valeurs de la sémantique des phrases sont les suivantes :

- $(t, \Delta[\Gamma \Vdash T], m, p, d)$, où t est un terme pur, $\Delta[\Gamma \Vdash T]$ est un but, m est un ensemble de couples numéros de métavariables-buts $(i, \Delta[\Gamma_i \Vdash T_i])$, p est une pile de valeurs ($\lfloor\rfloor$ représente la pile vide et $\triangleleft$ est l'opérateur de concaténation des piles), $d \in \mathcal{D}$ avec $\mathcal{D} = \{\mathsf{Lemma}; \mathsf{Let}_i; \mathsf{Lemma}_b; \mathsf{Let}_{i,b}\}$, tel que $\nexists j$, $?j \in \Delta[\Gamma \Vdash T]$ et $?j \in \Delta[\Gamma_i \Vdash T_i]$.
- Erreur

On notera $\mathcal{V}_{\mathcal{P}}$, l'ensemble des valeurs de la sémantique des phrases.

Évaluation

On appellera *état*, tout n-uplet $(t, \Delta[\Gamma \Vdash T], m, p, d)$, où t est un terme pur, $\Delta[\Gamma \Vdash T]$ est un but, m est un ensemble de couples numéros de métavariables-buts $(i, \Delta[\Gamma_i \Vdash T_i])$, p est une pile de valeurs et $d \in \mathcal{D}$. La pile des valeurs servira à *ouvrir* de nouvelles sessions de preuves lorsqu'il s'agira d'évaluer un `Let` dont on ne fournit pas le terme preuve, et à stocker ainsi l'état de la session de preuve courante.

Définition 5.4.13 (Évaluation des phrases) *On dira que la phrase "p." s'évalue en v, avec* $v \in \mathcal{V}_{\mathcal{P}}$*, dans l'état* $(t, \Delta[\Gamma \Vdash T], m, p, d)$*, que l'on notera* $(p., t, \Delta[\Gamma \Vdash T], m, p, d) \twoheadrightarrow v$*, si et seulement si* $(p., t, \Delta[\Gamma \Vdash T], m, p, d) \twoheadrightarrow v$ *est dérivable en utilisant les règles des figures 5.13, 5.14 et 5.15.*

Les fonctions utilisées dans les figures 5.13, 5.14 et 5.15, se définissent comme suit :

- $\mathsf{begin}(d)$: Si $d = \mathsf{Lemma}$, renvoie Lemma_b, sinon si $d = \mathsf{Let}_i$, renvoie $\mathsf{Let}_{i,b}$, sinon renvoie Erreur.

$$\frac{T = (x : T_1)T_2 \qquad \mathsf{new\_meta}(n)}{(\texttt{Intro}, \Delta[\Gamma \Vdash T]) \triangleright ([x : T_1]?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma, (x : T_1) \Vdash T_2])\}, \sigma_{id})}\ (\mathsf{AIntro})$$

$$\frac{T = (x : T_1)T_2 \qquad x \notin \Delta[\Gamma] \qquad \mathsf{new\_meta}(n)}{(\texttt{Intro}\ x, \Delta[\Gamma \Vdash T]) \triangleright ([x : T_1]?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma, (x : T_1) \Vdash T_2])\}, \sigma_{id})}\ (\mathsf{NIntro})$$

$$\frac{T = (x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n)T_{n+1} \qquad T_{n+1} \neq (x_{n+1} : T_{n+2})T_{n+3} \qquad \mathsf{new\_meta}(n)}{\begin{array}{c}(\texttt{Intros}, \Delta[\Gamma \Vdash T]) \triangleright ([x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n]?n, \Delta[\Gamma \Vdash T], \\ \{(n, \Delta[\Gamma, (x_1 : T_1), (x_2 : T_2), \ldots, (x_n : T_n) \Vdash T_{n+1}])\}, \sigma_{id})\end{array}}\ (\mathsf{AIntros})$$

$$\frac{\begin{array}{c}T = (x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n)T_{n+1} \qquad T_{n+1} \neq (x_{n+1} : T_{n+2})T_{n+3} \\ x_1, x_2, \ldots, x_n \notin \Delta[\Gamma] \qquad \mathsf{new\_meta}(n)\end{array}}{\begin{array}{c}(\texttt{Intros}\ x_1\ x_2\ \ldots\ x_n, \Delta[\Gamma \Vdash T]) \triangleright ([x_1 : T_1; x_2 : T_2; \ldots; x_n : T_n]?n, \Delta[\Gamma \Vdash T], \\ \{(n, \Delta[\Gamma, (x_1 : T_1), (x_2 : T_2), \ldots, (x_n : T_n) \Vdash T_{n+1}])\}, \sigma_{id})\end{array}}\ (\mathsf{NIntros})$$

$$\frac{\Gamma = \Gamma_1(id : T)\Gamma_2 \qquad \forall (x_i : T_i) \in \Gamma_1 . id \notin T_i \qquad id \notin T}{(\texttt{Clear}\ id, \Delta[\Gamma \Vdash T]) \triangleright (?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma \backslash id \Vdash T])\}, \sigma_{id})}\ (\mathsf{ClearHyp})$$

$$\frac{\begin{array}{c}(\texttt{Clear}\ id_1, \Delta[\Gamma \Vdash T]) \triangleright (?n_1, \Delta[\Gamma \Vdash T], \{(n_1, \Delta[\Gamma_1 \Vdash T])\}, \sigma_{id}) \\ \vdots \\ (\texttt{Clear}\ id_n, \Delta[\Gamma_{m-1} \Vdash T]) \triangleright (?n_m, \Delta[\Gamma_{m-1} \Vdash T], \{(n_m, \Delta[\Gamma_m \Vdash T])\}, \sigma_{id})\end{array}}{(\texttt{Clear}\ id_1\ \ldots\ id_m, \Delta[\Gamma \Vdash T]) \triangleright (?n_m, \Delta[\Gamma \Vdash T], \{(n_m, \Delta[\Gamma_m \Vdash T_2])\}, \sigma_{id})}\ (\mathsf{ClearHyps})$$

Fig. 5.9 – Évaluation de quelques tactiques (1/2).

$$\frac{\exists(x_i,T_i)\in\Gamma.\sigma=\mathsf{unify}(T,T_i)}{(\texttt{Assumption},\Delta[\Gamma\Vdash T])\triangleright(x_i,\Delta[\Gamma\Vdash T]\sigma,\emptyset,\sigma)}\ (\mathsf{Assumption})$$

$$\frac{\begin{array}{c}(T_1,\Delta[\Gamma])\triangleright(T_3,\Delta[\Gamma_1\Vdash s],\emptyset,\sigma_1)\qquad s\in S\\ \mathsf{new\_meta}(n_1,n_2)\end{array}}{\begin{array}{c}(\texttt{Cut }T_1,\Delta[\Gamma\Vdash T_2])\triangleright((?n_1\ ?n_2),\Delta[\Gamma_1\Vdash T_2\sigma_1],\\ \{(n_1,\Delta[\Gamma_1\Vdash(x:T_3)T_2\sigma_1]),(n_2,\Delta[\Gamma_1\Vdash T_3])\},\sigma_1)\end{array}}\ (\mathsf{Cut})$$

$$\frac{\begin{array}{c}(t,\Delta[\Gamma])\triangleright(t_{n+1},\Delta[\Gamma_1\Vdash(x_1:T_1;x_2:T_2;\ldots;x_n:T_n)T_{n+1}],\emptyset,\sigma_1)\\ T,T_{n+1}\neq(x_{n+1}:T_{n+2})T_{n+3}\\ (\{(x_1,t_1,T_{11}),(x_2,t_2,T_{21}),\ldots,(x_n,t_n,T_{n1})\},\sigma_2)=\\ \mathsf{apply\_unify}(T\sigma_1,(x_1:T_1;x_2:T_2;\ldots;x_n:T_n)T_{n+1},\Delta[\Gamma_1])\\ m=\{(n,\Delta[\Gamma_1\sigma_2\Vdash T])|(x_i,?_{sn},T)\in\\ \{(x_1,t_1,T_{11}),(x_2,t_2,T_{21}),\ldots,(x_n,t_n,T_{n1})\}\}\end{array}}{(\texttt{Apply }t,\Delta[\Gamma\Vdash T])\triangleright((t_{n+1}\sigma_2\ t_1\ t_2\ \ldots\ t_n),\Delta[\Gamma_1\Vdash T\sigma_1]\sigma_2,m,\sigma_1\sigma_2)}\ (\mathsf{Apply})$$

$$\frac{\begin{array}{c}(t,\Delta[\Gamma])\triangleright(t_1,\Delta[\Gamma_1\Vdash T_1],\emptyset,\sigma_1)\\ \mathsf{new}_s(n)\end{array}}{\begin{array}{c}(\texttt{Pattern }t,\Delta[\Gamma\Vdash T])\triangleright(?n,\Delta[\Gamma_1\Vdash T\sigma_1],\\ \{(n,\Delta[\Gamma\Vdash([x:T_1]T[t\backslash x]\sigma_1\ t_1)])\},\sigma_1)\end{array}}\ (\mathsf{PatternAll})$$

$$\frac{\begin{array}{c}(t,\Delta[\Gamma])\triangleright(t_1,\Delta[\Gamma_1\Vdash T_1],\emptyset,\sigma_1)\\ \forall i.T_{|n_i}=t,\ i=1\ldots m\qquad \mathsf{new}_s(n)\end{array}}{\begin{array}{c}(\texttt{Pattern }n_1\ n_2\ \ldots\ n_m\ t,\Delta[\Gamma\Vdash T])\triangleright(?n,\Delta[\Gamma_1\Vdash T\sigma_1],\\ \{(n,\Delta[\Gamma\Vdash([x:T_1]T[t\backslash x]_{\{n_1,n_2,\ldots,n_m\}}\sigma_1\ t_1)])\},\sigma_1)\end{array}}\ (\mathsf{PatternOcc})$$

FIG. 5.10 – Évaluation de quelques tactiques (2/2).

$$\frac{\mathsf{new\_meta}(n)}{(\mathtt{Idtac}, \Delta[\Gamma \Vdash T]) \triangleright (?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma \Vdash T])\}, \sigma_{id})}\ (\mathsf{Idtac})$$

$$\frac{}{(\mathtt{Fail}, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur}}\ (\mathsf{Fail})$$

$$\frac{(tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}{(\mathtt{Try}\ tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}\ (\mathsf{Try\text{-}1})$$

$$\frac{(tac, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur} \qquad \mathsf{new\_meta}(n)}{(\mathtt{Try}\ tac, \Delta[\Gamma \Vdash T]) \triangleright (?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma \Vdash T])\}, \sigma_{id})}\ (\mathsf{Try\text{-}2})$$

$$\frac{\begin{array}{c}(tac_1, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1) \\ \Gamma \Vdash T \neq \Gamma_1 \Vdash T_1\end{array}}{(tac_1\ \mathtt{Orelse}\ tac_2, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}\ (\mathsf{Orelse\text{-}1})$$

$$\frac{\begin{array}{c}(tac_1, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur} \\ \text{ou } (tac_1, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1),\ \Gamma \Vdash T = \Gamma_1 \Vdash T_1 \\ (tac_2, \Delta[\Gamma \Vdash T]) \triangleright (t_2, \Delta[\Gamma_2 \Vdash T_2], m_2, \sigma_2)\end{array}}{(tac_1\ \mathtt{Orelse}\ tac_2, \Delta[\Gamma \Vdash T]) \triangleright (t_2, \Delta[\Gamma_2 \Vdash T_2], m_2, \sigma_2)}\ (\mathsf{Orelse\text{-}2})$$

$$\frac{\begin{array}{c}(tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1) \\ \Gamma \Vdash T \neq \Gamma_1 \Vdash T_1\end{array}}{(\mathtt{Progress}\ tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}\ (\mathsf{Progress})$$

FIG. 5.11 – Évaluation des tacticals (1/2).

$$\frac{\begin{array}{c}(tac_1, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1],\\ \{(n_1, \Delta[\Gamma_{11} \Vdash T_{11}]), \ldots, (n_p, \Delta[\Gamma_{p1} \Vdash T_{p1}])\}, \sigma_1)\\ (tac_2, \Delta[\Gamma_{11} \Vdash T_{11}]\sigma_{s1}) \triangleright (t_{11}, \Delta[\Gamma_{12} \Vdash T_{12}], m_{11}, \sigma_{11})\\ \vdots \\ (tac_2, \Delta[\Gamma_{p1} \Vdash T_{p1}]\sigma_{sp}) \triangleright (t_{p1}, \Delta[\Gamma_{p2} \Vdash T_{p2}], m_{p1}, \sigma_{p1})\end{array}}{\begin{array}{c}(tac_1;tac_2, \Delta[\Gamma \Vdash T]) \triangleright (t_1\sigma_{s(m+1)}[?n_1\backslash t_{11}\sigma_{t1}; \ldots; ?n_p\backslash t_{p1}\sigma_{tp}],\\ m_{11}\sigma_{t1} \cup \ldots \cup m_{p1}\sigma_{tp}, \sigma_1\sigma_{s(p+1)})\end{array}} \text{(Then)}$$

où $\sigma_{si} = \sigma_{11} \ldots \sigma_{(i-1)1}$ et $\sigma_{ti} = \sigma_{(i+1)1}\sigma_{(i+2)1} \ldots \sigma_{p1}$

$$\frac{\begin{array}{c}(tac_0, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1],\\ \{(n_1, \Delta[\Gamma_{11} \Vdash T_{11}]), \ldots, (n_p, \Delta[\Gamma_{p1} \Vdash T_{p1}])\}, \sigma_1)\\ (tac_1, \Delta[\Gamma_{11} \Vdash T_{11}]\sigma_{s1}) \triangleright (t_{11}, \Delta[\Gamma_{12} \Vdash T_{12}], m_{11}, \sigma_{11})\\ \vdots \\ (tac_p, \Delta[\Gamma_{p1} \Vdash T_{p1}]\sigma_{sp}) \triangleright (t_{p1}, \Delta[\Gamma_{p2} \Vdash T_{p2}], m_{p1}, \sigma_{p1})\end{array}}{\begin{array}{c}(tac_0; [tac_1 | \ldots | tac_p], \Delta[\Gamma \Vdash T]) \triangleright\\ (t_1\sigma_{s(m+1)}[?n_1\backslash t_{11}\sigma_{t1}; \ldots; ?n_p\backslash t_{p1}\sigma_{tp}],\\ m_{11}\sigma_{t1} \cup \ldots \cup m_{p1}\sigma_{tp}, \sigma_1\sigma_{s(p+1)})\end{array}} \text{(ThenS)}$$

où $\sigma_{si} = \sigma_{11} \ldots \sigma_{(i-1)1}$ et $\sigma_{ti} = \sigma_{(i+1)1}\sigma_{(i+2)1} \ldots \sigma_{p1}$

$$\frac{(\texttt{Try}\ (tac; \texttt{Repeat}\ tac), \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}{(\texttt{Repeat}\ tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)} \text{(Repeat)}$$

$$\frac{n = 0 \qquad \mathsf{new\_meta}(n)}{(\texttt{Do}\ n\ tac, \Delta[\Gamma \Vdash T]) \triangleright (?n, \Delta[\Gamma \Vdash T], \{(n, \Delta[\Gamma \Vdash T])\}, \sigma_{id})} \text{(Do-1)}$$

$$\frac{n > 0 \qquad (tac; \texttt{Do}\ (n-1)\ tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)}{(\texttt{Do}\ n\ tac, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)} \text{(Do-2)}$$

$$\frac{min\ i.(tac_i, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1),\ i = 1 \ldots n}{(\texttt{First}\ [tac_1 | \ldots | tac_n], \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1)} \text{(First)}$$

$$\frac{min\ i.(tac_i, \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], \emptyset, \sigma_1),\ i = 1 \ldots n}{(\texttt{Solve}\ [tac_1 | \ldots | tac_n], \Delta[\Gamma \Vdash T]) \triangleright (t_1, \Delta[\Gamma_1 \Vdash T_1], \emptyset, \sigma_1)} \text{(Solve)}$$

FIG. 5.12 – Évaluation des tacticals (2/2).

– $\mathsf{unicity}(m)$: Si $\nexists i$, $(?i, \Delta[\Gamma_i \Vdash T_i]) \in m$, $(?i, \Delta[\Gamma'_i \Vdash T'_i]) \in m$, avec $\Gamma_i \neq \Gamma'_i$ ou $T_i \neq T'_i$, renvoie m, sinon renvoie Erreur.

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans les figures B.10, B.11 et B.12.

$$\frac{(c, t, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1)}{(c., t, \Delta[\Gamma \Vdash T], m, p, d) \twoheadrightarrow (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1)}\ (\mathsf{Sen})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ n = min\{i \mid (?i, b) \in m\} \qquad (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (T, \Delta[\Gamma_n]) \rhd (T_2, \Delta[\Gamma_1 \Vdash s], \emptyset, \sigma) \qquad \mathsf{new\_meta}(k) \end{array}}{\begin{array}{c} (\mathtt{Let}\ x : T, t, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow \\ (?1, \Delta[\Gamma_1 \Vdash T_2], \{(?1, \Delta[\Gamma_1 \Vdash T_2])\}, \\ \lfloor([x : T_2](?n\ ?k), \Delta[\Gamma \Vdash T_1]\sigma, (m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup \\ \{(?n, \Delta[\Gamma_n\sigma, (x : T_2) \Vdash T_n\sigma]; (?k, \Delta[\Gamma_1 \Vdash T_2])\}, p\sigma, d)\rfloor \lhd p\sigma, \mathsf{Let}_k) \end{array}}\ (\mathsf{LetCur})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (T, \Delta[\Gamma_n]) \rhd (T_2, \Delta[\Gamma_1 \Vdash s], \emptyset, \sigma) \qquad \mathsf{new\_meta}(k) \end{array}}{\begin{array}{c} (\mathtt{Let}\ [?n]\ x : T, t, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow \\ (?1, \Delta[\Gamma_1 \Vdash T_2], \{(?1, \Delta[\Gamma_1 \Vdash T_2])\}, \\ \lfloor([x : T_2](?n\ ?k), \Delta[\Gamma \Vdash T_1]\sigma, (m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup \\ \{(?n, \Delta[\Gamma_n\sigma, (x : T_2) \Vdash T_n\sigma]; (?k, \Delta[\Gamma_1 \Vdash T_2])\}, p\sigma, d)\rfloor \lhd p\sigma, \mathsf{Let}_k) \end{array}}\ (\mathsf{LetGiv})$$

FIG. 5.13 – Évaluation des phrases (1/3).

5.4.4 Sémantique des scripts

Valeurs

Les valeurs de la sémantique des scripts sont les suivantes :

– $(t, \Delta[\Vdash T])$, où t est un terme pur, $\Delta[\Vdash T]$ est un but, tel que $\nexists j$, $?j, ?_{ij} \in t, T$.
– Erreur

$\mathcal{V}_\mathcal{S}$ désignera l'ensemble des valeurs de la sémantique des scripts.

Évaluation

On nommera *lemme*, tout but $\Delta[\Vdash T]$, où $\Delta[\Vdash T]$ est un but introduit par `Lemma` (voir l'exemple de la section 5.3, ainsi que [84]).

Définition 5.4.14 (Évaluation des scripts) *On dira que le script "$p_1.\ p_2.\ \ldots\ p_n.$" du lemme $\Delta[\Vdash T]$ s'évalue en v, avec $v \in \mathcal{V}_\mathcal{S}$, que l'on notera $(p_1.\ p_2.\ \ldots\ p_n., \Delta[\Vdash T]) \downarrow v$, si et seulement si $(p_1.\ p_2.\ \ldots\ p_n., \Delta[\Vdash T]) \downarrow v$ est dérivable en utilisant les règles de la figure 5.16.*

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ n = min\{i \mid (?i,b) \in m\} \qquad (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (t, \Delta[\Gamma_n]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \end{array}}{\begin{array}{c} (\mathtt{Let}\ x := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow \\ ([x : T_2](?n\ t_2), \Delta[\Gamma \Vdash T]\sigma, m\sigma, p\sigma, d) \end{array}} (\mathsf{LetOne}_1)$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (t, \Delta[\Gamma_n]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \end{array}}{\begin{array}{c} (\mathtt{Let}\ [?n]\ x := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow \\ ([x : T_2](?n\ t_2), \Delta[\Gamma \Vdash T]\sigma, m\sigma, p\sigma, d) \end{array}} (\mathsf{LetOne}_2)$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ n = min\{i \mid (?i,b) \in m\} \qquad (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (t, \Delta[\Gamma_n \Vdash T]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \\ \sigma_1 = \mathsf{unify}(T_n\sigma, T_2) \end{array}}{\begin{array}{c} (\mathtt{Let}\ x : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow \\ ([x : T_2](?n\ t_2)\sigma_1, \Delta[\Gamma \Vdash T_1]\sigma\sigma_1, m\sigma\sigma_1, p\sigma\sigma_1, d) \end{array}} (\mathsf{LetOne}_3)$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \qquad x \notin \Gamma_n \\ (t, \Delta[\Gamma_n \Vdash T]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \\ \sigma_1 = \mathsf{unify}(T_n\sigma, T_2) \end{array}}{\begin{array}{c} (\mathtt{Let}\ [?n]\ x : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow \\ ([x : T_2](?n\ t_2)\sigma_1, \Delta[\Gamma \Vdash T_1]\sigma\sigma_1, m\sigma\sigma_1, p\sigma\sigma_1, d) \end{array}} (\mathsf{LetOne}_4)$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (\mathtt{Let}\ c_1, t_0, \Delta[\Gamma_0 \Vdash T_0], m_0, p_0, d) \Rrightarrow (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d) \\ \vdots \\ (\mathtt{Let}\ c_n, t_{n-1}, \Delta[\Gamma_{n-1} \Vdash T_{n-1}], m_{n-1}, p_{n-1}, d) \Rrightarrow (t_n, \Delta[\Gamma_n \Vdash T_n], m_n, p_n, d) \end{array}}{\begin{array}{c} (\mathtt{Let}\ c_1\ \mathtt{And}\ \ldots\ \mathtt{And}\ c_n, t, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow \\ (t_n, \Delta[\Gamma_n \Vdash T_n], m_n, p_n, d) \end{array}} (\mathsf{LetAnd})$$

où $t_0 = t,\ \Gamma_0 = \Gamma,\ T_0 = T,\ m_0 = m$ et $p_0 = p$

FIG. 5.14 – Évaluation des phrases (2/3).

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}], m_n, \sigma) \\ m_u = \mathsf{unicity}((m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup m_n) \end{array}}{(?n := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow (t_1\sigma[?n \backslash t_n], \Delta[\Gamma \Vdash T]\sigma, m_u, p\sigma, d)} \ (\mathsf{Inst})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma}_b \text{ ou } d = \mathsf{Let}_{i,b} \\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m \\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}], m_n, \sigma) \\ \sigma_1 = \mathsf{unify}(T_{n1}, T\sigma) \\ m_u = \mathsf{unicity}(((m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup m_n)\sigma_1) \end{array}}{(?n : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \Rrightarrow (t_1\sigma[?n \backslash t_n]\sigma_1, \Delta[\Gamma \Vdash T_1]\sigma\sigma_1, m_u, p\sigma\sigma_1, d)} \ (\mathsf{InstCast})$$

$$\frac{t = ?1 \qquad d_1 = \mathsf{begin}(d)}{(\mathtt{Proof}, t, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow (t, \Delta[\Gamma \Vdash T], m, p, d_1)} \ (\mathsf{Proof})$$

$$\frac{\begin{array}{c} ?i \notin t \qquad d = \mathsf{Let}_{n,b} \\ p = \lfloor (t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1) \rfloor \triangleleft p_1 \qquad (?n, \Delta[\Gamma_n \Vdash T_n]) \in m_1 \end{array}}{\begin{array}{c} (\mathtt{Qed}, t, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow \\ (t_1[?_n \backslash t], \Delta[\Gamma_1 \Vdash T_1], m_1 - \{(?n, \Delta[\Gamma_n \Vdash T_n])\}, p_1, d_1) \end{array}} \ (\mathsf{Qed})$$

$$\frac{\Gamma = \emptyset \qquad ?k, ?_{ik} \notin t \qquad ?_{ik} \notin T \qquad m = \emptyset \qquad p = \lfloor\rfloor \qquad d = \mathsf{Lemma}_b}{(\mathtt{Save}, t, \Delta[\Gamma \Vdash T], m, p, d) \Rrightarrow (t, \Delta[\Vdash T], m, p, \mathsf{Lemma})} \ (\mathsf{Save})$$

FIG. 5.15 – Évaluation des phrases (3/3).

Les règles d'erreurs correspondantes peuvent être consultées en annexe B, dans la figure B.13.

$$\frac{\begin{array}{c} (T,\Delta[]) \triangleright (T_0,[\Gamma_0 \Vdash s],m_0,\sigma_0) \qquad s \in S \qquad m_0 = \emptyset \\ (p_1,t_0,\Delta[\Gamma_0 \Vdash T_0],m_0,p_0,d_0) \twoheadrightarrow \\ (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,p_1,d_1) \\ \vdots \\ (p_n,t_{n-1},\Delta[\Gamma_{n-1} \Vdash T_{n-1}],m_{n-1},p_{n-1},d_{n-1}) \twoheadrightarrow \\ (t_n,\Delta[\Gamma_n \Vdash T_n],m_n,p_n,d_n) \\ \Gamma_n = \emptyset \qquad ?k, ?_{ik} \notin t_n \qquad ?_{ik} \notin T_n \\ m_n = \emptyset \qquad p_n = \lfloor\rfloor \qquad d_n = \mathsf{Lemma} \end{array}}{(p_1.\ \ldots\ p_n.,\Delta[\Vdash T]) \downarrow (t_n,\Delta[\Vdash T_n])}\ (\mathsf{Script})$$

où $\Gamma_0 = \emptyset,\ p_0 = \lfloor\rfloor$ et $d_0 = \mathsf{Lemma}$

FIG. 5.16 – Évaluation des scripts.

5.5 Prototype

Dans cette section, nous référencerons, à titre indicatif, certaines fonctions et fichiers spécifiques de l'implantation de Coq, et ce, relativement au répertoire d'installation des sources. Le lecteur intéressé pourra récupérer les sources de Coq à l'adresse suivante :

http ://coq.inria.fr/

La sémantique précédente a été implantée sous la forme d'un prototype pour la version V7 de Coq. Le traitement des arguments implicites a cependant été affaibli. Notamment, il n'est pas possible d'avoir des arguments implicites dans le contexte d'une métavariable (voir la règle Meta, de la figure 5.3), et une partie procédurale ne peut s'appliquer à un terme contenant des arguments implicites non résolus (voir la règle By, de la figure 5.8).

Pour les termes, la machine de preuves possédait déjà un système de raffinement et d'extraction (fonction `refiner`, dans les fichiers `proofs/refiner.ml[i]`, et fonctions `prim_refiner`, `prim_extractor`, dans les fichiers `proofs/logic.ml[i]`). Toutefois, il était partiel et, en particulier, il ne passait pas sous les λ et ne pouvait donc pas raffiner les branches d'un *Cases*. La tactique `Refine` (voir [6]) permettait de pallier ce problème, mais de manière externe (le principe étant de générer les tactiques qui correspondent à la structure du terme). Comme il s'agissait de modifier le langage de preuves de Coq, il nous a semblé plus opportun de changer le système en profondeur, et de faire en sorte que le `refiner` soit plus puissant, rendant, de ce fait, la tactique `Refine` obsolète.

Quant aux parties déclaratives, les `Let...In` (voir les règles LetOne_i, dans la figure 5.14) sont directement interprétés comme des termes de coupures (applications), conformément à notre sémantique. Enfin, les parties procédurales n'ont pas changé de comportement. Elles sont systématiquement extraites des termes (si elles sont dans le contexte d'un terme), pour être transformées en instantiations (voir règles Inst et Inst-Cast, de la figure 5.15), où elles apparaissent à la racine.

Pour implanter le principe d'instantiations, on a modifié la structure des buts (type `goal`, dans les fichiers `proof_type.ml[i]`), de manière à ce qu'un but soit systématiquement associé à un numéro de métavariable, et que les instantiations puissent être ainsi guidées. Pour les `Let`'s à toplevel (voir les règles LetCur et LetGiv dans la figures 5.13), comme ils *ouvrent* une nouvelle session de preuve, il a fallu modifier le principe de définition des preuves, en ajoutant une *force* spécifique de déclaration (identifiée au moment du `Qed`), qui fait que l'objet preuve n'est pas visible en dehors du contexte du but auquel le `Let` s'applique. Une fois le `Qed` rencontré, la preuve est ajoutée, de manière non primitive, comme une simple coupure dans le contexte du but correspondant (il ne peut toutefois pas accéder à la preuve en elle-même[8]).

Enfin, l'algorithme d'unification (des arguments implicites) est basé sur [27], et a été complètement hérité de la version V7 courante de Coq[9]. Il est important de comprendre que l'unification se fait seulement par rapport aux arguments implicites et non par rapport aux métavariables. Ainsi, les instantiations de métavariables proviennent uniquement de l'utilisateur et l'unification ne génère aucune contrainte sur les métavariables. En particulier, comme l'indique la règle Meta de la figure 5.3, les métavariables ne dépendent pas les unes des autres, c'est-à-dire qu'aucune métavariable ne peut apparaître dans le contexte ou le type d'une métavariable. Cela constitue une restriction par rapport à l'implantation d'ALF [55] et aux différents calculs présentés dans [64].

5.6 D'autres exemples

5.6.1 Problème des timbres

Comme premier exemple, nous allons donner la preuve du problème des timbres, emprunté au *tutorial* de PVS [20], et qui affirme que tout affranchissement postal d'au moins 8 *cents* peut être complété avec uniquement des timbres de 3 et 5 cents. Mathématiquement, cela signifie que tout entier naturel supérieur à 8 est la somme d'un multiple positif de 3 et d'un multiple positif de 5.

Pour exprimer le lemme, nous mettrons les multiples de 3 et de 5, dans les entiers relatifs (type `Z`), afin de pouvoir bénéficier de la tactique `Ring`, qui résout les égalités sur les anneaux abéliens. Sur les entiers naturels (type `nat`), `Ring` fonctionne aussi, mais seulement sur la structure de semi-anneau, si bien qu'il ne sait pas manipuler des expressions comme $n - m$, même si $n \geq m$. Puisque certains *témoins*, que l'on exhibera, auront cette forme, nous avons donc préféré nous mettre dans `Z`, pour éviter de faire les preuves manuellement. Toutefois, nous ferons en sorte de donner systématiquement des témoins positifs, de manière à respecter l'énoncé original.

Nous proposons la preuve suivante :

```
Lemma L3_plus_5:(n:nat)(EX t:Z|(EX f:Z|`(inject_nat n)+8=3*t+5*f`)).
Proof.
  Let cb:(EX t:Z|(EX f:Z|`(inject_nat O)+8=3*t+5*f`)).
  Proof.
    ?1 := (ex_intro ? ? `1` (ex_intro ? ? `1` <by Ring>)).
  Qed.
```

[8]La preuve pourra, cependant, être retrouvée au moment du `Save` final.

[9]L'algorithme de la V7 provient lui-même des anciennes versions V6.

```
7      Let ci:(n:nat)(EX t:Z|(EX f:Z|`(inject_nat n)+8=3*t+5*f`))->
8                    (EX t:Z|(EX f:Z|`(inject_nat (S n))+8=3*t+5*f`)).
9      Proof.
10       ?1 := [n;Hrec](ex_ind ? ? ? ?2 Hrec).
11       ?2 := [x;H:(EX f:Z|`(inject_nat n)+8=3*x+5*f`)](ex_ind ? ? ? ?3 H).
12       ?3 := [x0;H0:`(inject_nat n)+8 = 3*x+5*x0`]?4.
13       Let cir:(EX t:Z|(EX f:Z|`3*x+5*x0+1=3*t+5*f`)).
14       Proof.
15         ?1 :=
16           <[_:?](EX t:Z|(EX f:Z|`3*x+5*x0+1=3*t+5*f`))>
17           Cases (dec_eq `x0` `0`) of
18           | (or_introl H) => ?2
19           | (or_intror _) => ?3
20           end.
21         ?2 := (ex_intro ? ? `x-3` (ex_intro ? ? `2` <by Rewrite H;Ring>)).
22         ?3 := (ex_intro ? ? `x+2` (ex_intro ? ? `x0-1` <by Ring>)).
23       Qed.
24       ?4 := <by Rewrite inj_S;Rewrite Zplus_S_n;Unfold Zs;Rewrite H0;
25                 Exact cir>.
26     Qed.
27     ?1 := (nat_ind ([n:nat]?) cb ci).
28   Save.
```

Cette preuve s'effectue par récurrence (sur `n`). Le cas de base (`cb`) est *déclaré* en ligne 3, et le cas récurrent (`ci`), en lignes 7 et 8. Pour prouver le cas de base, où `n=0`, les témoins sont trivialement `1` et `1`, puisque `0+8=3*1+5*1`. En ligne 5, les témoins sont donnés directement avec le constructeur de l'existentielle (dans `Prop`), à savoir `ex_intro`, puis on appelle `Ring`, pour résoudre la précédente égalité. Pour le cas récurrent, on introduit, en ligne 10, le rang de l'entier (`n`), ainsi que l'hypothèse de récurrence (`Hrec`). Puis, on skolémise `Hrec`, en utilisant le schéma d'élimination de l'existentielle `ex_ind`. En ligne 11, on introduit le symbole de skolem (`x`) et l'hypothèse skolémisée (`H`), qui est, à nouveau skolémisée. En ligne 12, on introduit le symbole (`x0`) et l'hypothèse issus de la deuxième skolémisation. Ensuite, on remarque qu'en réarrangeant un peu la conclusion (notamment sortir le successeur), on peut faire apparaître le membre gauche de l'hypothèse de récurrence skolémisée. On fait donc, en ligne 13, une coupure (`cir`) de la conclusion après cette *réécriture triviale*. Pour prouver ce lemme intermédiaire, on effectue, des lignes 15 à 20, un raisonnement par cas, suivant que `x0=0` ou non. En ligne 21, si `x0=0`, alors les témoins sont `x-3` et `2`, puis l'égalité est résolue par `Ring`, après avoir remplacé `x0` par 0 (`Rewrite H`). Dans le cas contraire, en ligne 22, les témoins sont `x+2` et `x0-1`, puis l'égalité est résolue directement par `Ring`. Une fois `cir` prouvé, on effectue, en lignes 24 et 25, la réécriture en question pour faire apparaître le type de `cir`, et l'utiliser pour conclure (avec `Exact`) la preuve de `ci`. Enfin, en ligne 27, on réalise la récurrence sur `n` (avec `nat_ind`), à laquelle on donne directement les preuves correspondantes `cb` et `ci`.

On peut remarquer que, dans le cas récurrent, `x+2` et `x0-1` sont des témoins valides, quel que soit la valeur de `x0`. Toutefois, si, au rang d'avant, on a `x0=0` (par exemple, si `n=9`, `9=3*3+5*0`), on obtient alors, pour `n+1`, une valeur négative pour le multiple de 5 (pour `n+1=10`, on a `10=3*5+5*(-1)`), ce qui est gênant sachant qu'il s'agit d'un nombre de timbres de 5 cents. Pour éviter cela, on traite explicitement le cas `x0=0`, pour lequel les témoins sont `x-3` et `2` (pour `n+1=10`, on a alors `10=3*0+5*2`). Dans ce cas, il est inutile de scinder, à nouveau, en deux cas, suivant que `x<3` ou non, puisque `x0=0` et si `x<3`, alors `n+8>3*x`, ce qui contredit l'hypothèse de récurrence.

Concernant le style, on peut noter que la réécriture est, *a priori*, compatible avec $\mathcal{L}_{pdt}$. Il suffit, en effet, d'effectuer une coupure de l'expression après réécriture, que l'on utilise ensuite directement dans la partie procédurale chargée de faire la réécriture. Cette technique permet ainsi de faire de la réécriture de manière procédurale[10], et ce, au milieu d'une preuve. Cela semble plutôt raisonnable et satisfaisant, si la réécriture est utilisée de manière assez occasionnelle dans la preuve. Dans le cas contraire, par exemple, pour des preuves s'effectuant purement par réécritures (lemmes prouvés sur des axiomatisations), cette méthode risque de se révéler un peu rigide, et une extension intéressante de $\mathcal{L}_{pdt}$ serait de fournir une syntaxe appropriée au raisonnement équationnel.

Remarque 5.6.1 (Arguments implicites et définitions syntaxiques) *Comme on peut le voir dans la preuve précédente, étant donné que l'on utilise beaucoup le langage de termes, on est souvent amené à utiliser des arguments implicites et à donner uniquement le strict nécessaire (pour que le typage puisse résoudre et instantier les termes manquants). Cette utilisation intensive des arguments implicites a pour conséquence que l'on pourrait désirer ne pas mettre du tout ces arguments (pour aller plus vite et par souci de concision du script). Cela peut être fait au moyen des définitions syntaxiques (*`Syntactic Definition`*) de* `Coq`*, qui se comportent exactement comme des macros*[11]*. Par exemple, si l'on souhaite définir une instantiation et une skolémisation* allégées *de l'existentielle* `EX`*, on utilisera les commandes suivantes :*

```
Syntactic Definition choose := (ex_intro ? ?).
Syntactic Definition skolem := (ex_ind ? ? ?).
```

Ainsi, on pourra remplacer les lignes 5 et 10 respectivement par :

```
?1 := (choose ‘1‘ (choose ‘1‘ <by Ring>)).
?1 := [n;Hrec](skolem ?2 Hrec).
```

ce qui permet, au passage, d'associer une valeur sémantique spécifique à `ex_intro` *et* `ex_ind`*.*

5.6.2 Preuve en théorie des ensembles

Nous nous proposons de montrer une propriété connue de la théorie des ensembles, à savoir qu'étant donnés trois ensembles A, B, et C, on a $(A \cap B) \cup C \subset (A \cup C) \cap (B \cup C)$. L'inclusion inverse est également vérifiée et se montre de manière duale. Pour effectuer cette preuve, nous avons utilisé le type `Ensemble` du module `Ensembles`, disponible dans la bibliothèque standard de `Coq`.

Nous avons réalisé la preuve comme suit :

```
Lemma inter_union:
  (A,B,C:(Ensemble U))
    (Included U (Union U (Intersection U A B) C)
                         (Intersection U (Union U A C) (Union U B C))).
```

[10] C'est, pour le moment, la seule valable. En effet, on pourrait utiliser, au niveau du langage de λ-termes, le schéma d'élimination de l'égalité, mais c'est loin d'être pratique, car il faut expliciter beaucoup d'arguments, ce qui a tendance, de surcroît, à casser la lisibilité.

[11] En particulier, les définitions syntaxiques ne sont pas *déchargées* à la fermeture des sections, et si l'on souhaite des objets plus *persistants*, il faut alors utiliser la commande `Grammar`, qui, de surcroît, permet d'avoir une liberté totale sur la position des arguments implicites. Voir [6], pour plus de détails.

```
5   Proof.
6     ?1 := [A,B,C:(Ensemble U)]?2.
7     ?2 : (x:U)(In U (Union U (Intersection U A B) C) x) ->
8               (In U (Intersection U (Union U A C) (Union U B C)) x) :=
9       [x;H]?3.
10    Let in_ab: (x:U)(In U (Intersection U A B) x) ->
11                    (In U (Intersection U (Union U A C) (Union U B C)) x).
12    Proof.
13      Let in_a_b: (x:U)(In U A x) -> (In U B x) ->
14                       (In U (Intersection U (Union U A C)
15                                             (Union U B C)) x).
16      Proof.
17        ?1 := [x;Ha;Hb](Intersection_intro ? ? ? x ?2 ?3).
18        ?2 := <by Apply Union_introl;Assumption>.
19        ?3 := <by Apply Union_introl;Assumption>.
20      Qed.
21      ?1 :=
22        [x;Hab]<[u:U][_:(In U (Intersection U A B) u)]
23                (In U (Intersection U (Union U A C) (Union U B C)) u)>
24        Cases Hab of
25        | (Intersection_intro x0 a0 b0) => (in_a_b x0 a0 b0)
26        end.
27    Qed.
28    Let in_c: (x:U)(In U C x) ->
29                   (In U (Intersection U (Union U A C) (Union U B C)) x).
30    Proof.
31      ?1 := [x;Hc](Intersection_intro ? ? ? x ?2 ?3).
32      ?2 := <by Apply Union_intror;Assumption>.
33      ?3 := <by Apply Union_intror;Assumption>.
34    Qed.
35    ?3 :=
36      <[u:U][_:(In U (Union U (Intersection U A B) C) u)]
37       (In U (Intersection U (Union U A C) (Union U B C)) u)>
38      Cases H of
39      | (Union_introl x0 ab0) => (in_ab x0 ab0)
40      | (Union_intror x0 c0) => (in_c x0 c0)
41      end.
42  Save.
```

La preuve se réalise par cas, suivant que si $x \in (A \cap B) \cup C$, alors $x \in A \cap B$ ou $x \in C$. En ligne 6, on introduit les trois ensembles (`A`, `B` et `C`). Des lignes 7 à 9, on explicite ce que signifie `Included`, à savoir que `(Included U E F)`≡ `(x:U)(In U E x)->(In U F x)`, où `In` est le symbole d'appartenance. On introduit alors l'élément et l'hypothèse correspondante. Des lignes 10 à 27, il s'agit du cas où $x \in A \cap B$ (`in_ab`). Pour le prouver, on montre, des lignes 13 à 20, le même lemme, mais en supposant que $x \in A$ et $x \in B$ (`in_a_b`). Pour ce faire, on introduit l'élément (`x`) et les deux hypothèses d'appartenance (`Ha` et `Hb`), avant de séparer la conclusion en deux parties, où $x \in A \cup B$ et $x \in B \cup C$. En lignes 18 et 19, ces deux cas sont trivialement résolus, en choisissant le cas gauche du type `Union` (`Union_introl`). Pour maintenant montrer `in_ab`, des lignes 21 à 26, on introduit l'élément (`x`), puis l'hypothèse que $x \in A \cap B$, pour pouvoir raisonner par cas dessus, et appeler directement in_a_b, en ligne 25. Il reste alors le deuxième cas, où $x \in C$ (in_c). Cette preuve est réalisée, des lignes 30 à 34, de manière complètement similaire à in_a_b, excepté qu'il faut choisir le côté droit de Union (Union_intror), en lignes 32 et 33. Une fois les deux cas prouvés, on réalise effectivement, des lignes 35 à 41, le raisonnement par cas sur l'hypothèse $x \in (A \cap B) \cup C$, en appelant ensuite directement in_ab et in_c, respectivement en lignes 39 et 40.

Cette preuve montre une spécificité de $\mathcal{L}_{pdt}$, que l'on n'avait pas rencontrée jusqu'à présent, à savoir la conversion. En effet, on a vu, des lignes 7 à 9, qu'il suffit d'expliciter le nouveau type de la conclusion, pour réaliser automatiquement la δ-réduction sur Included (c'est exactement l'équivalent de la tactique Change, excepté qu'ici, on connaît aussi la conclusion d'origine). Il n'y a donc pas besoin de constructions supplémentaires pour la réduction, puisque c'est géré par le système. Cette position peut être considérée comme peu flexible, sachant que la technique, consistant à utiliser certaines procédures de réduction, et *à voir ce qui se passe*, peut sembler confortable, mais elle ne s'inscrit que dans une optique purement procédurale nécessitant la présence d'un toplevel, ce qui va, en partie, à l'encontre de la philosophie de $\mathcal{L}_{pdt}$.

5.7 Discussion

5.7.1 Synthèse

Comme on a pu le voir avec les exemples précédents, $\mathcal{L}_{pdt}$ se révèle être une proposition satisfaisante de fusion des styles procédural, déclaratif et à base de termes. Ce langage fournit ainsi une liberté significative dans le processus de construction et surtout d'expression d'une preuve. Toutefois, l'étude comparée, effectuée dans la section 5.1, a montré que cette grande flexibilité n'était, en réalité, intéressante que si chacun des styles était utilisé dans des situations de preuves bien définies. En effet, rien n'empêche, *a priori*, l'utilisateur de se limiter à un unique style, et de ne construire, par exemple, toutes ses preuves qu'en procédural, au risque de perdre, entre autres, la lisibilité de ses scripts. Pour éviter ce phénomène, il semble clair que $\mathcal{L}_{pdt}$ ne peut pas imposer de contraintes syntaxiques particulières, basées sur des critères plutôt sémantiques, et l'utilisateur doit donc se plier à une certaine discipline s'il souhaite optimiser la qualité de sa preuve. En particulier, on utilisera le style procédural pour de petites preuves simples, réalisées en backward et pour lesquelles on n'est pas intéressé par les détails formels. Le style déclaratif sera utilisé dans des parties de preuves plus élaborées, construites en mode forward, avec beaucoup d'informations pour le lecteur. Enfin, on utilisera le style à base de termes, pour traiter des preuves complexes, en backward, et pour lesquelles il est possible de choisir le niveau de détails (mettre le type des abstractions ou pas, utiliser des arguments implicites ou donner les termes complets).

Ainsi, $\mathcal{L}_{pdt}$ peut être vu à la fois comme un nouveau langage de description de preuves, mais aussi comme une volonté de créer une méthode permettant de connaître l'adéquation des différents styles de preuves. Cependant, comme cette méthode n'est pas contraignante, il n'est pas possible de caractériser, de manière générale, $\mathcal{L}_{pdt}$ selon les critères vus dans la section 1.3 du chapitre 1. Cela dépend, en effet, de la façon dont la preuve a été exprimée, et l'utilisateur est complètement responsable de la description de sa preuve, si bien qu'il peut choisir de bénéficier systématiquement des avantages des trois styles dans toute sorte de preuves, ou sacrifier certains critères qu'il peut juger ponctuellement non significatifs.

5.7.2 Extensions et travaux futurs

Raisonnement équationnel

Comme on a pu le voir avec la preuve du problème des timbres, dans la section 5.6.1, $\mathcal{L}_{pdt}$ n'est pas encore très approprié pour effectuer facilement des preuves par réécritures. On peut, certes, *simuler* des étapes de réécritures, mais plutôt ponctuellement, et il faut, de surcroît, que ces étapes puissent être considérées comme suffisamment triviales, afin que la coupure engendrée soit logiquement et même facilement reliable au but d'origine. Une autre méthode reste donc à trouver pour pouvoir traiter raisonnablement les preuves utilisant fortement le raisonnement équationnel.

Une idée pourrait être d'implanter un système similaire à celui du Mizar-mode [41] pour HOL Light [39], où l'on peut itérer le raisonnement égalitaire. Par exemple, on pourrait écrire :

```
 T1 = T2 <by tac1>
... = T3 <by tac2>
... = T4 <by tac3>
```

pour construire une preuve de `T1 = T4`, les `"..."` dénotant le membre droit de l'égalité précédente. Cela permettrait de réaliser des pas non triviaux de réécriture, qui seront toujours réalisés procéduralement, et qui seront compréhensibles en dehors du toplevel. Par ailleurs, cette chaîne n'est pas, *a priori*, orientée, et on pourrait imaginer faire aussi bien du *forward* que du *backward chaining*.

Plus de déclaratif

$\mathcal{L}_{pdt}$ fournit une variété de `Let`'s (voir les règles LetCur, LetGiv, et LetOne_i, dans les figures 5.13 et 5.14), qui ajoutent autant de traits déclaratifs (surtout les `Let`'s à toplevel). Toutefois, la manière, dont ils sont combinés, par la suite, pour prouver des lemmes plus complexes, n'est pas entièrement satisfaisante. En effet, comme on peut le remarquer pour les deux preuves précédentes, dans les sections 5.6.1 et 5.6.2, les lemmes intermédiaires, déclarés par des `Let`'s, sont simplement et directement utilisés. Aucune automatisation particulière, visant à les combiner, n'est sollicitée pour pouvoir résoudre. C'est une caractéristique des langages déclaratifs, qui serait intéressante à intégrer, car, si l'on fait abstraction d'ACL2, qui est très automatisé, même Mizar possède un système de déductions non trivial, caché derrière le mot-clé `by`.

Un premier pas, à moindre coût, vers cette extension, serait d'utiliser naturellement la tactique `Auto`. Par exemple, on peut, d'ores et déjà, écrire des scripts comme :

```
Lemma triv:A\/B.
Proof.
  Let prf_A:A.
  Proof.
    ...
  Qed.
  ?1 := <by Auto>.
Save.
```

car, le contexte du but initial, lié à la métavariable `?1`, est enrichi par la preuve `prf_A` de `A`, et `Auto` peut alors conclure. Il est clair qu'on souhaiterait encore plus, avec, par exemple, la possibilité, pour `Auto`, de faire certaines éliminations, afin de voir si les différents cas ont été prouvés (les preuves précédentes seraient plus courtes, surtout celle sur la théorie des ensembles, dans la section 5.6.2). Ainsi, plus de déclaratif dans $\mathcal{L}_{pdt}$, passe sans aucun doute par plus d'automatisation dans `Auto`.

Deuxième partie

Langages de tactiques

Chapitre 6

Le langage de tactiques $\mathcal{L}_{tac}$

Dans ce chapitre, nous allons étudier les langages qui permettent d'étendre l'automatisation d'un outil d'aide à la preuve. Nous nous placerons dans le cadre de systèmes dits *à la* LCF, où l'utilisateur peut coder ses propres automatisations au moyen d'un langage de haut niveau, appelé *métalangage*. Dans le cas particulier de Coq, nous allons montrer que ce métalangage n'est pas approprié dans certaines situations, et nous proposerons un nouveau langage, appelé $\mathcal{L}_{tac}$, censé non pas se substituer au métalangage, mais l'assister précisément dans ces situations. Ce chapitre est une version complétée de la publication réalisée à LPAR'2000 [23].

6.1 Métalangages

6.1.1 Le paradigme de LCF

En 1979, M. J. C. Gordon, R. Milner et C. P. Wadsworth [36] réalise un outil d'aide au raisonnement formel appelé LCF. LCF signifie alors *Logic for Computable Functions* (qui est le nom d'une logique conçue par Dana Scott), et est implémenté, à l'origine, à l'Université d'Edinburgh. Parmi ce qui constitue le paradigme de LCF, on peut notamment citer les éléments suivants :

1. **Preuves et calcul :** les preuves sont vues comme des fonctions, qui, à partir de théorèmes déjà prouvés, *calculent* d'autres théorèmes. Ces fonctions réalisent des inférences, qui sont garanties correctes par l'intermédiaire d'un type abstrait de théorèmes.
2. **Un type abstrait de théorèmes :** la logique de LCF est représentée par un type abstrait de théorèmes. Les constructeurs de ce type correspondent exactement aux axiomes et aux règles de déduction de la logique choisie, si bien que toute valeur de ce type est une preuve.
3. **Un métalangage fonctionnel :** à l'opposé du langage *objet* de LCF (le langage de la logique), un langage fonctionnel, appelé ML [34] (pour *MetaLanguage*), est conçu pour servir de métalangage. Il permet l'implantation du type abstrait de théorèmes, ainsi que l'élaboration de techniques pour construire des preuves automatiquement.
4. **Des tactiques :** les tactiques sont des méthodes de construction de preuves. À partir d'une proposition à montrer, appelé *but* (constitué d'*hypothèses* et d'une *conclusion*), elles produisent un ou plusieurs autres buts (potentiellement plus simples), appelés sous-buts, ainsi qu'une fonction capable de construire la dérivation correspondante (du type abstrait de théorèmes), lorsque les sous-buts auront été prouvés.

5. **Des fonctions d'ordre supérieur :** le métalangage autorise l'utilisation de fonctions d'ordre supérieur, qui se révèlent adaptées à la construction de nouvelles automatisations. En particulier, on utilisera des fonctions d'ordre supérieur dans la combinaison de tactiques.

Le métalangage sert principalement à coder de nouvelles automatisations, notamment sous la forme de tactiques. Ainsi, par la suite, dans le cadre d'un outil d'aide à la preuve *à la* LCF, nous désignerons, par *langage de tactiques*, le métalangage, et *vice-versa*, bien qu'il s'agisse d'un abus de langage, dans la mesure où le métalangage est strictement plus général.

6.1.2 Évolution de ML

Les premières versions de LCF sont implantées en Stanford Lisp [74]. À l'INRIA de Rocquencourt, Gérard Huet fait ensuite un portage vers Franz Lisp [30], et cette version sera alors appelée Cambridge LCF [68, 69] lorsqu'elle sera développée par Larry Paulson à Cambridge (la version originale de LCF devenant ainsi Edinburgh LCF). Il y a alors globalement deux directions distinctes suivant Rocquencourt et Cambridge. À Rocquencourt, Gérard Huet et Thierry Coquand commence, en 1984, l'implantation de Coq [6], descendant directement de LCF. À Cambridge, Mike Gordon et Tom Melham réalise les premières versions de HOL [35], au-dessus de Cambridge LCF.

C'est alors que ML devient un vrai langage de programmation, et non plus seulement le métalangage d'outils d'aide à la preuve basés sur LCF. Il possède ainsi de nombreuses variantes : Caml [89] (Caml Light [53], Caml Special Light [52], Objective Caml [54]), Standard ML [60, 61] (Moscow ML [78], Standard ML of New Jersey [85]), Lazy ML [3, 4, 5] (Haskell [45]), ... Ainsi, la version 4.10 de Coq, distribuée en 1989, est implantée en Caml, et, au début des années 90, HOL90 utilise Standard ML. D'autres systèmes, utilisant le même principe, vont suivre comme Lego [72], Nuprl [17, 43, 44] ou Isabelle [71, 70].

Cette évolution *hiérarchique* de ML, au-delà du fait qu'elle *bouscule* la conception de la programmation fonctionnelle, a des conséquences intéressantes sur les outils d'aide à la preuve à la LCF. En effet, cette fusion entre le métalangage et le langage d'implantation libère complètement le champ d'application des tactiques, qui peuvent désormais interagir plus *fortement* et plus *profondément* avec le système. C'est une avancée significative, qui permet l'élaboration de tactiques plus complexes[1].

6.2 Un exemple

Pour donner un premier exemple de tactique écrite en ML, nous nous placerons dans le cadre du système Coq, dont Objective Caml est à la fois le métalangage et le langage d'implantation. Nous nous proposons d'élaborer une tactique qui, étant donné un but, *élimine* toutes les hypothèses de la forme $A_1 \wedge A_2 \wedge \ldots \wedge A_n$, pour donner n hypothèses de types A_1, A_2, ..., et A_n. Cette tactique ne correspond à aucune tactique primitive de Coq, et ne peut pas être construite en utilisant les *tacticals* (qui sont des combinateurs de tactiques) du système. Il n'y a donc pas d'autre alternative que de coder cette tactique en Objective Caml. Pour ce faire, nous allons utiliser la version bytecode du toplevel de Coq (programme `coqtop.byte`), qui permet d'accéder à un toplevel Objective Caml, dans lequel nous allons écrire notre tactique :

```
> coqtop.byte
```

[1] En effet, il est maintenant possible d'utiliser *toutes* les fonctionnalités du système pour développer des tactiques, car le métalangage n'est plus seulement une interface non exhaustive de la machine de preuves.

```
Welcome to Coq 7.1 (September 2001)

Coq < Drop.
        Objective Caml version 3.01

        Camlp4 Parsing version 3.01

# #use "include";;
# open Tactics;;
# let is_and (n,c) =
    match (kind_of_term c) with
    | IsApp (h,_) ->
      let (sp,_) = op_of_mind h in
      if (string_of_id (basename sp)) = "and" then n
      else failwith "Not an And"
    | _ -> failwith "Not an And";;
val is_and : 'a * Term.constr -> 'a = <fun>
# let elim_and g =
    let hyps  = List.rev (pf_hyps_types g) in
    let lhyps = List.fold_left (fun a e -> try a@[is_and e] with _ -> a)
                               [] hyps in
    if lhyps = [] then tclFAIL 0 g
    else tclTHEN (List.fold_left (fun a e -> tclTHEN a (andE e))
                                 tclIDTAC lhyps) (clear lhyps) g;;
val elim_and :
  Proof_type.goal Tacmach.sigma ->
  Proof_type.goal list Proof_type.sigma * Proof_type.validation = <fun>
# let elim_ands = hide_atomic_tactic "ElimAnds" (tclREPEAT elim_and);;
val elim_ands : Tacmach.tactic = <fun>
# go ();;

Coq < Grammar tactic simple_tactic: ast :=
Coq < | elim_ands [ "ElimAnds" ] -> [(ElimAnds)].

Coq < Goal (A,B,C,D:Prop)A/\B->B/\C/\D->A.

Unnamed_thm < Intros. ElimAnds.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  H1 : A
  H2 : B
  H3 : B
  H : C
  H0 : D
  ============================
   A
```

En ligne 1, on exécute effectivement la version bytecode du toplevel de Coq, puis, en ligne 4, on *quitte* Coq momentanément, pour lancer un toplevel Objective Caml. En ligne 9, on interprète le fichier `include`, qui initialise partiellement l'environnement de développement. Notamment, il installe des pretty-printers (pour les termes, les tactiques, ...), et *ouvre* certains modules, permettant un accès direct aux fonctions exportées. En ligne 10, on ouvre le module `Tactics`, qui contient les principales tactiques primitives de Coq.

Des lignes 11 à 17, il s'agit de la fonction `is_and`, qui, pour une hypothèse $(x_i : A_i)$, teste si A_i est de la forme $A \wedge B$, et rend x_i si c'est le cas. En particulier, en ligne 12, la fonction `kind_of_term` permet d'effectuer du filtrage sur les termes Coq, de manière abstraite (sans faire aucune supposition sur leur représentation réelle[2]). En ligne 13, on tente de reconnaître une application (`IsApp`), puis, en ligne 12, on essaie d'extraire, de la tête de l'application (`h`), le nom du type inductif (avec `op_of_mind`). En ligne 15, si ce nom correspond bien à `and`, on renvoie le nom de l'hypothèse (`n`), sinon on échoue en ligne 16. En ligne 17, si ce n'est pas une application, on échoue directement.

Des lignes 19 à 25, c'est la tactique `elim_and`, qui, étant donné un but, élimine tous les $\wedge$ à la racine, dans toutes les hypothèses. En détails, on récupère d'abord, en ligne 20, la liste des hypothèses (avec `pf_hyps_types`) du but (`g`), que l'on inverse (en effet, elles sont inversées par rapport à l'ordre d'affichage). Ensuite, en lignes 21 à 22, on construit, grâce à `is_and`, la liste des noms d'hypothèses de la forme $A \wedge B$. Si la liste ainsi construite est vide, on échoue, en ligne 23, avec la tactique `tclFAIL` (correspondant à `Fail`). Dans le cas contraire, des lignes 24 à 25, on construit une séquence d'élimination avec `andE` (qui ne correspond à aucune tactique primitive définie à toplevel et qui introduit directement les composantes du $\wedge$ éliminé dans les hypothèses) et `tclTHEN` (correspondant au tactical `;`), avant d'effacer les hypothèses ainsi éliminées, avec `clear` (correspondant à `Clear`).

En ligne 29, on définit la tactique principale `elim_ands`, qui est *enregistrée* (avec `hide_atomic_tactic`) sous le nom `ElimAnds`, et qui consiste à réitérer, au moyen de `tclREPEAT` (correspondant au tactical `Repeat`), la tactique `elim_and`, jusqu'à ce qu'il n'y ait plus de $\wedge$ dans les hypothèses. On peut, ensuite, retourner au toplevel de Coq, en ligne 31, pour aller tester la tactique. On doit d'abord, en lignes 33 et 34, étendre la grammaire de Coq (commande `Grammar`), de manière à ce que la tactique soit reconnue en tant que tactique, et non comme un simple identificateur. On peut alors définir, en ligne 36, un nouveau but à prouver (commande `Goal`), et tester `ElimAnds`, avec succès, des lignes 38 à 51.

Remarque 6.2.1 (Mode batch) *Dans cet exemple, nous avons utilisé le toplevel de* Coq*, mais il est également possible de coder cette tactique en mode batch. Pour cela et à titre indicatif, la partie* ML *doit être mise dans un fichier* ML *(extension* `.ml`*), dans lequel il faut éventuellement rajouter certaines ouvertures de modules (typiquement, ceux qui sont ouverts dans* `include` *et utilisés dans la tactique). On doit aussi créer un fichier* Coq *(extension* `.v`*), dans lequel on doit mettre une commande de chargement du fichier* ML *(*`Declare ML Module`*) et la règle d'extension de grammaire vue précédemment. Enfin, il faut compiler le fichier* ML*. La tactique est alors utilisable, soit en interprétant le fichier* Coq *(commande* `Load`*), soit en compilant le fichier* Coq *(avec le programme* `coqc`*) et en le chargeant (commande* `Require`*).*

[2]En réalité, actuellement, la structure des termes coïncide exactement avec celle renvoyée par `kind_of_term`, mais cette barrière d'abstraction permettra de changer éventuellement la représentation des termes dans de futures versions du système, sans avoir à modifier les fonctions manipulant les termes. En particulier, cela tend à accroître la robustesse du code des tactiques (système ou utilisateurs), qui, de ce point de vue là, n'auront pas à être portées.

6.3 Adéquation du métalangage

Dans tout système d'aide à la preuve à la LCF, il semble légitime que le métalangage soit Turing-complet, c'est-à-dire, que l'utilisateur doit être en mesure d'élaborer des tactiques sans aucune limitation imposée par le langage lui-même. Si cette caractéristique est généralement vérifiée, elle en devient d'autant plus concrète lorsque le métalangage correspond aussi au langage d'implantation du prouveur. Ainsi, le choix d'un tel métalangage a plusieurs conséquences, qui doivent être prises en compte :

- le système doit fournir des moyens pour éviter d'éventuelles incohérences introduites par des tactiques utilisateurs. Cela peut être fait de plusieurs façons. Dans le plus pur style LCF, comme en HOL, on passe naturellement par le type abstrait de théorèmes, où seules les opérations données par ce type (supposées sûres) sont utilisées. Dans le style à la Curry-Howard, comme en Coq, les tactiques ne sont pas contraintes, et c'est le vérificateur de types, qui vérifie que le terme, construit par la tactique, est du type du théorème à prouver.
- l'utilisateur doit apprendre un autre langage, qui est, en général, très différent du langage objet et du langage de preuves (tactiques primitives et tacticals). Il est donc important de considérer combien de temps l'utilisateur est-il prêt à passer dans cette tâche, qui peut s'avérer plutôt difficile ou, au moins, fastidieuse.
- le langage doit avoir un debugger complet, car, trouver des erreurs dans le code d'une tactique, est bien plus complexe que dans les scripts de preuves développés dans le langage de preuves, où le système est supposé assister l'utilisateur dans la localisation des erreurs.
- le système doit avoir un code clair et bien documenté, surtout en ce qui concerne la machine de preuves. L'utilisateur doit être capable d'identifier facilement et rapidement les primitives nécessaires à l'élaboration de sa tactique, ou il pourrait facilement se perdre dans tous les fichiers et simplement abandonner.
- si le métalangage est aussi le langage d'implantation du système, l'utilisateur doit être conscient que ses tactiques seront potentiellement sensibles aux évolutions internes du système. L'utilisateur devra se servir, au maximum, des barrières d'abstraction du système, et moins le système en fournira, moins les tactiques utilisateurs seront robustes.

Ainsi, on peut remarquer qu'écrire des tactiques dans un langage complet implique beaucoup de contraintes pour les développeurs du système et plus spécialement pour les utilisateurs. En fait, il faut reconnaître que la procédure n'est pas vraiment facile, mais il n'y pas d'autre choix si nous voulons éviter des restrictions sur les tactiques. Cependant, on peut se demander si cette méthode est appropriée dans tous les cas. Par exemple, si nous voulons une tactique qui résout les équations et inéquations dans l'arithmétique de Presburger, c'est plutôt un problème non trivial, nécessitant un langage complet. Mais, supposons maintenant que nous souhaitions montrer que l'ensemble des entiers naturels contient plus que deux éléments. Cela peut s'exprimer de la façon suivante :

$$\vdash (\exists x : \mathbb{N}.\exists y : \mathbb{N}.\forall z : \mathbb{N}.x = z \vee y = z) \rightarrow \bot$$

Pour montrer cette proposition, nous introduisons le membre gauche et nous le skolémisons successivement deux fois. Nous obtenons alors le séquent :

$$\forall z : \mathbb{N}.x = z \vee y = z \vdash \bot$$

Nous contractons alors trois fois l'hypothèse, et l'instantions avec trois entiers naturels distincts (par exemple 0, 1 et 2). Nous éliminons alors tous les $\vee$. On obtient ainsi huit séquents, avec, en hypothèses, trois égalités avec x ou y, en membre gauche, et 0, 1 ou 2, en membre droit. Pour conclure dans tous les cas, il suffit d'appliquer la transitivité de l'égalité entre deux égalités possédant le même membre gauche, ce qui donne une égalité entre deux entiers naturels distincts, d'où la contradiction.

Dans le système Coq, la preuve ressemblerait ainsi au script suivant :

```
Require Arith.

Lemma card_nat: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
Proof.
  Red;Intro H.
  Elim H;Intros x Hx.
  Elim Hx;Intros y Hy.
  Elim (Hy (0));Elim (Hy (1));Elim (Hy (2));Intros.
  Cut (0)=(1);[Discriminate|Apply trans_equal with x;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with x;Auto].
  Cut (0)=(2);[Discriminate|Apply trans_equal with x;Auto].
  Cut (1)=(2);[Discriminate|Apply trans_equal with y;Auto].
  Cut (1)=(2);[Discriminate|Apply trans_equal with x;Auto].
  Cut (0)=(2);[Discriminate|Apply trans_equal with y;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with y;Auto].
  Cut (0)=(1);[Discriminate|Apply trans_equal with y;Auto].
Save.
```

Ce script suit exactement notre schéma de preuve formelle. Toutefois, on aimerait bien pouvoir automatiser la dernière partie de la preuve, où il faut utiliser la transitivité (puisqu'il s'agit uniquement de repérer, dans les hypothèses, deux égalités avec le même membre gauche). Malheureusement, même si cette automatisation semble tout à fait facile à réaliser, aucune tactique primitive n'est en mesure de s'en charger, et les tacticals ne sont pas assez puissants pour le faire. Ainsi, face à une telle preuve, l'utilisateur a deux choix : faire la preuve à la main (comme ici), ou écrire sa propre tactique en Objective Caml (comme dans la section 6.2).

Cependant, il semble un peu excessif de devoir utiliser Objective Caml pour une automatisation très ponctuelle comme celle-là. C'est essentiellement dû au fait que l'interfaçage avec Objective Caml est trop lourd par rapport au résultat que l'utilisateur veut obtenir. Ainsi, de manière générale, un langage complet de programmation n'est pas *forcément* adapté pour automatiser de petites parties de preuves, et, de fait, il semble y avoir un fossé entre le langage de preuves et le langage utilisé pour écrire les tactiques.

Ainsi, nous voulons proposer, dans le contexte du système Coq, l'idée d'un langage intermédiaire, intégré au prouveur et moins puissant qu'un langage Turing-complet pour écrire les tactiques, qui est capable de traiter les petites parties de preuves que l'on peut vouloir automatiser localement. Ce langage, que nous appellerons $\mathcal{L}_{tac}$, est supposé être une sorte de juste milieu, où il est possible de mieux profiter à la fois du langage habituel de Coq et de quelques caractéristiques du métalangage complet.

6.4 Définition de $\mathcal{L}_{tac}$

Dans le langage de preuves de Coq, le seul moyen de combiner les tactiques primitives est d'utiliser des tacticals. C'est un petit ensemble d'opérateurs prédéfinis, dont la liste se trouve en figure 6.1. Comme on l'a vu précédemment, dans la section 6.3, avec l'exemple sur la cardinalité de l'ensemble des entiers naturels, aucun tactical n'est approprié pour automatiser cette petite preuve. En fait, on aimerait pouvoir faire du filtrage sur les termes, et même mieux, sur les contextes de preuves (les buts).

tac_1 ;tac_2	Applique tac_1 et tac_2 à tous les sous-buts
tac ; [tac_1 | ... | tac_i | ... | tac_n]	Applique tac et tac_i au i-ème sous-but
tac_1 `Orelse` tac_2	Applique tac_1 ou tac_2 si tac_1 échoue
`Do` n tac	Applique tac n fois
`Repeat` tac	Applique tac jusqu'à ce qu'il échoue
`Try` tac	Applique tac et n'échoue pas s'il échoue
`First` [tac_1 | ... | tac_i | ... | tac_n]	Applique la première tac_i qui n'échoue pas
`Solve` [tac_1 | ... | tac_i | ... | tac_n]	Applique la première tac_i qui résout
`Idtac`	Laisse le but inchangé
`Fail`	Échoue toujours

FIG. 6.1 – Les tacticals de Coq.

Ainsi, l'idée de $\mathcal{L}_{tac}$ est de fournir un petit noyau fonctionnel avec de la récursion, de l'ordre supérieur et des opérateurs de filtrage, à la fois pour les termes et les contextes de preuves, afin de pouvoir manipuler le processus de preuve. La syntaxe de ce langage est donné, en utilisant une notation à la BNF, par l'entrée <*expr*> de la figure 6.2, où les entrées <*nat*>, <*ident*>, <*term*>, <*primitive-tactic*> et <*red-tactic*> représentent respectivement les entiers naturels, les identificateurs, les termes Coq (du CCI), les tactiques primitives (`Intro`, `Cut`, ...) et les tactiques de réduction (`Compute`, `Simpl`, ...). Dans l'entrée <*term*>, il peut y avoir des variables spécifiques comme `?n`, où `n` est un <*nat*>, ou `?`, qui sont des métavariables pour le filtrage. `?n` permet de garder les instantiations et d'effectuer des contraintes (nous verrons que l'on peut faire du filtrage non-linéaire), tandis que `?` est utilisé lorsque l'on n'est pas intéressé par ce qui a été filtré.

$\mathcal{L}_{tac}$ peut aussi être utilisé dans des définitions toplevel, pouvant être appelées ou utilisées ultérieurement, et dont l'entrée <*top*> de la figure 6.3 indique la syntaxe. Il y a deux sortes de définitions : les `Tactic Definition`'s, qui définissent des tactiques, et les `Meta Definition`'s, qui définissent d'autres types d'expressions, comme des entiers naturels, des termes, ...

6.5 Retour sur l'exemple

Avant de donner la sémantique complète de $\mathcal{L}_{tac}$, revenons sur l'exemple de la section 6.3, où l'on veut montrer que l'ensemble des entiers naturels possède plus que deux éléments. Comme on a pu le voir avec la preuve Coq, après les éliminations des $\vee$ (tactique `Elim`), il reste huit cas, qui sont résolus par des instructions très similaires, où seuls changent la coupure d'égalité et le terme utilisé pour appliquer la transitivité. Comme nous savons que cette égalité, disons $a = b$, est telle que les égalités $z = a$ et $z = b$ sont en hypothèses, cela serait facile d'automatiser cette partie, si on pouvait manipuler le contexte de la preuve. Cela peut être fait en utilisant $\mathcal{L}_{tac}$, et, en particulier, la construction `Match Context` :

```
Require Arith.

Lemma card_nat: ~(EX x:nat|(EX y:nat|(z:nat)(x=z)\/(y=z))).
```

```
<expr>           ::= <expr> ; <expr>
                  |  <expr> ; [ (<expr> |)* <expr> ]
                  |  <pre-atom>

<pre-atom>       ::= <expr> <expr>+
                  |  <atom>

<atom>           ::= Fun <input-fun>+ -> <expr>
                  |  Let (<let-clauses> And)* <let-clauses> In <expr>
                  |  Rec <rec-clause>
                  |  Rec (<rec-clause> And)* <rec-clause> In <expr>
                  |  Match Context With (<context-rule> |)* <context-rule>
                  |  Match <term> With (<match-rule> |)* <match-rule>
                  |  ( <expr> )
                  |  ( <expr> <expr>+ )
                  |  <atom> Orelse <atom>
                  |  Do (int | <ident>) <atom>
                  |  Repeat <atom>
                  |  Try <atom>
                  |  Progress <atom>
                  |  First [ (<expr> |)* <expr> ]
                  |  Solve [ (<expr> |)* <expr> ]
                  |  Idtac
                  |  Fail [<nat>]
                  |  <primitive-tactic>
                  |  <arg>

<input-fun>      ::= <ident>
                  |  ()

<let-clauses>    ::= <ident> = <expr>

<rec-clause>     ::= <ident> <input-fun>+ -> <expr>

<context-rule>   ::= [ (<context-hyps> ;)* <context-hyps> |- <pattern> ] ->
                     <expr>
                  |  [ |- <pattern> ] -> <expr>
                  |  _ -> <expr>

<context-hyps>   ::= <ident> : <pattern>
                  |  _ : <pattern>

<match-rule>     ::= [ <pattern> ] -> <expr>
                  |  _ -> <expr>

<pattern>        ::= <term>
                  |  [<ident>][ <term> ]

<arg>            ::= ()
                  |  <nat>
                  |  <ident>
                  |  ?<nat>
                  |  Eval <red-tactic> in <term>
                  |  Inst <ident>[ <term> ]
                  |  '<term>
```

FIG. 6.2 – Syntaxe de $\mathcal{L}_{tac}$.

<top>	::=	*<flag-def>* `Definition` *<ident>* *<input-fun>** `:=` *<expr>*
	|	`Recursive` *<flag-def>* `Definition` (*<trec-clauses>* `And`)* *<trec-clauses>*
<flag-def>	::=	`Tactic` | `Meta`
<trec-clauses>	::=	*<ident>* *<input-fun>*$^+$ `:=` *<expr>*

FIG. 6.3 – Définitions toplevel de $\mathcal{L}_{tac}$.

```
 4  Proof.
 5    Red;Intro H.
 6    Elim H;Intros x Hx.
 7    Elim Hx;Intros y Hy.
 8    Elim (Hy (0));Elim (Hy (1));Elim (Hy (2));Intros;
 9      Match Context With
10      | [_:?1=?2;_:?1=?3 |- ?] ->
11        Cut ?2=?3;[Discriminate|Apply trans_equal with ?1;Auto].
12  Save.
```

Des lignes 1 à 8, la preuve est complètement similaire à la version précédente. La différence se situe en lignes 9 à 11, où l'on utilise la construction `Match Context`, pour manipuler le contexte de preuve. La syntaxe du motif, en ligne 10, est isomorphe à celle d'un séquent, où ce qui est à gauche de `|-` constitue les hypothèses et ce qui à droite, la conclusion. Ici, le motif filtre bien ce que l'on voulait, à savoir deux égalités avec le même membre gauche (le filtrage est non-linéaire, et on peut donc utiliser plusieurs fois le même numéro de métavariable pour ajouter des contraintes). La partie droite de la règle du `Match Context`, en ligne 11, peut alors utiliser l'instantiation des métavariables numérotées provenant du filtrage. Au passage, on peut noter que, pour les noms d'hypothèses, nous avons utilisé `_`, puisque nous n'en avons pas besoin dans la partie droite de la règle. De même, pour la conclusion, on utilise `?`, qui filtre tout terme (et l'instantiation ne pourra pas être récupérée dans la partie droite de la règle).

On peut remarquer que la preuve est significativement plus courte[3], et c'est d'autant plus vrai, lorsque l'on rajoute des cas (avec trois, quatre, ... éléments). De plus, le travail est moins fastidieux que dans le cas de la preuve réalisée manuellement, et le script peut être écrit sans l'aide du toplevel. Cela signifie que cela va plus dans le sens d'un style de preuves en mode batch.

À titre indicatif, on peut aussi noter que l'exemple de la section 6.2, codé en Objective Caml, où il s'agissait d'éliminer tous les connecteurs $\wedge$ en hypothèses, s'écrit de manière complètement triviale en utilisant $\mathcal{L}_{tac}$:

```
Tactic Definition ElimAnds :=
  Repeat
    Match Context With
    | [id:?/\? |- ?] -> Elim id;Intros;Clear id.
```

[3] À ce sujet, on peut voir que le filtrage non-linéaire permet de résoudre le problème en un seul motif, au lieu de deux motifs successifs.

Par rapport à la version Objective Caml, le code est ici considérablement plus court. Il est, par ailleurs, plus lisible, puisque ne nécessitant, entre autres, pas de connaissances particulières en Objective Caml, et surtout, sur l'implantation du système Coq. Comme conséquence de cette bonne lisibilité, le code sera facilement maintenable, et ce, d'autant plus qu'il est complètement portable, dans la mesure où il ne fait pas appel, contrairement à la version en Objective Caml, à des fonctions spécifiques de l'implantation du système. En effet, comme $\mathcal{L}_{tac}$ fait plutôt partie du langage objet de Coq, les évolutions de ce langage se feront de manière la plus conservative possible.

6.6 Sémantique

Nous allons donner ici une sémantique informelle de $\mathcal{L}_{tac}$. Nous distinguerons deux étapes : l'évaluation, qui permet d'associer, à une expression, une valeur (pas forcément une tactique), et l'application des valeurs de tactiques (à des buts), qui s'effectue dans le mode d'édition de preuves.

6.6.1 Valeurs

Les valeurs de $\mathcal{L}_{tac}$ sont données par la figure 6.4. On appellera valeur de tactique, toute expression de l'entrée *<vexpr>*, privée de `Fun`, `Rec`, et *<varg>*.

```
<vexpr>   ::= <vexpr> ; <vexpr>
           |  <vexpr> ; [ (<vexpr> |)* <vexpr> ]
           |  <vatom>

<vatom>   ::= Fun <input-fun>+ -> <expr>
           |  Rec <rec-clause>
           |  Rec (<rec-clause> And)* <rec-clause> In <expr>
           |  Match Context With (<context-rule> |)* <context-rule>
           |  ( <vexpr> )
           |  <vatom> Orelse <vatom>
           |  Do (<int> | <ident>) <vatom>
           |  Repeat <vatom>
           |  Try <vatom>
           |  Progress <vatom>
           |  First [ (<vexpr> |)* <vexpr> ]
           |  Solve [ (<vexpr> |)* <vexpr> ]
           |  Idtac
           |  Fail [<nat>]
           |  <primitive-tactic>
           |  <varg>

<varg>    ::= ()
           |  <nat>
           |  <ident>
```

FIG. 6.4 – Valeurs de $\mathcal{L}_{tac}$.

6.6.2 Évaluation

Définitions locales

Les définitions locales sont de la forme suivante :

Let x_1 = e_1
And x_2 = e_2
$\vdots$
And x_n = e_n In
t

Les expressions e_i sont simultanément évaluées en v_i, puis t est évalué en substituant v_i à chaque occurence de x_i, pour $i = 1, \ldots, n$. Il n'y a aucune dépendance entre les différents x_i et e_i.

Application

Une application est une expression de la forme :

(f a_1 ... a_n)

f est évaluée en v et les a_i en v_i, pour $i = 1, \ldots, n$. Si v est `Fun` ou `Rec`, alors le corps est évalué en substituant les v_i aux paramètres formels. Pour les clauses récursives, les corps sont substitués paresseusement (lorsque l'identificateur à évaluer est le nom d'une clause récursive).

Message d'erreur :

`Illegal tactic application`
Si v n'est ni `Fun`, ni `Rec`, ou n est supérieur strictement au nombre de paramètres formels de `Fun` ou `Rec`.

Filtrage sur les termes

Le filtrage sur les termes se définit comme suit :

Match t With
| [p_1] -> e_1
| [p_2] -> e_2
$\vdots$
| [p_n] -> e_n
| _ -> e_{n+1}

Si p_1 est de la forme [p_1'] ou C[p_1'], on essaie de trouver un sous-terme s de t, filtré par p_1' (unification non-linéaire du premier ordre), et tel que toute variable libre de s ne soit pas liée dans t. Si un tel sous-terme existe, alors e_1 est évaluée en substituant les instantiations du filtrage aux métavariables, et éventuellement le contexte du filtrage (un terme de la forme $\lambda x.t[x/s]$) à C. Si le filtrage échoue ou si l'évaluation de e_1 échoue, alors p_2 est essayé et ainsi de suite. Si aucun p_i, avec $i = 1, \ldots, n$, ne filtre t ou toutes les évaluations des expressions e_i lorsque p_i filtre t échouent, alors la dernière clause est utilisée et e_{n+1} est évaluée.

Si p_1 est de la forme t_1, alors on tente de filtrer t avec t_1. Si t et t_1 sont unifiables, on évalue e_1 en substituant les instantiations du filtrage aux métavariables. Si l'unification échoue ou si l'évaluation de e_1 échoue, alors, de même que précédemment, on essaie p_2 et etc...

Message d'erreur :

`No matching clauses for Match`
Aucun motif ne peut être utilisé.

Termes et métavariables

Un terme t s'évalue en t si et seulement si il est typable dans le contexte Δ (contexte global) ou $\Delta[\Gamma]$ (contextes global et local, dans le mode d'édition de preuves).
Une métavariable `?n` s'évalue toujours sur l'erreur :

`Metavariable n is unbound`

Réduction de terme

Une réduction de terme se définit comme suit :

`Eval` *red* `in` t

Le terme t est évalué, puis on lui applique la réduction *red*.

Instantiation

Une instantiation est de la forme :

`Inst` C`[`t`]`

Si C est de la forme $\lambda x.t'$, on évalue le terme $t'[x\backslash t]$.

6.6.3 Application des valeurs de tactiques

Séquence

Une séquence est une expression de la forme :

e_1 `;` e_2

e_1 and e_2 sont évaluées en v_1 et v_2, qui doivent être des valeurs de tactiques. v_1 est alors appliquée, puis v_2 est appliquée aux sous-buts générés. La séquence associe à gauche.

Séquence n-aire

Une autre forme de séquence est :

e_0 `; [` e_1 `|` ... `|` e_n `]`

Les e_i sont évaluées en v_i, qui doivent être des valeurs de tactiques, pour $i = 0, \ldots, n$. v_0 est appliquée et v_i est appliquée au $i$ème sous-but généré.

Message d'erreur :

`Wrong number of tactics`
Si l'application de v_0 ne génère pas exactement n sous-buts.

Branchement

Un branchement est une expression de la forme :

e_1 `Orelse` e_2

e_1 and e_2 sont évaluées en v_1 et v_2, qui doivent être des valeurs de tactiques. v_1 est appliquée et, si cela échoue ou ne progresse pas (laisse le but inchangé), alors v_2 est appliquée. Le branchement associe à gauche.

Boucle `Do`

La boucle `Do` se définit comme suit :

`Do` n e

e est évaluée en v, qui doit être une valeur de tactique. v est appliquée n fois. En supposant que $n > 1$, après la première application de v, v est appliquée, au moins une fois, aux sous-buts générés et ainsi de suite. Il échoue, une des applications de v échoue.

Boucle `Repeat`

La boucle `Repeat` a la forme suivante :

`Repeat` e

e est évaluée en v, qui doit être une valeur de tactique. v est appliquée jusqu'à ce que l'application échoue. En supposant que la première application de v n'échoue pas, v est, de nouveau, appliquée aux sous-buts générés. La boucle s'arrête lorsque l'application échoue pour tous les sous-buts générés. `Repeat` n'échoue jamais.

Rattrapage d'erreurs

On peut rattraper les erreurs de tactiques comme suit :

`Try` e

e est évaluée en v, qui doit être une valeur de tactique, et v est appliquée. Si l'application de v échoue avec le niveau d'échec n, soit $n = 0$ et l'erreur est rattrapée, laissant le but inchangé, soit $n > 0$ et l'erreur est transmise avec le niveau $n - 1$. Toutes les tactiques primitives échouent avec un niveau 0.

Progression

On peut imposer qu'une tactique fasse évoluer le but avec l'expression :

`Progress` e

e est évaluée en v, qui doit être une valeur de tactique, et v est appliquée. Si l'application de v laisse le but inchangé, il échoue avec le niveau d'échec 0.

Message d'erreur :

`Failed to progress`
Si l'application de v laisse le but inchangé.

Première tactique à s'appliquer

On peut considérer la première tactique à s'appliquer (c'est-à-dire qui n'échoue pas) parmi une liste de tactiques :

`First [` e_1 `| ... |` e_n `]`

Les e_i sont évaluées en v_i, qui doivent être des valeurs de tactiques, pour $i = 1, \ldots, n$. En supposant que que $n > 1$, il applique v_1, si cela marche, il arrête, sinon il essaie d'appliquer v_2 et ainsi de suite.

Message d'erreur :

`No applicable tactic`
Si aucun des v_i ne peut être appliquée.

Première tactique à résoudre

On peut considérer la première tactique qui résoud (c'est-à-dire qui ne génère aucun sous-but) parmi une liste de tactiques :

`Solve [` e_1 `| ... |` e_n `]`

Les e_i sont évaluées en v_i, qui doivent être des valeurs de tactiques, pour $i = 1, \ldots, n$. En supposant que que $n > 1$, il applique v_1, si cela résoud, il arrête, sinon il essaie d'appliquer v_2 et ainsi de suite.

Message d'erreur :

`Cannot solve the goal`
Si aucun des v_i ne peut résoudre.

Identité

La tactique identité est :
`Idtac`

Elle laisse le but inchangé (mais elle apparaît dans le script de preuve).

Échec

La tactique d'échec est :

`Fail` n

Il échoue avec le niveau n et laisse le but inchangé. Il n'apparaît pas dans le script d'erreur et peut être rattrapé par `Try`. `Fail` est équivalent à `Fail 0`.

Filtrage sur les contextes de preuves

Le filtrage sur les contextes de preuves se fait au moyen de l'expression suivante :

```
Match Context With
| [hp_{1,1} ;... ;hp_{1,m_1} |-p_1]   -> e_1
| [hp_{2,1} ;... ;hp_{2,m_2} |-p_2]   -> e_2
⋮
| [hp_{n,1} ;... ;hp_{n,m_n} |-p_n]   -> e_n
| _                                   -> e_{n+1}
```

Si chaque motif d'hypothèse $hp_{1,i}$, avec $i = 1, ..., m_1$ filtre (unification non-linéaire du premier ordre) une hypothèse du but et si p_1 filtre la conclusion du but, alors e_1 est évaluée en v_1 en substituant les métavariables (éventuellement aussi les contextes des filtrages sur des sous-termes) et les noms d'hypothèses effectifs aux noms d'hypothèses formels apparaissant dans les motifs d'hypothèses. Si v_1 est une valeur de tactique, alors elle est appliquée. Si l'application échoue avec un niveau $n > 0$, alors on échoue avec le niveau $n-1$. Si l'application échoue avec le niveau 0, alors une autre combinaison des hypothèses est essayée avec le même motif. S'il n'y a plus d'autres combinaisons d'hypothèses, alors le second motif est essayé et ainsi de suite. Si l'avant-dernier motif échoue, alors on évalue e_{n+1}.

`Message d'erreur :`

`No matching clauses for Match Context`
Aucun motif ne peut être utilisé.

6.6.4 Définitions toplevel

Les définitions toplevel sont effectuées comme suit :

`Tactic Definition` x `:=` e
`Meta Definition` x `:=` e

e est évalué en v. S'il s'agit d'une `Tactic Definition`, v doit être une valeur de tactique, et, dans le cas d'une `Meta Definition`, v ne doit pas être une valeur de tactique. Ensuite, v est associé à x, et tout script de preuve est évalué en substituant v à x.

On peut définir les définitions fonctionnelles par :

`Tactic Definition` x a_1 ... a_n `:=` e

`Meta Definition` x a_1 ... a_n `:=` e

qui sont du sucre syntaxique pour :

`Tactic Definition` x `:= Fun` a_1 ... a_n `->` e
`Meta Definition` x `:= Fun` a_1 ... a_n `->` e

Ces définitions sont alors traitées comme ci-dessus.
Enfin, les définitions de fonctions mutuellement récursives sont possibles avec :

`Recursive Tactic Definition`
x_1 $a_{1,1}$ ... a_{1,m_1} `:=` e_1
`And` x_2 $a_{2,1}$ ... a_{2,m_2} `:=` e_2
...
`And` x_n $a_{n,1}$ ... a_{n,m_n} `:=` e_n

Ce bloc est un ensemble de définitions fonctionnelles (utilisant la même notation que ci-dessus) et les autres scripts sont évalués comme d'habitude, excepté que les substitutions sont effectuées de manière paresseuse (quand un identificateur est le nom d'une définition récursive).

6.7 Implantation

$\mathcal{L}_{tac}$ a été implanté et intégré à la version V7 du système `Coq`. Pour cela, il a fallu faire des choix par rapport au code existant. Tout d'abord, nous avons conservé un langage interprété. En effet, il n'est pas clair que l'on puisse économiser un temps significatif dans l'exécution de scripts compilés, en général exécutés une seule fois, surtout si l'on considère le coût du temps de compilation. Par rapport à l'ancien noyau d'interprétation de tactiques[4] (voir fichiers `proofs/tacinterp.ml[i]`), des changements importants ont été faits dans la fonction principale d'interprétation, en ajoutant, par exemple, les nouvelles constructions, que l'on a vues précédemment (voir figure 6.2). Aussi, pour pouvoir gérer les substitutions venant des variables abstraites (`Fun`) et des métavariables (`Match Context` et `Match`), les arguments de tactiques sont interprétées dans la fonction principale. Ainsi, les tactiques prennent désormais des arguments déjà interprétés, plutôt que des AST's (Abstract Syntax Trees) venant de l'analyse syntaxique. Pour être extensible, il est possible d'associer dynamiquement des fonctions d'interprétation à des nœuds d'AST's spécifiques.

6.8 D'autres exemples

6.8.1 Inégalités entre constantes réelles entières

Un autre exemple, un peu moins trivial que précédemment, est de montrer automatiquement que deux constantes réelles entières sont distinctes. De telles preuves sont généralement assez fastidieuses, puisque, dans le cas de nombres réels axiomatisés (comme ceux de `Coq`) et contrairement aux entiers naturels, par exemple, qui sont définis inductivement et pour lesquels l'application de la tactique `Discriminate` (qui permet de montrer que deux constructeurs distincts d'un type inductif sont effectivement distincts) est suffisante pour

[4]Des précédentes versions V6.

résoudre ces preuves, il n'y a pas d'autre alternative que de combiner les différents lemmes ou axiomes existants relatifs à la bibliothèque des nombres réels.

Une autre conséquence de l'axiomatisation des nombres réels est qu'il faut définir les constantes réelles entières en fonction d'opérateurs, et il y a un choix de représentation à faire. Dans la bibliothèque actuelle, 0 est représenté par 0 (constante faisant partie de l'axiomatisation), une constante strictement positive n est représentée comme $1+(1+\ldots(1+1))$, avec n fois 1 (constante faisant également partie de l'axiomatisation), et une constante strictement négative est représentée comme $-(1+(1+\ldots(1+1)))$, avec n fois 1. Pour rendre la tactique un peu plus générale, nous nous proposons de traiter aussi les inégalités entre expressions réelles pouvant se ramener à des inégalités entre constantes réelles entières. Ainsi, nous considérerons les expressions réelles formées uniquement de 0, 1, $+$, $-$ (unaire et binaire) et $*$.

Pour faire la preuve, on se ramènera d'abord à deux constantes à gauche et à droite, $c_1 <> c_2$, que l'on transformera en $c_1 - c_2 <> 0$, en simplifiant $c_1 - c_2$ en une nouvelle constante c. Si $c > 0$, alors $c <> 0$ devient $c > 0$ à montrer. Si $c = 1$, on conclut (lemme de la bibliothèque), sinon on a $c = 1 + c'$ et on utilise la transitivité de $>$ pour montrer que $c' > 0$. Si $c < 0$, on se ramène au cas positif, puisque si $-r <> 0$ alors $r <> 0$. L'algorithme de cette preuve est donc assez simple. Utiliser Objective Caml pour le coder est certainement tout à fait excessif, et $\mathcal{L}_{tac}$ semble être ici bien plus approprié.

Dans un premier temps, définissons la tactique `Isrealint`, qui teste si l'expression réelle est bien une constante réelle entière, ou peut se ramener à une constante réelle entière :

```
1   Recursive Tactic Definition Isrealint trm:=
2     Match trm With
3     | [``0``] -> Idtac
4     | [``1``] -> Idtac
5     | [``?1+?2``] -> Isrealint ?1;Isrealint ?2
6     | [``?1-?2``] -> Isrealint ?1;Isrealint ?2
7     | [``?1*?2``] -> Isrealint ?1;Isrealint ?2
8     | [``-?1``] -> Isrealint ?1
9     | _ -> Fail.
```

|

Cette tactique ne fait rien (rend `Idtac`), si `trm` est une constante réelle entière, ou échoue (rend `Fail`), sinon. Cela se fait en déstructurant `trm`, avec un `Match`, en ligne 2. En lignes 3 et 4, si on a 0 ou 1, on rend `Idtac`. Des lignes 5 à 8, on passe simplement au contexte suivant $+$, $-$ (unaire et binaire) et $*$. On peut noter que, ce passage au contexte occasionnant des appels récursifs de `Isrealint`, la tactique est déclarée avec `Recursive`, en ligne 1. Enfin, en ligne 9, si on a autre chose que 0, 1, $+$, $-$ ou $*$, ce n'est ni une constante réelle entière, ni une expression réelle pouvant se ramner à une constante réelle entière, et on échoue avec `Fail`.

Remarquons, au passage, que les nombres réels sont dotés d'une syntaxe particulière, qui permet de les exprimer naturellement. Pour en profiter, il suffit de parenthéser l'expression réelle par “...“. En particulier, on peut écrire “3“, pour $1 + (1 + 1)$, et on peut également utiliser des métavariables, comme on peut le voir en lignes 5 à 8.

Écrivons maintenant la tactique `Sup0`, qui montre que $c > 0$, si c est une constante réelle entière strictement positive :

```
11  Recursive Tactic Definition Sup0 :=
12    Match Context With
```

```
13      | [ |- ``1>0`` ] -> Unfold Rgt;Apply Rlt_R0_R1
14      | [ |- ``1+?1>0`` ] ->
15        Apply (Rgt_trans ``1+?1`` ?1 ``0``);
16          [Pattern 1 ``1+?1``;Rewrite Rplus_sym;Unfold Rgt;
17           Apply Rlt_r_r_plus_R1|Sup0].
```

Cette tactique fonctionne en filtrant la conclusion du but courant, au moyen du `Match Context` de la ligne 12. En ligne 13, c'est le cas de base où $1 > 0$, que l'on résout directement. En ligne 14, on est dans le cas où $1 + c > 0$. On applique le lemme de transitivité de $>$ (`Rgt_trans`) avec $1 + c$, c et 0. $1 + c > c$ est montré directement, et il reste alors à montrer que $c > 0$, ce qui est fait par l'appel récursif de la ligne 17.

Nous pouvons enfin donner le code de la tactique principale, que nous appellerons `DiscrR` (par similitude avec la tactique `Discriminate`) :

```
19    Tactic Definition DiscrR :=
20      Try Match Context With
21      | [ |- ~(?1==?2) ] ->
22        Isrealint ?1;Isrealint ?2;
23        Apply Rminus_not_eq; Ring ``?1-?2``;
24          (Match Context With
25          | [ |- [``-1``] ] ->
26            Repeat Rewrite <- Ropp_distr1;Apply Ropp_neq
27          | _ -> Idtac);Apply Rgt_not_eq;Sup0.
```

En ligne 21, on filtre l'inégalité $c_1 <> c_2$, puis, en ligne 22, on teste si les deux membres sont bien des constantes réelles entières ou des expressions réelles pouvant se ramener à des constantes réelles entières, avec `Isrealint`. En ligne 23, on transforme le but en $c1 - c2 <> 0$, avec le lemme `Rminus_not_eq`, et on transforme $c1 - c2$ en une constante réelle entière, au moyen de la tactique `Ring` (qui associe à droite pour $+$, comme attendu pour les constantes). En ligne 25, si l'expression est négative, c'est-à-dire s'il existe une occurence de -1 dans l'expression (on peut remarquer l'utilisation d'un motif pour filtrer sur les sous-termes), on la transforme en constante négative, avec `Ropp_distr1`, pour *sortir* les $-$ unaires et transformer l'expression $-1 + (-1 + \ldots (-1 + (-1)))$ en $-(1 + (1 + \ldots (1 + 1)))$, puisque `Ring` distribue les $-$ unaires. Ensuite, on transforme l'expression $-c <> 0$ en $c <> 0$, avec `Ropp_neq`. En ligne 27, on a une constante positive et il n'y a rien à faire, on renvoie donc `Idtac`. Enfin, il reste à transformer le but $c <> 0$ en $c > 0$, et à appeler `Sup0`. Le `Try` de la ligne 20 permet de ne pas échouer si les deux membres ne correspondent pas à des constantes réelles entières (même après simplification), ou si les deux membres sont égaux.

Voici deux exemples d'application de `DiscrR` :

```
Coq < Goal ``-2<>5``.
1 subgoal

  ============================
   `` -2 <> 5``

Unnamed_thm < DiscrR.
Subtree proved!

Coq < Goal ``2*(-3)+1<>7-4``.
1 subgoal
```

```
  ============================
  ‘‘2* -3+1 <> 7-4‘‘

Unnamed_thm < DiscrR.
Subtree proved!
```

La tactique DiscrR fait actuellement partie de la bibliothèque des nombres réels, et son code correspond exactement à celui que nous avons décrit. Il y a d'autres tactiques de cette bibliothèque, qui sont aussi écrites en utilisant $\mathcal{L}_{tac}$, comme SplitAbsolu (qui sépare la valeur absolue $|x|$ en deux cas, $x \geq 0$ et $x < 0$) et SplitMult (qui sépare l'expression $a_1 * a_2 * \ldots * a_n <> 0$ en $a_1 <> 0$, $a_2 <> 0$, ... et $a_n <> 0$). À titre indicatif, SplitAbsolu avait été codée, à l'origine, en Objective Caml, et le code faisait environ une centaine de lignes, alors que le code actuel en $\mathcal{L}_{tac}$ fait une dizaine de lignes. Ce gain de place est assez significatif, et nous verrons, dans le chapitre 7, un exemple où le rapport de taille est encore plus important.

6.8.2 Permutations sur des listes closes

Nous nous proposons de considérer le problème des permutations sur des listes closes, où il s'agit de montrer, étant données deux listes closes l_1 et l_2, que l_1 est une permutation de l_2 et *vice-versa*. Typiquement, on peut rencontrer ce problème quand on veut montrer qu'une liste est triée, et il peut être ennuyant de faire la preuve à la main, quand nous savons que cela peut être fait automatiquement. Là encore, Objective Caml ne semble pas le plus approprié, compte-tenu de la difficulté de ce que l'on veut résoudre, et nous allons voir comment $\mathcal{L}_{tac}$ s'avère plus adéquat.

Pour ce faire, définissons d'abord le prédicat de permutation sur les listes :

```
Inductive permut [A:Set] : (list A)->(list A)->Prop :=
| permut_cons: (a:A)(l0,l1:(list A))
               (permut A l0 l1)->(permut A (cons a l0) (cons a l1))
| permut_append: (a:A)(l:(list A))
                 (permut A (cons a l) (l^(cons a (nil A))))
| permut_refl: (l:(list A))(permut A l l)
| permut_sym: (l1,l2:(list A))(permut A l1 l2)->(permut A l2 l1)
| permut_trans: (l0,l1,l2:(list A))
                (permut A l0 l1)->(permut A l1 l2)->(permut A l0 l2).
```

où ^ est du sucre syntaxique pour app, la fonction de concaténation sur les listes.

Nous pouvons maintenant écrire la tactique en utilisant $\mathcal{L}_{tac}$:

```
Recursive Tactic Definition Permut n :=
  Match Context With
  | [|- (permut ? ?1 ?1)] -> Apply permut_refl
  | [|- (permut ? (cons ?1 ?2) (cons ?1 ?3))] ->
    Let newn = Eval Compute in (length ?2) In
    Apply permut_cons;Permut newn
  | [|- (permut ?1 (cons ?2 ?3) ?4)] ->
    (Match Eval Compute in n With
    | [(1)] -> Fail
    | _ ->
```

```
11        Let l = (?3^(cons ?2 (nil ?1))) In
12        Apply (permut_trans ?1 (cons ?2 ?3) l ?4);
13          [Apply permut_append|Compute;Permut '(pred n)]).
14
15  Tactic Definition PermutProve :=
16    Match Context With
17    | [|- (permut ? ?1 ?2)] ->
18      (Match Eval Compute in (length ?1)=(length ?2) With
19      | [?1=?1] -> Permut ?1).
```

La tactique est codée au moyen de deux définitions toplevel, `Permut` et `PermutProve`. La tactique principale (à appeler) est `PermutProve`, qui est supposée résoudre des buts de la forme ... $\vdash$ (`permut` l_1 l_2), où l_1 et l_2 sont des expressions de listes closes.

En ligne 17, la tactique `PermutProve` récupère d'abord les deux listes, puis, en ligne 18, elle calcule les longueurs des deux listes. Si les deux listes ont bien la même longueur, elle appelle, en ligne 19, `Permut` avec cette longueur (elle échoue dans le cas contraire). La tactique `Permut` étudie les différents cas du prédicat `permut`, qui lui permettent de résoudre. En ligne 3, si les deux listes sont égales, elle conclut. Sinon, en ligne 4, si les deux listes ont le même élément de tête, on calcule la longueur de la queue de la liste, en ligne 5, avant de se rappeler récursivement avec la queue de la liste, en ligne 6. Enfin, en ligne 7, si les deux listes ont des éléments de tête distincts, on met l'élément de tête de la première liste en queue de cette liste, en ligne 11, si c'est possible, c'est-à-dire, si on n'a pas atteint le nombre maximal de rotations, représenté par le compteur `n` et calculé en ligne 8. Sinon, on échoue explicitement en ligne 9.

On peut remarquer que pour vérifier que toutes les rotations ont été faites, on utilise un compteur (initialisé avec la longueur n de la liste, et décrémenté, jusqu'à 1, exclu, car, pour une liste de longueur n, on peut faire exactement $n-1$ rotations, pour générer au plus n listes distinctes), qui est un entier naturel de `Coq` (type `nat`). Dans la figure 6.2, on peut voir qu'il est possible d'utiliser des entiers naturels usuels, mais ils sont essentiellement utilisés dans les tactiques primitives, et, en particulier, ils ne permettent pas d'effectuer de calculs. Ainsi, un choix naturel est d'utiliser les structures de données de `Coq`, si bien que `Coq` se charge de faire les calculs (réductions) avec `Eval...in`, et nous pouvons récupérer les termes avec `Match`.

On peut maintenant tester `PermutProve`, mais, définissons avant, un peu de syntaxe pour les listes :

```
Grammar constr constr_list : constr :=
| nil [ "#" ] -> [(nil ?)]
| cons [ constr($c) constr_list($t) ] -> [(cons $c $t)]

with constr0 : constr :=
| begin [ "#" constr_list($l) ] -> [$l].

Syntax constr level 0:
| nil [ (nil $s) ] -> [ "##" ]
| cons [ (cons $a $t) ] -> [ [<hov 0> "#" $a:E (LIST $t)] ]
| nil_list [ << (LIST (APPLIST $n $s)) >> ] -> [ "#" ]
| cons_list [ << (LIST (APPLIST $c $e $t)) >> ] -> [ [1 0] $e:E (LIST $t) ].
```

Les commandes `Grammar` et `Syntax` permettent respectivment de modifier dynamiquement le parser et le pretty-printer de `Coq` (se reporter à [6], pour plus de détails sur ces commandes). Avec les lignes précédentes, nous sommes en mesure de saisir `#a b#`, pour le terme `(cons a (cons b (nil ?)))`, et il sera aussi affiché comme on l'a saisi.

Pour tester `PermutProve`, voici deux exemples utilisant respectivement des listes d'entiers naturels et d'entiers relatifs (le deuxième exemple correspond au pire des cas pour `PermutProve`, lorsque les deux listes sont inversées l'une par rapport à l'autre) :

```
Coq < Goal (permut nat #(2) (3) (1) (4) (0)# #(1) (2) (0) (4) (3)#).
1 subgoal

  ============================
   (permut nat #(2) (3) (1) (4) (0)# #(1) (2) (0) (4) (3)#)

Unnamed_thm < PermutProve.
Subtree proved!

Coq < Goal (permut Z #`-2` `-1` `0` `1` `2`# #`2` `1` `0` `-1` `-2`#).
1 subgoal

  ============================
   (permut Z #`-2` `-1` `0` `1` `2`# #`2` `1` `0` `-1` `-2`#)

Unnamed_thm < PermutProve.
Subtree proved!
```

6.9 Discussion

$\mathcal{L}_{tac}$ s'avère être un réel lien entre les tactiques primitives et le métalangage complet (Objective Caml), utilisé pour écrire des tactiques de taille importante ou, du moins, résolvant des problèmes non triviaux. En particulier, il traite les petites parties de preuves que l'on souhaite automatiser. On a pu voir que cette nouvelle couche de métalangage possédait quelques caractéristiques intéressantes :

– il est intégré au toplevel de Coq. L'utilisateur n'a besoin d'aucun compilateur spécifique, ni d'aucune spécification de l'implantation de Coq, pour écrire des tactiques dans ce langage. De plus, apprendre ce petit langage est certainement plus facile que de s'attaquer au manuel du langage d'implantation. Bien sûr, ces remarques doivent être considérées par rapport à de petites tactiques.
– la longueur du code est, en général, très courte, comparé aux mêmes preuves réalisées manuellement, et aussi par rapport aux mêmes automatisations codées en Objective Caml. On verra, dans le chapitre 7, que cette caractéristique est conservée lorsque l'on écrit des tactiques résolvant des problèmes plus complexes. Ainsi, les scripts sont plus compacts et beaucoup plus simples.
– les scripts sont plus lisibles. C'est le cas pour des petites preuves, mais même plus avec des tactiques plus importantes (voir chapitre 7).
– les scripts sont portables. En effet, on ne fait aucun appel direct à l'implantation du système, contrairement à des tactiques écrites en Objective Caml. On ne fait qu'utiliser une interface, dont on est en droit d'attendre qu'elle évoluera de manière conservative, quelles que soient les restructurations internes du code du système.
– les scripts sont plus maintenables, du fait de l'amélioration notable de la lisibilité. Par ailleurs, cette maintenance ne se fait qu'au niveau des spécifications, puisque, comme

on l'a dit précédemment, les scripts sont complètement portables et sont donc robustes aux évolutions internes du système.

Du point de vue de l'utilisateur, cela peut être un problème épineux de décider quel langage utiliser pour résoudre son problème. L'utilisateur doit savoir si le problème en question peut être codé avec $\mathcal{L}_{tac}$ ou non. Il n'y a pas de règles générales, mais on peut identifier plusieurs critères selon lesquelles Objective Caml doit être utilisé plutôt que $\mathcal{L}_{tac}$. Tout d'abord, $\mathcal{L}_{tac}$ n'est pas approprié pour des tactiques qui manipulent l'environnement. Par exemple, effectuer une recherche dans le contexte global est seulement possible en utilisant Objective Caml et certaines fonctions du code de Coq. Un autre indicateur, que $\mathcal{L}_{tac}$ ne convient pas, est l'utilisation des structures de données auxiliaires. Plus on utilise des structures de données, plus le problème est complexe, et plus le sera la tactique à construire. Comme on a pu le voir avec l'exemple de la section 6.8.2, concernant les permutations de listes closes, on peut utiliser des structures de données dans $\mathcal{L}_{tac}$, au moyen des structures de données de Coq[5], qui peuvent être manipulées par `Match` (et éventuellement `Match Context`), et le nombre de structures de données nécessaires est une bonne indication sur la difficulté de la tactique à écrire. De plus, si on est intéressé par les performances, il est préférable d'utiliser les structures de données d'Objective Caml, qui sont bien plus efficaces que celle de Coq. Il y a, par ailleurs, plus de bibliothèques implantant les structures de données usuelles en Objective Caml qu'en Coq, et cela peut être un argument décisif dans certains cas. Ainsi, en général, l'utilisation des structures de données doit être limité en $\mathcal{L}_{tac}$, et l'utilisateur doit faire des choix. Par exemple, l'utilisation des entiers naturels, dans l'exemple des permutations de listes closes, semble tout à fait raisonnable, et on peut considérer que c'est aussi le cas pour d'autres structures de données, comme les booléens ou les listes.

Malgré ces différents critères qui peuvent donner des pistes sur le choix à faire, il est toujours possible que l'utilisateur ait commencé son développement en $\mathcal{L}_{tac}$, et qu'il s'aperçoive, *a posteriori*, qu'il a besoin de caractéristiques plus profondes du système, ne pouvant pas être gérées par $\mathcal{L}_{tac}$. Dans ce cas, il doit naturellement passer en Objective Caml, mais il semble légitime qu'il ne perde pas le code déjà écrit, et, dans le chapitre 9, nous décrirons une interface pour utiliser du code $\mathcal{L}_{tac}$ en Objective Caml. Dans ce chapitre, nous verrons aussi comment il est possible d'avoir la fonctionnalité inverse, à savoir comment utiliser du code Objective Caml en $\mathcal{L}_{tac}$, ce qui rend le processus encore plus intéressant, puisque l'on a, dès lors, une réelle interaction entre les deux métalangages.

La situation entre Objective Caml et $\mathcal{L}_{tac}$ peut être résumée par le tableau suivant :

Interface

	Objective Caml	$\mathcal{L}_{tac}$
Code	Boîte noire	Dans les sources
Évaluation	Compilé	Interprété
Structures de données	ML	Coq
Gestion de l'environnement	Oui	Non

[5]Cela peut être vu comme un pas vers un système *bootstrappé*, puisqu'il est désormais possible d'écrire des tactiques utilisant plus intensivement le langage objet de Coq.

Chapitre 7

Automatisations non triviales en utilisant $\mathcal{L}_{tac}$

Comme nous l'avons vu précédemment, $\mathcal{L}_{tac}$ a pour objectif d'insérer une nouvelle couche de métalangage au niveau du langage de l'utilisateur, lui permettant ainsi d'ignorer les détails de l'implantation lorsqu'il s'agit d'élaborer de petites automatisations plutôt ponctuelles. Constitué essentiellement d'un petit noyau fonctionnel et d'opérateurs de filtrage sur les termes Coq ainsi que sur les contextes de preuves, $\mathcal{L}_{tac}$ peut également convenir à des automatisations plus conséquentes. Cette possibilité provient du fait qu'il est difficile de contrôler exactement le pouvoir d'automatisation d'un outil d'aide à la preuve. En effet, dans le cas de $\mathcal{L}_{tac}$, l'ajout de certaines primitives, telles que les opérateurs de filtrage avec des comportements très spécifiques (backtracking), est complètement nécessaire pour gérer les cas qui nous intéressaient mais ouvre, de même, très nettement le champ d'action dans la construction de nouvelles tactiques pouvant être considérées comme non triviales. Ceci doit, cependant, être considéré comme un bonus et non comme une motivation de $\mathcal{L}_{tac}$, dont l'objectif n'est pas de se substituer au métalangage complet de Coq (Objective Caml), mais de traiter de petites automatisations. Ce chapitre donne une description de quelques tactiques non triviales pouvant être élaborées au moyen de $\mathcal{L}_{tac}$.

7.1 Tautologies propositionnelles intuitionnistes

Une toute première application non triviale de $\mathcal{L}_{tac}$ a été de coder une procédure de recherche de preuve pour les tautologies propositionnelles intuitionnistes. Le calcul utilisé est le calcul des séquents sans contraction LJT de Roy Dyckhoff [28]. L'idée essentielle de ce calcul est basée sur le fait qu'en logique propositionnelle intuitionniste, on peut éliminer la coupure tandis que la contraction (gauche) est nécessaire uniquement pour la règle concernant l'implication gauche. On obtient alors des systèmes comme celui de la figure 7.1, proposé par Melvin C. Fitting[1] [29]. Pour éviter de traiter le problème des cycles potentiels dus à la règle $\rightarrow_{\mathsf{g}}$, Roy Dyckhoff a remplacé cette règle par quatre autres règles équivalentes de la figure 7.2, où il n'y a plus aucune contraction et où la règle $\rightarrow_{\mathsf{g4}}$ est non-inversible[2].

[1] Dragalin a proposé une version avec plusieurs formules à droite pour que les décisions soient guidées par le choix d'une formule et non par les connecteurs logiques.

[2] Une règle est dite non-inversible si la conclusion n'implique pas les prémisses. Utiliser une telle règle dans la recherche d'une preuve peut donc être un mauvais choix puisque la conclusion peut, *a priori*, être impliquées par d'autres prémisses. Ces règles sont exactement les sources potentielles de backtracking dans

L'implantation d'un tel système ne pose donc plus beaucoup de problèmes, si ce n'est qu'il faut bien gérer les possibles backtracks (dues à trois règles non-inversibles $\vee_{\mathsf{d1}}$, $\vee_{\mathsf{d1}}$ et $\rightarrow_{\mathsf{g4}}$). Par ailleurs, pour des raisons d'efficacité, il est important de ne pas traiter toutes les règles de manière uniforme. Une bonne heuristique est de, par exemple, traiter en priorité toutes les règles inversibles (on est sûr de ne pas le regretter) puis de choisir une règle non-inversible à appliquer.

$$\dfrac{}{\Gamma \vdash \top}\,(\top_{\mathsf{d}}) \qquad \dfrac{}{\Gamma, \bot \vdash A}\,(\bot_{\mathsf{g}}) \qquad \dfrac{}{\Gamma, A \vdash A}\,(\mathsf{Ax})$$

$$\dfrac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\,(\wedge_{\mathsf{g}}) \qquad \dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,(\wedge_{\mathsf{d}})$$

$$\dfrac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\,(\vee_{\mathsf{g}}) \qquad \dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,(\vee_{\mathsf{d1}}) \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,(\vee_{\mathsf{d2}})$$

$$\dfrac{\Gamma, A \rightarrow B \vdash A \qquad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}\,(\rightarrow_{\mathsf{g}}) \qquad \dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\,(\rightarrow_{\mathsf{d}})$$

FIG. 7.1 – Système de Fitting

$$\dfrac{\Gamma, A, B \vdash C \qquad A \textit{ atomique}}{\Gamma, A \rightarrow B, A \vdash C}\,(\rightarrow_{\mathsf{g1}})$$

$$\dfrac{\Gamma, A \rightarrow (B \rightarrow C) \vdash D}{\Gamma, (A \wedge B) \rightarrow C \vdash D}\,(\rightarrow_{\mathsf{g2}})$$

$$\dfrac{\Gamma, A \rightarrow C, B \rightarrow C \vdash D}{\Gamma, (A \vee B) \rightarrow C \vdash D}\,(\rightarrow_{\mathsf{g3}})$$

$$\dfrac{\Gamma, B \rightarrow C \vdash A \rightarrow B \qquad \Gamma, C \vdash D}{\Gamma, (A \rightarrow B) \rightarrow C \vdash D}\,(\rightarrow_{\mathsf{g4}})$$

FIG. 7.2 – Règles de l'implication gauche de Dyckhoff

L'utilisation de $\mathcal{L}_{tac}$ pour implanter le système de Dyckhoff semble complètement appropriée dans la mesure où une implantation directe (ce que nous nous proposons de faire) tend à faire appel systématiquement à du filtrage sur les contextes de preuves de manière à distinguer les connecteurs logiques, qu'ils soient en hypothèses ou en conclusion. Par ailleurs, comme nous l'avons dit, la recherche de la preuve peut potentiellement backtracker et la possibilité de backtracking des opérateurs de filtrage (notamment au niveau des hypothèses)

la recherche d'une preuve.

est plutôt incontournable (surtout en l'absence d'évaluation paresseuse). Il est à signaler aussi qu'une telle procédure, appelée `Tauto`, a déjà été implantée, en utilisant également le système LJT, dans le système Coq (version 5.10) par César Muñoz [63] en 1994. Cette tactique a été intégralement réalisée en ML et ce sera justement une bonne occasion pour comparer les deux approches, même si, on le rappelle, les deux métalangages ne sont pas censés s'opposer, ni se substituer l'un à l'autre.

La tactique, que nous appellerons `Tauto'`, sera réalisée essentiellement en trois temps : une tactique s'occupera des règles axiomatiques, une tactique fera une normalisation en utilisant les règles inversibles et, enfin, une dernière tactique gérera le backtracking avec, notamment, les trois règles non-inversibles. Le système de Dyckhoff sera étendu avec les constantes $\top$ et $\bot$, tandis que $\neg A$ sera considéré comme $A \to \bot$ et $A \leftrightarrow B$, comme $A \to B \wedge B \to A$.

7.1.1 Axiomes

Compte-tenu des constantes étendant le système de Dyckhoff, il y a donc trois règles axiomatiques qui peuvent être implantées par la définition toplevel suivante :

```
1  Tactic Definition Axioms :=
2    Match Context With
3    | [|- True] -> Trivial
4    | [_:False |- ?] -> ElimType False;Assumption
5    | [_:?1 |- ?1] -> Auto.
```

Il n'y a rien de particulier à signaler sur cette partie de code plutôt explicite, si ce n'est que l'on peut remarquer l'utilisation avantageuse du filtrage non-linéaire à la ligne 5. Bien évidemment, on pourrait faire sans, en utilisant un test d'égalité (au niveau méta[3]), mais ce n'est pas très pratique et cette possibilité offre un certain confort de programmation, par ailleurs assez intuitif.

7.1.2 Normalisation

La normalisation consiste à utiliser toutes les règles inversibles du système de Dyckhoff. Ces règles seront appliquées une fois pour toutes et ne seront jamais backtrackées. Elles sont traitées par la tactique suivante :

```
Tactic Definition Simplif :=
  Intros;
  Repeat
    ((Match Context With
      | [id:?/\? |- ?] -> Elim id;Do 2 Intro;Clear id
      | [id:?\/? |- ?] -> Elim id;Intro;Clear id
      | [id:?1/\?2->?3 |- ?] ->
        Cut ?1->?2->?3;
          [Intro;Clear id|Intros;Apply id;Try Split;Assumption]
      | [id:?1\/?2->?3 |- ?] ->
        Cut ?2->?3;[Cut ?1->?3;[Intros;Clear id|Intro;Apply id;Left;
          Assumption]|Intro;Apply id;Right;Assumption]
```

[3] En effet, au niveau de Coq, même si les deux objets sont structurellement égaux, rien ne permet de le montrer automatiquement, sauf si un lemme de décidabilité de l'égalité sur les types des objets en question a été montré, mais ce n'est pas toujours le cas et il faudrait le rechercher dans l'environnement.

```
      | [id0:?1->?2; id1:?1 |- ?] ->
        Generalize (id0 id1);Intro;Clear id0
      | [|- ?/\?] -> Split);Intros).
```

On remarque que tous les cas du filtrage correspondent exactement aux règles inversibles du système de Dyckhoff. Pour normaliser, le tactical `Repeat` est tout à fait approprié. On aurait pu aussi rendre `Simplif` récursive et la rappeler dans chaque branche du filtrage, mais cela aurait été certainement un peu plus lourd.

7.1.3 Tactique principale

Enfin, il nous reste à traiter le cas de la tactique principale qui appellera les deux définitions précédentes et surtout gérera le backtracking dû aux trois règles non-inversibles. Cette tactique est décrite de la manière suivante :

```
1  Recursive Tactic Definition TautoMain :=
2    Simplif;Axioms
3    Orelse
4      Match Context With
5      | [id:(?1->?2)->?3|- ?] ->
6        Cut ?2->?3;[Intro;Cut ?1->?2;[Intro;Cut ?3;[Intro;Clear id|
7          Intros;Apply id;Assumption]|Clear id]|Intros;Apply id;Intro;
8          Assumption];TautoMain
9      | [|- ?\/?] -> (Left;TautoMain) Orelse (Right;TautoMain).
```

Cette tactique est récursive (utilisation du mot-clé `Recursive` devant `Tactic Definition`), car les appels récursifs sont un peu plus subtils que dans la tactique précédente, où l'on se rappelait systématiquement à la fin des branches du filtrage (avec le `Repeat`). Par exemple, en ligne 9, on a deux appels récursifs distincts qui ne peuvent pas être remplacés par un seul en fin de ligne.

En regardant ce code, on reconnaît bien l'algorithme que l'on s'était fixé : d'abord, on simplifie, puis on essaie d'appliquer les axiomes (ce choix est géré par le `Orelse` de la ligne 3), si cela marche alors on a fini, sinon on essaie une des règles non-inversibles et on recommence.

Le backtracking des règles non-inversibles est globalement pris en main par la construction `Match Context With` pour le choix entre la règle $\rightarrow_{\mathtt{g4}}$ et les règles $\vee_{\mathtt{d}*}$ (le backtracking entre ces deux règles est alors assuré par un `Orelse`[4]). En effet, dans le premier motif, par exemple, la recherche des hypothèses se faisant toujours dans le même sens (de la plus ancienne à la plus récente), si une hypothèse filtrée s'avère un mauvais choix alors la sémantique de `Match Context With` permet alors d'essayer la prochaine hypothèse qui filtrera. Sans ce système intrinsèque à `Match Context With`, un appel récursif en cas d'échec ferait systématiquement retomber sur la même hypothèse et on ne pourrait pas capturer les autres hypothèses de la même forme qui peuvent potentiellement apporter la preuve. On peut aussi remarquer que le backtraking donne clairement la priorité à la règle concernant l'implication gauche mais ce n'est pas problématique puisque le backtracking implanté dans `Match Context With` est complet et, en cas d'échec total sur toutes les hypothèses ayant la forme du motif de la ligne 5, la règle de la ligne 9 serait alors utilisée avec le même système de backtracking.

[4] On aurait pu tout traiter avec `Match Context With` en doublant le motif de la ligne 9 et en partant soit à gauche, soit à droite, mais la solution choisie est un peu plus compacte et certainement plus élégante car elle groupe les règles $\vee_{\mathtt{d}*}$ selon leur connecteur logique.

Pour traiter le cas des constantes $\neg$ et $\leftrightarrow$, il suffit de les δ-réduire et cela peut être fait par la définition suivante :

```
Tactic Definition Tauto' := Unfold not iff;TautoMain.
```

7.1.4 Exemples de tautologies

Nous allons maintenant donner quelques exemples de tautologies propositionnelles intuitionnistes et montrer le résultat de l'application de la tactique `Tauto'` que nous venons de décrire. Tous les exemples sont réalisés dans la version 7.0 de Coq (en natif), où, on le rappelle, $\mathcal{L}_{tac}$ a été intégré. La machine utilisée est un Intel Pentium III à 600 MHz sous Linux Mandrake 7.2. On suppose que toutes les définitions précédentes ont déjà été évaluées par le toplevel de Coq.

```
Coq < Parameter A,B,C,D,E,F:Prop.

Coq < Goal (A->(B/\C)) -> ((A->B)\/(A->C)).
1 subgoal

  ============================
   (A->B/\C)->(A->B)\/(A->C)

Unnamed_thm < Time Tauto'.
Subtree proved!
Finished transaction in 0 secs (0.01u,0s)

Coq < Goal ~(~(A \/ ~A)).
1 subgoal

  ============================
   ~~(A\/~A)

Unnamed_thm < Time Tauto'.
Subtree proved!
Finished transaction in 0 secs (0.01u,0s)

Coq < Goal (~~B->B)->(~B->~A)->~~A->B.
1 subgoal

  ============================
   (~~B->B)->(~B->~A)->~~A->B

Unnamed_thm < Time Tauto'.
Subtree proved!
Finished transaction in 0 secs (0u,0s)

Coq < Goal (~A<->B)->(~(C\/E)<->D/\F)->~(C\/A\/E)<->D/\B/\F.
1 subgoal

  ============================
```

```
  (~A<->B)->(~(C\/E)<->D/\F)->~(C\/A\/E)<->D/\B/\F

Unnamed_thm < Time Tauto'.
Subtree proved!
Finished transaction in 0 secs (0.12u,0s)
```

7.1.5 Remarques

Une première remarque consiste à noter que le code de `Tauto'` est très court (30 lignes exactement), essentiellement grâce à l'opérateur de filtrage sur les buts et à sa sémantique particulière qui permet le backtracking. Par rapport à la tactique d'origine écrite en ML, le gain de taille paraît même vertigineux puisque le code avoisinait les 2000 lignes.

Par ailleurs, le code est complètement lisible. En effet, on reconnaît, grâce aux motifs utilisés, l'implantation des règles du système de Dyckhoff. Cette lisibilité permet un débugage rapide et la maintenance d'une telle tactique est alors grandement facilitée. C'est un progrès important par rapport à son équivalente en ML, où il fallait, en général, être bien renseigné sur l'architecture globale pour traquer les incomplétudes de l'implantation.

Par rapport à la première implantation, un autre bonus a été observé concernant les performances de la tactique. De manière général, sur tous les exemples testés, `Tauto'` est bien plus rapide allant jusqu'à 95% de gain (ce maximum est, entre autre, atteint pour le troisième exemple de la section 7.1.4). Cet accroissement impressionnant des performances n'est pas dû à des caractéristiques particulières de $\mathcal{L}_{tac}$, mais plus à quelques sources d'inefficacité dans le code d'origine. Toutefois, le gain de lisibilité apporté par $\mathcal{L}_{tac}$ a permis d'avoir une vision plus globale de l'algorithme, et, de ce fait, de tendre vers une plus grande rationalisation du code.

Un dernier point concerne la portabilité de `Tauto'`, qui, étant donné qu'elle utilise un langage appartenant à la spécification de Coq, pourra être conservée telle quelle au fil des différentes versions de Coq. En effet, $\mathcal{L}_{tac}$ appartient, d'une certaine manière, au langage de Coq et est, à ce titre, documenté comme le langage de spécification. L'utilisateur est donc en droit d'attendre de ce langage une évolution plutôt conservative quelle que soient les modifications internes (du code) effectuées. Cette portabilité est un gain significatif par rapport à la même tactique écrite en ML qui pouvait être amenée à évoluer suivant la profondeur des changements entrepris dans tout le code, puisque le métalangage utilisé était également le langage d'implantation. Pour illustrer cette observation, on peut citer le cas du passage de la version 6 à la version 7 de Coq, où une importante restructuration du code a nécessité des modifications conséquentes, voire des réécritures totales de certaines tactiques, occasionnant, à terme, une perte de temps significative.

Pour toutes ces raisons, la tactique d'origine a été remplacée par le code que nous avons donné précédemment. Quelques modifications ont été apportées pour gérer la δ-réduction, le comportement vis-à-vis des types dépendants et pour fonctionner modulo isomorphismes des connecteurs logiques[5]. Nous verrons plus loin comment ces ajouts ont pu être effectués.

7.2 Isomorphismes de types

Nous nous proposons de montrer automatiquement que deux types du λ-calcul simplement typé avec produit cartésien et *top* sont isomorphes. La notion d'isomorphisme de type

[5] C'était un comportement intéressant de la tactique initiale, que nous avons voulu conserver.

généralement utilisée est essentiellement syntaxique[6] (voir [26]) et nous dirons que deux types A et B sont isomorphes dans un calcul $\mathcal{C}$ si et seulement si il existe deux fonctions f et g, de types respectifs $A \to B$ et $B \to A$, dans $\mathcal{C}$, telles que $f \circ g =_{\mathcal{L}} \mathcal{I}_B$ et $g \circ f =_{\mathcal{L}} \mathcal{I}_A$, où $\mathcal{I}_A$ et $\mathcal{I}_B$ représentent les fonctions identités respectivement sur A et B. Les termes f et g sont appelés termes inversibles ou termes de conversion. L'égalité $=_{\mathcal{L}}$ est, en général, plus large que la simple égalité syntaxique.

En 1981, Sergei Soloviev [80] a montré que, pour les catégories cartésiennes fermées (CCC's en anglais), les seuls isomorphismes étaient exactement ceux de la figure 7.3, où $\times$ représente le produit cartésien et $\top$, le type top.

$$\boxed{\begin{gathered}
A \times B = B \times A \\
A \times (B \times C) = (A \times B) \times C \\
(A \times B) \to C = A \to (B \to C) \\
A \to (B \times C) = (A \to B) \times (A \to C) \\
A \times \top = A \\
A \to \top = \top \\
\top \to A = A
\end{gathered}}$$

FIG. 7.3 – Isomorphismes dans les CCC's

Comme les CCC's sont les modèles du λ-calcul simplement typé avec produit cartésien et top, on en déduit donc que les isomorphismes de la figure 7.3 sont également les seuls isomorphismes de types du λ-calcul simplement typé avec produit cartésien et top. Ce sont ces isomorphismes que nous voulons décider.

Pour ce faire, notons, tout d'abord, deux remarques importantes concernant cette théorie. La première est qu'elle passe au contexte et, de ce fait, pour prouver d'autres isomorphismes de types, nous pouvons utiliser les règles habituelles de la logique équationnelle. La deuxième est qu'elle est décidable et l'algorithme consiste essentiellement à réécrire les deux types selon les règles orientables de la figure 7.3, puis à comparer les deux formes normales selon les autres règles.

Plus précisément, le système de réécriture est donné par la figure 7.4. Ces règles sont essentiellement issues des équations de la figure 7.3, où le premier axiome a été supprimé et où, de ce fait, le cinquième axiome a été symétrisé (par rapport au produit cartésien). On peut montrer que ce système est confluent et fortement normalisant. Les formes normales ont la forme suivante :

$$A_{11} \to (A_{12} \to \ \ldots (A_{1n_1-1} \to A_{1n_1})) \times (\ldots \times A_{m1} \to (A_{m2} \to \ \ldots (A_{mn_m-1} \to A_{mn_m})))$$

où les A_{ij} ne contiennent pas de $\times$ et de $\top$. Il reste alors à comparer les formes normales selon le premier axiome[7] de la figure 7.3, c'est-à-dire modulo permutation des composantes du produit cartésien. Toutefois, ce n'est pas suffisant (pour être complet), car la deuxième

[6] Par opposition à une vision plus sémantique, où les isomorphismes de types sont définis par rapport aux modèles du langage.

[7] Cet axiome est complètement symétrique et ne pouvait clairement pas être orienté pour obtenir le système normalisant de la figure 7.4.

règle de la figure 7.4 (curryfication) élimine les produits cartésiens à gauche des flèches (produits non dépendants). Pour raisonner modulo commutativité du produit cartésien, il faut donc aussi raisonner modulo permutation des types des arguments de fonctions, c'est-à-dire l'axiome suivant[8] :

$$A \to (B \to C) = B \to (A \to C)$$

$$
\begin{array}{c}
(A \times B) \times C \longrightarrow A \times (B \times C) \\
(A \times B) \to C \longrightarrow A \to (B \to C) \\
A \to (B \times C) \longrightarrow (A \to B) \times (A \to C) \\
A \times \top \longrightarrow A \\
\top \times A \longrightarrow A \\
A \to \top \longrightarrow \top \\
\top \to A \longrightarrow A
\end{array}
$$

FIG. 7.4 – Normalisation des types

On peut montrer que cette procédure de décision est correcte et complète pour la théorie de la figure 7.3, ainsi que terminante. On pourra se reporter à [26] pour plus de détails.

Nous allons maintenant décrire le codage de cette procédure de décision directement à toplevel et voir comment $\mathcal{L}_{tac}$ permet de résoudre des parties complexes de l'algorithme.

7.2.1 Axiomatisation

Pour spécifier les axiomes de la figure 7.3, nous pouvons utiliser l'égalité du système Coq, à savoir l'égalité de Leibniz (égalité syntaxique). Cela permet de bénéficier de toutes les tactiques de réécriture prédéfinies du système, sans avoir à montrer de lemmes de compatibilité. Ces axiomes étendent alors l'égalité de Leibniz avec des relations entre objets qui ne sont pas syntaxiquement égaux. De ce fait, avec cette nouvelle axiomatisation, l'égalité ne peut plus être dite de Leibniz.

Une autre possibilité, qui peut éviter une axiomatisation, est de définir réellement la relation d'isomorphisme de type, que nous avons vue précédemment. Cela peut être fait par la définition suivante :

```
Definition iso [A,B:Set] :=
  {c:(A->B)*(B->A)|
   let (f,g) = c in
   (([x:B](f (g x)))=[x:B]x)/\(([x:A](g (f x)))=[x:A]x)}.
```

Toutefois, dans le cas des axiomes de la figure 7.3, l'égalité qui permet de montrer que les termes inversibles commutent, n'est pas que l'égalité syntaxique. Cette égalité est plus extensionnelle car elle contient la β-conversion, la η-conversion, *surjective-pairing*, ... L'idée est donc soit d'étendre l'égalité de Leibniz avec ces nouvelles règles de manière à pouvoir montrer les commutations, soit de définir cette nouvelle égalité récursivement en montrant

[8] Cet axiome n'est pas un axiome supplémentaire et il se déduit de la théorie de la figure 7.3.

quelques lemmes de congruence au passage. La deuxième solution est certainement préférable car elle supprime tous les axiomes et, surtout, permet de ne pas étendre l'égalité de Leibniz, qui reste une véritable égalité syntaxique.

Pour des raisons de rapidité de formalisation, nous avons opté pour la première possibilité, c'est-à-dire une axiomatisation totale. Les axiomes de la figure 7.3 sont alors définis directement de la manière suivante :

```
1   Section Iso_axioms.
2
3   Variable A,B,C:Set.
4
5   Axiom Com:(A*B)==(B*A).
6   Axiom Ass:(A*(B*C))==((A*B)*C).
7   Axiom Cur:((A*B)->C)==(A->B->C).
8   Axiom Dis:(A->(B*C))==((A->B)*(A->C)).
9   Axiom P_unit:(A*unit)==A.
10  Axiom AR_unit:(A->unit)==unit.
11  Axiom AL_unit:(unit->A)==A.
12
13  End Iso_axioms.
```

où `*` représente le produit cartésien (dans `Set`), `unit`, le type $\top$ et `==`, l'égalité dans `Type`. On est forcé de prendre cette égalité car l'égalité de Leibniz usuelle `=` relie deux objets dont le type est de type `Set`. Ici, nous devons relier deux objets dont le type est `Set` et donc de type `Type`, d'où l'utilisation de `==`[9].

La section `Iso_axioms` permet d'utiliser la quantification universelle implicite. En effet, tous les axiomes sont implicitement universellement quantifiées par rapport aux variables `A`, `B` et `C`, qui sont déclarées à la ligne 3. En dehors de la section, ces variables ne sont pas visibles et toutes les quantifications sont automatiquement ajoutées aux différents axiomes suivant leurs dépendances vis-à-vis de ces variables[10]. Par exemple, après la ligne 13, on aura les types suivants :

```
Coq < Check Com.
Com
     : (A,B:Set)(A*B)==(B*A)

Coq < Check Ass.
Ass
     : (A,B,C:Set)(A*B*C)==((A*B)*C)
```

7.2.2 Quelques lemmes utiles

Avant de coder la tactique proprement dite, voici quelques lemmes (ainsi que leurs preuves), que nous utiliserons exclusivement lors de la phase de comparaison (après la normalisation) :

```
15  Lemma CartCons:(A,B,C:Set)B==C->(A*B)==(A*C).
16  Proof.
```

[9]Ces deux égalités, `=` et `==`, sont à valeur dans `Prop`. Il existe une troisième égalité `===`, similaire à `==`, mais directement à valeur dans `Type`.

[10]Ce principe est appelé *discharge* et est déclenché par la commande `End nom_section`.

```
17      Intros A B C Heq;Rewrite Heq;Reflexivity.
18    Save.
19
20    Lemma ProdCons:(A,B,C:Set)B==C->(A->B)==(A->C).
21    Proof.
22      Intros A B C Heq;Rewrite Heq;Reflexivity.
23    Save.
24
25    Lemma ProdPermut:(A,B,C:Set)(A->B->C)==(B->A->C).
26    Proof.
27      Intros;Rewrite <- Cur;Rewrite Com;Rewrite Cur;Reflexivity.
28    Save.
```

7.2.3 Normalisation

La partie normalisation ne pose pas de problème particulier. Il s'agit d'implanter les règles du système de la figure 7.4 en ajoutant la règle (implicite) de congruence. Ceci peut être réalisé par les définitions mutuellement récursives suivantes :

```
30    Recursive Tactic Definition OneStepSimplif trm :=
31      (Match trm With
32       | [[?1*unit]] -> Rewrite (P_unit ?1)
33       | [[unit*?1]] -> Rewrite (Com unit ?1)
34       | [[?1->unit]] -> Rewrite (AR_unit ?1)
35       | [[unit->?1]] -> Rewrite (AL_unit ?1)
36       | [[(?1*?2)->?3]] -> Rewrite (Cur ?1 ?2 ?3)
37       | [[?1->(?2*?3)]] -> Rewrite (Dis ?1 ?2 ?3)
38       | [[(?1*?2)*?3]] -> Rewrite <- (Ass ?1 ?2 ?3));Try Simplif
39    And Simplif :=
40      Match Context With
41      | [|- ?1==?2] -> Try (OneStepSimplif ?1);Try (OneStepSimplif ?2).
```

La fonction principale est `Simplif` qui identifie les membres de l'égalité à prouver (par la construction `Match Context` de la ligne 40) et les simplifie. Elle fait appel à la fonction `OneStepSimplif` qui réalise exactement un pas de réécriture si c'est possible. La congruence n'est pas implantée totalement de manière explicite[11], mais au moyen du filtrage sous-terme (disponible à la fois pour `Match` et `Match Context`) dénoté par l'introduction de `[]` autour du motif. Cette possibilité offerte par les opérateurs de filtrage se révèle ici très pratique car l'implantation est complètement fidèle aux règles de la figure 7.4 et il n'y a pas de règles supplémentaires nécessaires pour passer au contexte.

Cette implantation n'est pas très efficace car l'idée consiste essentiellement à réécrire sur des sous-termes jusqu'à ce que l'on ne puisse plus. Toutefois, cette version naïve a l'avantage d'être une implantation directe et, de ce fait, rapidement formalisable. Par ailleurs, quelques précautions ont été prises pour améliorer les performances, notamment en ce qui concerne l'ordre de priorité des règles (qui n'est pas du tout celui de la figure 7.4). En effet, la règle de distribution de la flèche sur le produit cartésien créant une duplication (troisième règle de la figure 7.4), les règles de simplification de `unit` ainsi que la curryfication (respectivement

[11] En effet, dans la fonction `OneStepSimplif`, il n'y a pas de règle qui rappelle récursivement `Simplif` sur les parties gauche et droite du produit cartésien ou de la flèche.

la troisième et les quatre dernières règles de la figure 7.4) sont effectuées en priorité. L'associativité du produit cartésien (première règle de la figure 7.4) est faite en dernier lorsque la forme normale a été obtenue (le faire avant pourrait se révéler complètement inutile du fait de simplifications ultérieures).

7.2.4 Comparaison des formes normales

Comme nous l'avons vu précédemment, la comparaison des formes normales se fait modulo permutation des composantes du produit cartésien et, pour chaque composante, modulo permutation des types des arguments de fonctions. Comme nous en aurons besoin pour comparer deux composantes de produit cartésien, nous allons donc commencer par présenter la partie comparaison des types fonctionnels.

Types fonctionnels

Avant de réaliser la comparaison en question, nous avons besoin de quelques définitions auxiliaires. Tout d'abord, voici une fonction qui calcule le nombre de types dans un type fonctionnel avec associativité à droite (c'est-à-dire qu'il peut y avoir des composantes fonctionnelles non atomiques qui n'entreront pas dans le décompte) :

```
Recursive Meta Definition ProdLength trm:=
  Match trm With
  | [?->?1] -> Let succ=(ProdLength ?1) In '(S succ)
  | _ -> '(1).
```

où `S` représente le constructeur successeur des entiers naturels `nat` et où `(1)` est du sucre syntaxique pour `(S O)`[12].

Cette fonction est une `Meta Definition` car elle ne raffine pas un but (comme les `Tactic Definition`) mais rend un résultat d'un autre type (ici un terme de Coq). Pour le compteur, on n'utilise pas les entiers prédéfinis de $\mathcal{L}_{tac}$ (car on ne peut réaliser aucune opération sur eux et, ici, il nous faut pouvoir incrémenter), mais ceux de Coq[13] (qui peuvent être facilement manipulés par les opérateurs de filtrage).

Nous aurons besoin aussi d'une fonction qui vérifie (avant de permuter les types des arguments) que les deux types fonctionnels ont la même conclusion (dernier type atomique le plus à droite) :

```
48  Recursive Tactic Definition SameConclusion trm :=
49    Match trm With
50    | [?1==?1] -> Idtac
51    | [(?->?1)==(?->?2)] -> SameConclusion '(?1==?2)
52    | _ -> Fail.
```

Dans cette fonction, on suppose que les deux types sont donnés sous la forme d'une égalité (visible dans les motifs des lignes 50 et 51). On ne rend pas de résultat booléen (bien que ce serait complètement possible en utilisant le type prédéfini `bool` de Coq) mais on ne fait rien (tactique `Idtac`) si les deux types ont la même conclusion ou on échoue (tactique

[12] Cette extension syntaxique est disponible en important le module `Arith`.

[13] Comme nous l'avons vu précédemment, ce choix se justifie pleinement dans la mesure où $\mathcal{L}_{tac}$ n'est pas censé se substituer à Objective Caml et l'apparition de structures de données iraient probablement dans ce sens. Par ailleurs, si les structures de données de Coq sont accessibles, autant les utiliser, même s'il faut s'attendre à un surcoût par rapport à celles d'Objective Caml.

`Fail`) sinon. Ce système de programmation se révèle fort pratique car, par exemple ici, cette tactique pourra être appelée en début de séquence et laissera les autres tactiques de la séquence s'évaluer normalement en cas de succès ou fera échouer toute la séquence dans le cas contraire[14].

Il nous faut aussi une fonction qui réalise la permutation du type de tête et du type juste avant la conclusion du type fonctionnel. Plus précisément, il s'agit de montrer l'égalité (au sens isomorphe) suivante :

$$A_1 \to (A_2 \to \ldots (A_{n-1} \to A_n)) = A_{n-1} \to (A_2 \to \ldots (A_1 \to A_n))$$

où A_n est un type atomique[15].

Cette permutation est réalisée au moyen des deux définitions suivantes :

```
Recursive Meta Definition GiveArg ctr trm :=
  Match trm With
  | [?1->?2] ->
    (Match ctr With
     | [(2)] -> '(?1->?2)
     | _ -> GiveArg '(pred ctr) ?2)
  | [?1] -> ?1.

Recursive Tactic Definition ApplyProdPermut n ctr :=
  Match Eval Compute in (n=ctr) With
  | [?1=?1] -> Idtac
  | _ ->
    Let trm =
      (Match Context With
       | [|- ?1==?] -> (GiveArg ctr ?1)) In
    (Match trm With
     | [?1->?2->?3] ->
       Pattern 1 trm;Rewrite (ProdPermut ?1 ?2 ?3);
         ApplyProdPermut n '(S ctr)
     | _ -> Fail).
```

La difficulté ici réside dans le fait que la permutation que nous voulons réaliser n'est pas élémentaire et ne correspond pas à une simple application du lemme `ProdPermut`, que nous avons montré précédemment. Nous devons donc permuter d'abord les deux premières composantes du type fonctionnel, puis la deuxième et la troisième, et ainsi de suite jusqu'à atteindre l'avant-dernière composante (juste avant la conclusion).

Pour ce faire, l'idée consiste à conserver un indice identifiant la composante à permuter. Cet indice est la variable `ctr`. Elle commence à 2 et s'arrête lorsque l'on a atteint le nombre total de composantes du type fonctionnels, qui correspond à la variable `n`. En effet, si on a n composantes, alors on doit faire exactement $n-2$ permutations. Cela suppose que ces fonctions soient appelées uniquement si on a une flèche au minimum, le cas dégénéré (type atomique) étant traité à part.

La fonction `GiveArg` prend l'indice de la composante (`ctr`) à permuter ainsi que le type fonctionnel (`trm`) et rend le sous-terme du type fonctionnel pointé par cet indice. La

[14] C'est exactement le comportement recherché dans ce cas précis, où ce n'est pas la peine d'aller permuter les types des arguments si les deux types n'ont pas la même conclusion (ils sont trivialement non isomorphes).

[15] En fait, dans ce cas, le type est atomique et de surcroît différent de `unit`, puisque la comparaison a lieu après la normalisation.

fonction `ApplyProdPermut`, qui prend le nombre de composantes du type fonctionnel (`n`) et l'indice (`ctr`), récupère le type à permuter (par un `Match Context` en ligne 67 et le motif de la ligne 68), le transmet à `GiveArg`, stocke le terme rendu par `GiveArg`, l'isole (commande `Pattern` de la ligne 71) pour réaliser la permutation (réécriture par le `Rewrite` de la ligne 71) et se rappelle récursivement en incrémentant le compteur (construction de l'expression `(S ctr)` à la ligne 72) jusqu'à ce que le compteur soit égal au nombre total de composantes du type fonctionnel (premier cas de filtrage du `Match` à la ligne 64).

Nous pouvons maintenant écrire la tactique qui montrera que deux types sont égaux modulo permutation des types des arguments. Voici deux définitions mutuellement récursives qui réalisent cette preuve si c'est possible :

```
75  Recursive Tactic Definition DoProdCompare n :=
76    Match Context With
77    | [|- ?1==?1 ] -> Reflexivity
78    | [|- (?1->?2)==(?3->?4) ] ->
79      (Cut ?1==?3;
80         [Intro H;Pattern 1 ?1;Rewrite H;Apply ProdCons;
81          Let newl = (ProdLength ?2) In DoProdCompare newl
82         |ProdCompare])
83      Orelse
84      (Match n With
85       | [ (2) ] -> Fail
86       | _ ->
87         Let np = (ProdLength '(?1->?2)) In
88         ApplyProdPermut np '(2);DoProdCompare '(pred n))
89  And ProdCompare :=
90    Match Context With
91    | [|- ?1==?2] ->
92      SameConclusion '(?1==?2);
93      Let l1 = (ProdLength ?1)
94      And l2 = (ProdLength ?2) In
95      (Match Eval Compute in l1=l2 With
96       | [?1=?1] -> (DoProdCompare ?1)).
```

La fonction principale est `ProdCompare` (ligne 89). Dans cette fonction, on vérifie d'abord que les deux types ont la même conclusion (appel de `SameConclusion` en ligne 92). Ensuite, on calcule le nombre de composantes du type fonctionnel (appels de `ProdLength` en lignes 93 et 94) et si les deux types ont bien le même nombre de composantes (motif non linéaire de la ligne 96) alors on appelle la seconde définition `DoProdCompare` (avec ce nombre de composantes calculé), censée réaliser les permutations pour trouver une identification entre les deux types.

La fonction `DoProdCompare` vérifie d'abord si les deux types sont égaux (motif non linéaire de la ligne 77), au cas où la comparaison est trivialement terminée. Ensuite, si les deux types sont des flèches, alors il faut vérifier que les deux têtes sont égales. Mais ces deux composantes n'étant pas forcément atomiques[16], il faut rappeler `ProdCompare` récursivement. Ceci est fait dans la première composante du `Orelse` de la ligne 83 par une coupure (ligne 79) posant l'égalité des composantes de tête. Les lignes 80 et 81 supposent que l'on a cette égalité et on se rappelle donc récursivement sur les queues des deux types (`DoProdCompare` de la ligne 81). La ligne 82 s'occupe de montrer cette égalité. En cas d'échec,

[16] On peut effectivement avoir d'autres flèches à l'intérieur (fonctions d'ordre supérieur).

on se reporte sur la seconde composante du `Orelse` pour effectuer la permutation que nous avons décrite précédemment. Si le compteur `n` arrive à 2 (ligne 85) alors on a réalisé toutes les permutations possibles[17] et on échoue (avec `Fail`). Si on a encore le droit de faire des permutations (cas `_` en ligne 86), on calcule le nombre de composantes du type fonctionnel (ce nombre n'est pas `n` qui, certes au départ est égal au nombre de composantes, mais qui est relié au nombre de permutations effectuées) pour appeler `ApplyProdPermut` (ligne 88), qui fait la permutation proprement dite, puis on rappelle récursivement `DoProdCompare` en décrémentant `n` (ligne 88), ce qui revient à incrémenter le nombre de permutations réalisées.

Remarque 7.2.1 A priori*, on suppose que la tactique pourra être appelée dans n'importe quelle condition, c'est-à-dire qu'elle pourra être appliquée à tout but. Ce ne sera pas tout à fait le cas pour* `ProdCompare` *et, plus particulièrement, pour* `DoProdCompare`*. En effet, dans* `DoProdCompare`*, après la coupure de la ligne 79, on introduit l'expression de la coupure, en ligne 80, par un* `Intro H` *afin de nommer l'hypothèse en vue d'une réutilisation ultérieure par* `Rewrite`*, toujours en ligne 80. Si l'hypothèse n'est pas nommée, la réutilisation par* `Rewrite` *est impossible, car ne connaissant pas le nom automatiquement donné par le système. Toutefois, ce nommage statique peut échouer s'il y a déjà une hypothèse nommée* `H` *dans le contexte local (le système échoue en effet dans ce cas et ne renomme pas l'hypothèse que l'on cherche à introduire[18]).*

Pour éviter ce potentiel échec, une solution consiste à introduire l'hypothèse sans lui donner de nom (avec un simple `Intro`*) et à récupérer, par la suite, le nom (identificateur) donné par le système pour cette hypothèse avec la définition suivante :*

```
1   Meta Definition GrepHypId trm :=
2     Match Context With
3     | [ id:trm |- ? ] -> id.
```

Puisque les métavariables de filtrage ne sont pas de simples identificateurs (mais des expressions toujours préfixées par `?`*), cela nous permet d'utiliser les constantes, ainsi que des paramètres formels, dans les motifs. Ici, il suffit d'indiquer le type (variable* `trm` *de la ligne 1) de l'hypothèse dont on veut le nom et le motif de la ligne 3 sera complètement dynamique, c'est-à-dire, interprété en fonction de la valeur de* `trm` *au moment de l'évaluation du* `Match Context`*.*

L'utilisation dans `DoProdCompare` *se ferait donc de la manière suivante :*

```
79       (Cut ?1==?3;
80          [Intro;Pattern 1 ?1;Let id = (GrepHypId '(?1==?3)) In Rewrite id;
81           Apply ProdCons;Let newl = (ProdLength ?2) In DoProdCompare newl
82          |ProdCompare])
```

Cette version est complètement robuste vis-à-vis du nommage des hypothèses. Nous n'avons pas utilisé cette modification afin de ne pas alourdir le codage de la tactique, sachant que, dans les exemples que nous verrons par la suite, le problème ne se pose pas.

Maintenant que nous avons la tactique qui vérifie que deux types fonctionnels sont égaux modulo permutation des types de leurs arguments, nous pouvons décrire la comparaison des produits cartésiens que nous obtenons après normalisation.

---

[17] On a vu que, pour un type fonctionnel avec n composantes, on pouvait réaliser $n-2$ permutations des types des arguments.

[18] Cette politique peut sembler peu flexible mais elle évite des situations peu intuitives, où l'on introduit, par exemple, une hypothèse `H`, et où l'on réutilise, par la suite, cette même hypothèse avec `H0` !

Produits cartésiens

Avant d'entreprendre la fonction de comparaison en question, nous avons besoin de quelques définitions auxiliaires. D'abord, tout comme pour les types fonctionnels, nous avons besoin d'une fonction donnant le nombre de composantes d'un produit cartésien :

```
98   Recursive Meta Definition CartLength trm:=
99     Match trm With
100    | [?*?1] -> Let succ=(CartLength ?1) In '(S succ)
101    | _ -> '(1).
```

`CartLength` est complètement similaire à `ProdLength` et l'unique différence se situe au niveau du symbole infixe utilisé (ici `*` dans le motif de la ligne 100).

Nous aurons aussi besoin de la tactique suivante qui permet d'associer à droite tout produit cartésien :

```
103  Tactic Definition CartAssoc := Repeat Rewrite <- Ass.
```

L'algorithme de `CartAssoc` est assez trivial et consiste à associer à droite les produits cartésiens dans tous les sous-termes (`Rewrite` permet d'associer sur une certaine occurence et `Repeat` permet de recommencer sur d'éventuels autres).

Nous avons maintenant tous les éléments pour comparer les produits cartésiens et ceci est fait par les définitions mutuellement récursives suivantes, basées sur un schéma très similaire aux fonctions `DoProdCompare` et `ProdCompare` :

```
105  Recursive Tactic Definition DoCartCompare n:=
106    Match Context With
107    | [|- (?1*?2)==(?3*?4)] ->
108      (Cut ?1==?3;
109         [Intro H;Pattern 1 ?1;Rewrite H;Apply CartCons;
110          Let newl = (CartLength ?2) In DoCartCompare newl
111         |ProdCompare])
112      Orelse
113      (Match Eval Compute in n With
114       | [(1)] -> Fail
115       | _ ->
116         Pattern 1 (?1*?2);Rewrite Com;CartAssoc;DoCartCompare '(pred n))
117    | [|- ?1==?2] -> ProdCompare
118  And CartCompare :=
119    Match Context With
120    | [|- ?1==?2] ->
121      Let l1=(CartLength ?1)
122      And l2=(CartLength ?2) In
123      (Match Eval Compute in l1=l2 With
124       | [?1=?1] -> DoCartCompare ?1).
```

La fonction principale à appeler est `CartCompare`. Elle vérifie d'abord que les deux produits cartésiens (récupérés par le `Match Context` de la ligne 119) ont le même nombre de composantes (appels à `CartLength` en lignes 121 et 122). Si c'est le cas, `DoCartCompare` est appelé avec ce nombre de composantes.

Dans la fonction `DoCartCompare`, on traite d'abord le cas où les deux types sont un produit (motif de la ligne 107). On vérifie alors que les deux composantes de tête (qui sont des types fonctionnels[19]) sont isomorphes modulo permutation des types de leurs arguments. Ceci est fait par une coupure de leur égalité (ligne 108). En lignes 109 et 110, on suppose cette égalité et on rappelle récursivement `DoCartCompare` sur les queues des produits cartésiens. En ligne 111, on essaie de prouver cette égalité avec `ProdCompare`, précédemment décrite. En cas d'échec, le `Orelse` de la ligne 112 rattrape l'échec et redirige vers la partie du code réalisant la permutation. On regarde si on peut encore faire des permutations, c'est-à-dire si le compteur `n` est supérieur strictement à 1. En effet, sur un produit cartésien de longueur n, on peut faire au plus $n-1$ permutations. Ainsi, si `n` arrive à 1 (ligne 114), alors on échoue (avec `Fail`). Dans le cas contraire (lignes 115 et 116), on extrait le produit cartésien à permuter avec `Pattern` pour ne permuter que celui de droite (qui est la référence) avec `Rewrite`, puis on rappelle `DoProdCompare` en décrémentant `n` pour indiquer que l'on vient de réaliser une permutation. Le dernier cas (arrêt) consiste à comparer deux types sans produits cartésiens (ligne 117), ce qui est directement traité en appelant `ProdCompare`, servant à comparer deux composantes.

Cela clôt la partie comparaison des formes normales, qui est effectuée en appelant la tactique `CartCompare`.

7.2.5 Tactique principale

La tactique principale est plutôt succincte car il s'agit de normaliser les deux types, puis de comparer les deux formes normales ainsi obtenues. On aura donc la définition suivante :

```
126   Tactic Definition IsoProve := Simplif;CartCompare.
```

où les tactiques `Simplif` et `CartCompare` ont été précédemment décrites.

7.2.6 Quelques exemples

Nous allons maintenant donner quelques isomorphismes de types dans les CCC's pour tester notre tactique `IsoProve`. Nous utiliserons la version 7.0 de Coq, compilée en natif. La machine, sur laquelle ces tests ont été réalisés, est un Intel Pentium III à 600 MHz sous Linux Mandrake 7.2. Toutes les évaluations ont été faites en mode interactif (sous le toplevel de Coq) et, dans ce qui suit, on suppose que les définitions précédentes (concernant `IsoProve`) ont déjà été interprétées.

```
Coq < Parameter A,B,C,D:Set.

Coq < Goal ((A*unit)*B)==(B*(unit*A)).
1 subgoal

  ============================
   ((A*unit)*B)==(B*unit*A)

Unnamed_thm < Time IsoProve.
Subtree proved!
Finished transaction in 0 secs (0.03u,0s)

Coq < Goal ((A*unit)->(unit->B)->C)==((B*unit)->(unit->A)->C).
1 subgoal
```

[19] Le cas atomique est un cas dégénéré.

```
  ============================
   (A*unit->(unit->B)->C)==(B*unit->(unit->A)->C)

Unnamed_thm < Time IsoProve.
Subtree proved!
Finished transaction in 0 secs (0.03u,0s)

Coq < Goal ((A*unit)->(B*C*unit))==
Coq <      (((A*unit)->((C->unit)*C))*(unit->(A->B))).
1 subgoal

  ============================
   (A*unit->B*C*unit)==((A*unit->(C->unit)*C)*(unit->A->B))

Unnamed_thm < Time IsoProve.
Subtree proved!
Finished transaction in 0 secs (0.06u,0s)

Coq < Goal ((unit->A)->((((B*unit)->A->D)->C)*(B->unit->C)))==
Coq <      ((((A*unit)->B->unit->D)->A->C)*((unit->B)->A->C)).
1 subgoal

  ============================
   ((unit->A)->((B*unit->A->D)->C)*(B->unit->C))
    ==(((A*unit->B->unit->D)->A->C)*((unit->B)->A->C))

Unnamed_thm < Time IsoProve.
Subtree proved!
Finished transaction in 0 secs (0.12u,0s)
```

7.2.7 Observations

Le code de cette tactique est certes plus long que celui de la tactique `Tauto'`, mais il reste plutôt court (126 lignes au total) et très compact. Il est également complètement lisible si bien qu'il peut être facilement maintenu. Un autre atout de cette lisibilité est que l'on peut espérer étendre facilement cette tactique à d'autres λ-calculs typés plus élaborés. Par exemple, on peut imaginer ajouter du polymorphisme [25, 26], des types dépendants [22] ou des modules [1, 2].

Quant aux performances, elles sont très acceptables. On peut s'en apercevoir plus particulièrement sur le dernier exemple de la section précédente, où il y a beaucoup d'opérations à effectuer, aussi bien au niveau de la simplification que de la comparaison, avant d'identifier les deux types.

7.3 Tautologies propositionnelles classiques

Une autre application non triviale de $\mathcal{L}_{tac}$ peut consister à automatiser les preuves de tautologies propositionnelles classiques. Une idée pourrait être d'adapter la tactique `Tauto'` (voir section 7.1), écrite précédemment, sachant qu'en logique propositionnelle classique et en calcul des séquents, toute proposition valide admet une preuve sans coupure et sans contraction. Toutefois, afin de varier sensiblement l'approche, nous nous proposons d'utiliser une démarche plus sémantique, à savoir la méthode de Davis-Putnam-Logemann-Loveland [73]. En effet,contrairement aux méthodes traditionnelles syntaxiques comme les tableaux ou la résolution, la méthode de Davis-Putnam-Logemann-Loveland se base essentiellement sur des affectations de variables propositionnelles et sur des simplifications visant à minimiser les affectations à réaliser[20]. Cependant, il est également possible d'interpréter cette méthode syntaxiquement et nous verrons comment la traduire en une méthode de recherche de preuve[21].

Pour rédiger les deux parties suivantes, nous nous sommes basés sur [37].

7.3.1 Prérequis

L'ensemble des propositions que nous souhaitons décider se définit comme suit :

Définition 7.3.1 (Propositions) *Soit χ, un ensemble infini dit de variables propositionnelles. L'ensemble des propositions est le plus petit ensemble contenant χ, et tel que si Φ et Φ' sont des propositions, alors $\Phi \wedge \Phi'$, $\Phi \vee \Phi'$, $\Phi \rightarrow \Phi'$ et $\neg\Phi$ sont des propositions.*

Nous interpréterons ces propositions dans l'ensemble $\mathbb{B} = \{\top, \bot\}$, dit des valeurs de vérité ou des booléens, où $\top \neq \bot$. Les connecteurs seront interprétés par les fonctions booléennes $\bar{\wedge}$, $\bar{\vee}$, $\bar{\rightarrow}$, de $\mathbb{B} \times \mathbb{B}$ vers $\mathbb{B}$ et $\bar{\neg}$ de $\mathbb{B}$ vers $\mathbb{B}$. Ces fonctions sont représentées par les tables de vérité suivantes :

$\bar{\wedge}$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

$\bar{\vee}$	$\top$	$\bot$
$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$

$\bar{\rightarrow}$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\top$	$\top$

$\bar{\neg}$	
$\top$	$\bot$
$\bot$	$\top$

L'interprétation des propositions se définit formellement de la manière suivante :

Définition 7.3.2 (Affectation, interprétation) *Une affectation ou interprétation ρ est une application de l'ensemble des variables propositionnelles χ vers $\mathbb{B}$.*

La sémantique $[\![\Phi]\!]_\rho$ d'une proposition Φ dans l'affectation ρ est définie par récurrence structurelle sue Φ par :
- $[\![A]\!]_\rho = \rho(A)$, *si* $A \in \chi$
- $[\![\Phi' \wedge \Phi'']\!]_\rho = [\![\Phi']\!]_\rho \bar{\wedge} [\![\Phi'']\!]_\rho$
- $[\![\Phi' \vee \Phi'']\!]_\rho = [\![\Phi']\!]_\rho \bar{\vee} [\![\Phi'']\!]_\rho$
- $[\![\Phi' \rightarrow \Phi'']\!]_\rho = [\![\Phi']\!]_\rho \bar{\rightarrow} [\![\Phi'']\!]_\rho$
- $[\![\neg\Phi']\!]_\rho = \bar{\neg}[\![\Phi']\!]_\rho$

Nous pouvons maintenant définir la notion de satisfiabilité et de validité :

[20]On évite, par là-même, de tomber dans la méthode sémantique naïve qui consiste à énumérer toutes les affectations des variables propositionnelles.

[21]Pour ce faire, nous serons obligés d'introduire un certain nombre de coupures et la preuve obtenue ne pourra pas être garantie sans coupure, contrairement, par exemple, aux preuves fournies par la méthode des tableaux qui peut se voir comme une recherche systématique de preuves sans coupure dans le calcul des séquents. Ce n'est pas un problème car si les méthodes syntaxiques cherchent à éliminer la règle de coupure, c'est pour limiter l'espace de recherche. En effet, avec la coupure, on doit remonter de la conclusion vers les prémisses, en inventant une formule arbitraire qui ouvre l'espace de recherche à toutes les preuves possibles. Ainsi, le problème ne réside pas dans les preuves avec coupures, mais plutôt dans le traitement de la règle de coupure dans la recherche de preuve. De plus, dans notre cas, il sera toujours possible de β-normaliser la preuve obtenue afin d'éliminer toutes les coupures.

Définition 7.3.3 (Satisfiabilité, validité) *Soit Φ une formule propositionnelle et ρ une affectation.*

Nous disons que ρ est un modèle (resp. contre-modèle) de Φ, ou que ρ satisfait (resp. ne satisfait pas) Φ, ou que Φ est vraie (resp. fausse) dans ρ, que nous écrivons $\rho \Vdash \Phi$ (resp. $\rho \nVdash \Phi$), si et seulement si $[\![\Phi]\!]_\rho = \top$ (resp. $[\![\Phi]\!]_\rho = \perp$).

Φ est satisfiable (resp. invalide) si et seulement si Φ est vraie (resp. fausse) dans au moins une affectation.

Φ est valide (resp. insatisfiable) si et seulement si Φ est vraie (resp. fausse) dans toute affectation. Une proposition valide est aussi appelée une tautologie.

7.3.2 Méthode de Davis-Putnam-Logemann-Loveland

La méthode de Davis-Putnam-Logemann-Loveland a été inventée en 1960 par Martin Davis et Hillary Putnam et améliorée en 1963 par Martin Davis, George Logemann et Donald Loveland, dans le cadre plus général du premier ordre. Comme dans la plupart des procédures de preuve automatique, dans cette méthode, si Φ est la formule dont on cherche à prouver la validité, le problème est présenté en niant Φ et en réfutant $\neg\Phi$, c'est-à-dire en montrant que $\neg\Phi$ est insatisfiable[22]. De même que pour la résolution, une étape préliminaire est nécessaire, à savoir que, pour prouver Φ (ou réfuter $\neg\Phi$), nous devons mettre d'abord $\neg\Phi$ en forme normale conjonctive. Définissons la notion de forme normale conjonctive :

Définition 7.3.4 (Formes normales) *Une formule Φ est en forme négation-normale (nnf) si et seulement si elle est construite avec les connecteurs $\wedge$, $\vee$ et $\neg$ uniquement et si toute sous-formule niée $\neg\Phi'$ est telle que Φ' est une variable propositionnelle (les négations ne gouvernent pas les formules composites).*

Un atome est un autre nom pour une variable propositionnelle. Un littéral est une formule de la forme A ou $\neg A$, où A est un atome.

Une formule Φ est en forme normale disjonctive (dnf) si et seulement si elle est en nnf et aucune disjonction n'apparaît comme argument d'une conjonction (Φ est une disjonction n-aire de clauses conjonctives, qui sont des conjonctions m-aires de littéraux).

De façon symétrique, une formule Φ est en forme normale conjonctive (cnf) si et seulement si elle est en nnf et aucune conjonction n'apparaît comme argument d'une disjonction (Φ est une conjonction n-aire de clauses disjonctives, qui sont des disjonctions m-aires de littéraux).

La clause disjonctive vide est notée $\Box$.

Pour abréger, nous dirons clause au lieu de clause disjonctive. Une clause est donc une disjonction de littéraux.

Ces normalisations sont toutes décidables. Pour mettre une formule Φ en nnf, il faut d'abord réécrire toutes les sous-formules $\Phi' \rightarrow \Phi''$ sous la forme $\neg\Phi' \vee \Phi''$, de sorte à obtenir une formule équivalente n'utilisant que $\wedge$, $\vee$ et $\neg$. Ensuite, on utilise les identités de De Morgan :

$$\neg(\Phi' \wedge \Phi'') \leftrightarrow (\neg\Phi') \vee (\neg\Phi'')$$
$$\neg(\Phi' \vee \Phi'') \leftrightarrow (\neg\Phi') \wedge (\neg\Phi'')$$

pour pousser les négations vers l'intérieur de la formule. Maintenant, si Φ est en nnf, nous la transformons en cnf (resp. dnf) en réécrivant toutes les sous-formules $\Phi' \vee (\Phi_1 \wedge \Phi_2)$

[22]Ce point de vue est plutôt traditionnel et finalement assez peu naturel. Sa justification est essentiellement historique.

(resp. $\Phi' \wedge (\Phi_1 \vee \Phi_2)$) en $(\Phi' \vee \Phi_1) \wedge (\Phi' \vee \Phi_2)$ (resp. $(\Phi' \wedge \Phi_1) \vee (\Phi' \wedge \Phi_2)$) et $(\Phi_1 \wedge \Phi_2) \vee \Phi'$ (resp. $(\Phi_1 \vee \Phi_2) \wedge \Phi'$) en $(\Phi_1 \vee \Phi') \wedge (\Phi_2 \vee \Phi')$ (resp. $(\Phi_1 \wedge \Phi') \vee (\Phi_2 \wedge \Phi')$).

L'idée de la méthode de Davis-Putnam-Logemann-Loveland est la suivante. Elle prend en entrée un ensemble S de clauses représentant $\neg\Phi$ (en forme cnf). Φ est valide si et seulement si S est insatisfiable. Si S est vide, alors S est satisfiable, donc Φ est invalide. Si S contient $\Box$[23], il est insatisfiable, donc Φ est valide. Sinon, nous choisissons une variable propositionnelle libre A dans S, et, essentiellement, nous remplaçons A par $\top$ (resp. $\bot$) dans S, et simplifions le tout, ce qui donne un ensemble S_1 (resp. S_2) de clauses : S est alors insatisfiable si et seulement si S_1 et S_2 sont insatisfiables.

Ce qui fait que la méthode de Davis-Putnam-Logemann-Loveland est efficace est qu'elle identifie un certain nombre de cas particuliers où nous n'avons pas à séparer S en S_1 et S_2 selon la valeur de la variable A. En effet, cette séparation multiplie le nombre de sous-problèmes à résoudre par 2 pour chaque variable propositionnelle, ce qui est la cause de l'explosion exponentielle du nombre de sous-problèmes à résoudre.

Les règles sont les suivantes :

Définition 7.3.5 (Tautologies) *Une clause est appelée tautologie si et seulement si elle contient A et $\neg A$, où A est une variable.*

On peut remarquer, en effet, qu'une clause est valide si et seulement si elle contient à la fois A et $\neg A$, pour une certaine variable A. Étant donné que l'on cherchera à rendre S insatisfiable, on pourra éliminer les tautologies de S.

Définition 7.3.6 (Séparation) *Soit S un ensemble de clauses et A une variable propositionnelle.*

Définissons $S[\top/A]$ comme l'ensemble S, où toutes les clauses de la forme $C \vee A$ ont été supprimées, et où toutes les clauses de la forme $C \vee \neg A$ ont été remplacées par C.

Définissons $S[\bot/A]$ comme l'ensemble S, où toutes les clauses de la forme $C \vee A$ ont été remplacées par C, et où toutes les clauses de la forme $C \vee \neg A$ ont été supprimées.

Cette phase consiste à séparer S selon les valeurs possibles de A. Ainsi, ni $S[\top/A]$, ni $S[\bot/A]$ n'ont A comme variable libre.

Définition 7.3.7 (Clause unitaire) *Une clause C est dite unitaire si et seulement si elle contient exactement un littéral.*

Les clauses unitaires sont intéressantes parce que si S contient une clause unitaire, disons A, alors S est satisfiable si et seulement si A est satisfiable. Toute affectation satisfaisant S doit attribuer $\top$ à A, donc S est satisfiable si et seulement si $S[\top/A]$ l'est. Symétriquement, si la clause unitaire est de la forme $\neg A$, S est satisfiable si et seulement si $S[\bot/A]$ l'est. Ainsi, si S contient une clause unitaire, nous n'avons pas besoin d'effectuer une séparation sur A ou $\neg A$, car, dans les deux cas, il n'y a qu'un choix possible.

Définition 7.3.8 (Clauses pures) *Un littéral A (resp. $\neg A$) est dit pur dans un ensemble de clauses S si et seulement si $\neg A$ (resp. A) n'apparaît dans aucune clause de S.*

Une clause est pure dans S si elle contient un littéral pur dans S.

[23] Cette clause vide permet de formaliser syntaxiquement la notion sémantique du faux, puisqu'elle est insatisfiable. On aurait pu se passer de cette notion de clause vide, assez peu naturelle, en étendant la syntaxe avec une constante particulière insatisfiable. Toutefois, on sortirait un peu du cadre de la logique propositionnelle pure et ce, uniquement afin de formaliser une méthode plus sémantique.

Supposons que A (resp. $\neg A$) est pur dans S. $S[\top/A]$ (resp. $S[\bot\ /A]$) est alors un sous-ensemble de S, donc si S est satisfiable, alors $S[\top/A]$ (resp. $S[\bot\ /A]$) l'est aussi. Réciproquement, si $S[\top/A]$ est satisfiable, soit ρ l'affectation satisfaisant toutes ses clauses, et ρ' l'affectation envoyant A sur $\top$ (resp. $\bot$) et toutes les autres variables B vers $\rho(B)$. Alors ρ' satisfait toutes les clauses de S. En effet, pour chaque clause C, soit A n'apparaît pas dans C et C est dans $S[\top/A]$ (resp. $S[\bot\ /A]$), donc ρ' satisfait C, soit C est de la forme $C' \vee A$ (resp. $C' \vee \neg A$), et par définition de ρ' en A, ρ' satisfait C de nouveau. Ainsi, S est satisfiable si et seulement si $S[\top/A]$ (resp. $S[\bot\ /A]$) l'est.

Une autre façon de dire est que si S contient une clause pure, alors S sans cette clause pure est satisfiable si et seulement si S est satisfiable. Nous pouvons donc éliminer les clauses pures de S sans changer sa satisfiabilité.

La procédure de Davis-Putnam-Logemann-Loveland fonctionne comme suit (au départ, S est l'ensemble de clauses correspondant à la cnf de $\neg\Phi$, Φ étant la proposition à prouver) :

1. Éliminer toutes les tautologies de S
2. Si S est vide, retourner "Φ invalide"
3. Si S contient $\square$, retourner "Φ valide"
4. Si S contient une clause pure, l'éliminer de S et aller à l'étape 2
5. Si S contient une clause unitaire A (resp. $\neg A$), remplacer S par $S[\top/A]$ (resp. $S[\bot\ /A]$) et aller à l'étape 2
6. Sinon, choisir une variable de S, puis retourner à l'étape 1 pour $S[\top/A]$ et $S[\bot\ /A]$. Si l'un des deux résultats retourne "Φ invalide", retourner "Φ invalide", sinon retourner "Φ valide".

L'élimination des tautologies se fait en préliminaire et une fois pour toutes, car, aucune des autres phases ne peut créer de tautologies. Ce n'est, par contre, pas le cas des clauses pures dont l'élimination peut rendre d'autres clauses pures. Par exemple, considérons les clauses $A \vee \neg B$, $\neg A \vee \neg B$, $B \vee C$. Parmi ces clauses, la troisième est pure car $\neg C$ n'apparaît pas dans l'ensemble de clauses. On peut donc l'éliminer, ce qui rend les deux premières clauses pures (elles peuvent être, à leur tour, éliminées) car B n'apparaît plus nul part.

7.3.3 Remarques préliminaires sur l'implantation

Nous allons transformer cette méthode sémantique (qui, de fait, est une procédure de décision) en méthode de recherche automatique de preuve. Pour ce faire, nous donnerons, tout d'abord, des équivalents syntaxiques à certaines notions vues précédemment. En particulier, la clause vide $\square$ sera représentée, en `Coq`, par la constante `False` et, en ce qui concerne la séparation ou l'instantiation des clauses unitaires, nous utiliserons les constantes `True` et `False` (pour respectivement $\top$ et $\bot$), que nous substituerons aux variables concernées.

Initialement, pour une formule `P`, nous la transformerons en `~~P` (application d'une forme du tiers exclus : $\forall P.\neg\neg P \to P$), puis en `~P->False` (expansion de `~`, qui, en `Coq`, est une constante s'exprimant en fonction de `False`). Ensuite, nous calculerons la forme cnf de `~P`, `P'`, et nous remplacerons (par une coupure) la proposition à prouver par `P'->False`. Le fait que `(P'->False)->~P->False` n'est pas difficile à montrer puisque toutes les opérations réalisées (transformation de `->` en `\/`, règles de De Morgan et distribution du `\/` par rapport au `/\`) sont purement intuitionnistes et la tactique `Tauto` se chargera de faire la preuve. Ensuite, on introduit `P'` dans le contexte local et on casse tous les connecteurs `/\`, de manière à obtenir un ensemble de clauses dans le contexte (équivalent à notre ensemble S vu précédemment). On cherchera alors à rendre cet ensemble insatisfiable, c'est-à-dire à trouver la constante `False` dans cet ensemble.

Les règles qui éliminent des clauses (tautologies et clauses pures) seront trivialement traitées en enlevant les hypothèses du contexte local concernées (affaiblissement) par la tactique `Clear`. Pour la séparation, on utilisera le tiers exclus pour instantier la variable en question par `True` et `False`. On fera alors des substitutions par ces valeurs dans l'ensemble de clauses et on simplifiera les clauses dans lesquelles on aura substitué. On remplacera les anciennes clauses par ces nouvelles au moyen de coupures. Les preuves de ces coupures seront, à nouveau, prouvées par `Tauto`, puisque les instantiations ainsi que les simplifications (par rapport à `True` et `False`) sont intuitionnistes.

Comme nous avons étendu la syntaxe des propositions avec `True` et `False` pour pouvoir coder cette méthode, une idée naturelle sera de les traiter aussi. Cela pourra être fait, en préliminaire (comme pour les tautologies), en simplifiant l'ensemble de clauses au moyen des mêmes fonctions utilisées après les instantiations.

7.3.4 Gestion des variables propositionnelles

Tout d'abord, nous allons écrire une tactique qui introduit toutes les produits quantifiant sur les variables propositionnelles (de type `Prop`), sachant que l'on supposera que la proposition à résoudre est de la forme :

$$\forall_{\mathcal{P}} \overrightarrow{A_i}.P(\overrightarrow{A_i})$$

où $\forall_{\mathcal{P}}$ quantifie sur l'ensemble des propositions $\mathcal{P}$ et $P(\overrightarrow{A_i})$ est un terme de la logique propositionnelle pour lequel $\overrightarrow{A_i}$ sont les seules variables propositionnelles.

Nous avons donc la tactique d'introduction suivante :

```
1   Recursive Tactic Definition PropIntros :=
2     Try Match Context With
3     | [|- (_:Prop)?] -> Intro;PropIntros.
```

Comme nous pouvons le voir en ligne 3, avec $\mathcal{L}_{tac}$, il est possible de préciser, au niveau du motif de filtrage, que l'on souhaite filtrer un produit dépendant (et pas un produit non dépendant). On peut donner le type du produit, ainsi que le nom de la variable liée bien qu'il soit ignoré lors du filtrage pour gérer la α-conversion. En ligne 2, un `Try` permet d'éviter d'échouer initialement si la proposition à montrer ne dépend d'aucune variable propositionnelle ou si les variables propositionnelles dont elle dépend ont déjà été introduites manuellement. Ce `Try` permet aussi de s'arrêter (lorsque l'on a introduit toutes les variables propositionnelles) sans échec (qui provoquerait un backtrack sur toutes les introductions effectuées).

Toujours en ce qui concerne les variables propositionnelles, nous allons avoir aussi besoin d'une fonction qui donne la liste des variables propositionnelles. Nous aurons besoin de ce type d'information notamment pour l'élimination des clauses pures, l'instantiation des clauses unitaires et la séparation. Cette donnée est dynamique puisque toutes ces règles élimine des variables propositionnelles. Cette fonction sera donc systématiquement appelée dans le traitement de ces règles.

Avant de donner le code de cette fonction, nous avons besoin de définir les listes dans `Type`, qui nous serviront à stocker les variables propositionnelles, car, comme on l'a vu précédemment, pour $\mathcal{L}_{tac}$, les seules structures de données disponibles sont celles de Coq[24]. En effet, dans la bibliothèque standard, les seules listes disponibles (`PolyList.list`) permettent de stocker des éléments de types dans `Set`. Or, ici, nous désirons construire des

[24]Cela semble plutôt raisonnable pour un métalangage dédié au toplevel.

listes avec des éléments de type `Prop`, qui est de type `Type`. Nous avons donc la définition inductive suivante :

```
5   Inductive Tlist [A : Type] : Set :=
6   | nil : (Tlist A)
7   | cons : A->(Tlist A)->(Tlist A).
```

Pour construire la liste des variables propositionnelles, la méthode consiste essentiellement à filtrer les hypothèses de type `Prop` et à les mettre dans une liste en vérifiant au préalable qu'elles n'y sont pas déjà. Ceci est réalisé par les deux définitions suivantes :

```
9   Recursive Meta Definition IsIn id lvar :=
10    Match lvar With
11    | [(nil ?)] -> false
12    | [(cons ? id ?)] -> true
13    | [(cons ? ? ?1)] -> IsIn id ?1.
14
15  Meta Definition PropVarList :=
16    Rec PropVarListRec lvar ->
17      (Match Context With
18       | [id:Prop |- ?] ->
19         Let check=(IsIn id lvar) In
20         (Match check With
21          | [true] -> Fail 1
22          | _ -> PropVarListRec '(cons Prop id lvar))
23       | _ -> lvar) In
24    Fun () -> PropVarListRec '(nil Prop).
```

La fonction `IsIn` (ligne 9) vérifie que la variable (propositionnelle) `id` n'est pas déjà dans la liste `lvar`. Elle utilise le type prédéfini `Coq` des booléens, dont `true` et `false` sont les uniques constructeurs. Pour vérifier que `id` n'est pas déjà dans `lvar`, il suffit d'utiliser un motif directement avec `id` (ligne 12), qui, comme nous l'avons vu précédemment (cf. remarque 7.2.1), sera dynamiquement évalué (selon les valeurs de `id`).

La fonction `PropVarList` (ligne 15) définit une fonction récursive locale `PropVarListRec` (ligne 16) au moyen d'une structure `Rec` ... `In`. Cela permet d'éviter d'imposer à `PropVarList` de donner la valeur d'initialisation, à savoir la liste vide `(nil Prop)` (ligne 24), au moment de l'appel. On peut remarquer aussi que l'appel à `PropVarListRec` est enfermé dans une fermeture factice (ligne 24) par `Fun () ->` ..., car, sinon, l'évaluation de l'application de `PropVarListRec` serait évaluée une fois pour toutes, ce qui fixerait définitivement la valeur de `PropVarList`. En effet, dans ce cas, il s'agit d'une `Meta Definition` (ligne 15) et il n'y a donc pas d'arité cachée comme pour les définitions de tactiques (`Tactic Definition`) qui attendent implicitement un but pour pouvoir s'appliquer.

Pour construire la liste des variables propositionnelles, on filtre, par un `Match Context` (ligne 17), les hypothèses de type `Prop` (ligne 18). On vérifie que l'hypothèse filtrée n'est pas déjà dans la liste en appelant à `IsIn` et en stockant le résultat dans la variable `check` (ligne 19). Si l'hypothèse est dans la liste, c'est-à-dire que `check` vaut `true`, alors il faut étudier la prochaine hypothèse du contexte local. Ceci est fait (et doit être fait) en échouant dans le membre droit de la règle concernée du `Match Context` (lignes 19 à 22). Pour ce faire, on utilise la tactique `Fail` (ligne 21), à qui il faut faire casser une boucle de backtracking (celle du `Match` de la ligne 20), en lui donnant l'argument `1`. En effet, l'utilisation simple de `Fail` (sans argument ou, sous la forme équivalente `Fail 0`) ne ferait échouer que la règle correspondante du `Match` (ligne 21) et le backtracking (qui existe aussi pour la construction `Match`) irait essayer le prochain motif (ligne 22). Étant donné que le motif suivant est `_` et qu'il filtre avec succès systématiquement, on rappellerait `PropVarListRec` en ajoutant la variable à la liste (ligne 22). La prochaine évaluation de `Match Context` donnerait à nouveau la même hypothèse et on aurait exactement le même scénario, ce qui nous ferait boucler. Le `Fail 1` nous permet donc de casser la boucle de backtracking du `Match`, en ressortant avec un `Fail 0` (c'est-à-dire `Fail`) au niveau de `Match Context`, qui peut backtracker en étudiant la prochaine hypothèse. Dans le cas contraire, si l'hypothèse n'est pas dans la liste, c'est-à-dire que `check` vaut `false`, on la met dans la liste et on rappelle `PropVarListRec` (ligne 22). La liste est rendue lorsque `Match Context` ne trouve plus aucune hypothèses de type `Prop` (ligne 23).

7.3.5 Création de l'ensemble de clauses

Comme nous l'avons vu précédemment en prélimaires (section 7.3.3), pour une proposition `P`, on la transforme d'abord en `~~P`, puis en `~P->False`. On met alors `~P` en cnf et on l'introduit en hypothèses en éliminant toutes les conjonctions. Ainsi, le contexte local représente exactement (aux variables de type `Prop` près) l'ensemble des clauses S (de `~P`) manipulé dans la méthode de Davis-Putnam-Logemann-Loveland (voir section 7.3.2). Nous allons décrire ici cette phase de création de l'ensemble de clauses, à savoir la mise en forme cnf et l'élimination des conjonctions.

Mise en forme cnf

Pour mettre en forme cnf, il faut d'abord transformer les $\rightarrow$ en $\vee$ (voir section 7.3.2). Ceci peut être fait par la définition suivante :

```
26  Recursive Meta Definition ImplyNF trm :=
27    Match trm With
28    | [C[?1->?2]] -> ImplyNF 'Inst C[~?1\/?2]
29    | _ -> trm.
```

Le motif de la ligne 28 permet de filtrer des sous-termes de `trm` qui sont des implications et `C` représente le contexte de ces implications (terme avec un "trou"). Dans le membre droit de cette règle, on récupère le contexte et on l'instantie avec la disjonction correspondante au moyen de `Inst`. On rappelle alors `ImplyNF` avec ce nouveau terme pour remplacer d'éventuelles autres implications. Lorsqu'il n'y a plus d'implications (ligne 29), on rend alors le terme inchangé.

Uns fois les implications éliminées, on met en forme nnf en utilisant les règles de De Morgan (voir section 7.3.2), comme suit :

```
31  Recursive Meta Definition NNF trm :=
32    Match trm With
33    | [~(?1/\?2)] ->
34      Let t1=(NNF '~?1)
35      And t2=(NNF '~?2) In
36      't1\/t2
37    | [~(?1\/?2)] ->
38      Let t1=(NNF '~?1)
39      And t2=(NNF '~?2) In
40      't1/\t2
```

```
41     | [~~?1] -> (NNF ?1)
42     | [?1/\?2] ->
43       Let t1=(NNF ?1)
44       And t2=(NNF ?2) In
45       't1\/t2
46     | [?1\/?2] ->
47       Let t1=(NNF ?1)
48       And t2=(NNF ?2) In
49       't1\/t2
50     | _ -> trm.
```

Pour être efficace, l'idée de NNF est, pour chaque motif, de normaliser les sous-termes avant d'appliquer une éventuelle transformation. Les motifs des lignes 33 et 37 correspondent aux règles de De Morgan pour le $\neg$. Le motif de la ligne 41 n'est pas une règle de De Morgan mais utilise le lemme (intuitionniste) de la logique propositionnelle $P \rightarrow \neg\neg P$, où P est une proposition. Les motifs des lignes 42 et 46 permettent de passer au contexte pour le reste des connecteurs, à savoir le $\wedge$ et le $\vee$. Enfin, le motif de la ligne 50 filtre le cas de A et $\neg A$, où A est une variable propositionnelle.

Maintenant que les $\neg$ sont appliqués uniquement aux variables propositionnelles, on peut effectuer la dernière phase de la mise en forme cnf, qui consiste à distribuer les $\vee$ par rapport aux $\wedge$ (voir section 7.3.2). Pour ce faire, on définit trois fonctions. Les deux premières sont dédiées aux connecteurs $\wedge$ et $\vee$ et la troisième est la fonction principale :

```
52   Recursive Meta Definition CNF_And trm :=
53     Match trm With
54     | [(?1/\?2)/\?3] -> Let t=(CNF_And '(?2/\?3)) In '(?1/\t)
55     | _ -> trm.
56
57   Recursive Meta Definition CNF_Or trm :=
58     Match trm With
59     | [?1\/(?2/\?3)] ->
60       Let t1=(CNF_Or '(?1\/?2))
61       And t2=(CNF_Or '(?1\/?3)) In
62       CNF_And '(t1/\t2)
63     | [(?1/\?2)\/?3] ->
64       Let t1=(CNF_Or '(?1\/?3))
65       And t2=(CNF_Or '(?2\/?3)) In
66       CNF_And '(t1/\t2)
67     | [(?1\/?2)\/?3] ->
68       Let t=(CNF_Or '(?2\/?3)) In CNF_Or '(?1\/t)
69     | _ -> trm.
70
71   Recursive Meta Definition CNF trm :=
72     Match trm With
73     | [?1/\?2] ->
74       Let t1=(CNF ?1)
75       And t2=(CNF ?2) In
76       CNF_And 't1/\t2
77     | [?1\/?2] ->
78       Let t1=(CNF ?1)
```

```
79        And t2=(CNF ?2) In
80        CNF_Or 't1\/t2
81      | _ -> trm.
```

La fonction `CNF_And` est chargée uniquement d'associer à droite les conjonctions n-aires (ligne 54). La fonction `CNF_Or` effectue les distributions de $\vee$ par rapport à $\wedge$ à gauche et à droite (ligne 59 et 63). De même que `CNF_And`, elle associe aussi à droite les disjonctions n-aires (ligne 67). La fonction `CNF` permet, selon les connecteurs filtrés, d'appeler `CNF_And` ou `CNF_Or`, en ayant au préalable simplifié les sous-termes. Ces fonctions ne sont pas mutuellement récursives et cela permet d'obtenir un code plutôt efficace par rapport à des solutions certes plus naturelles mais aussi plus naïves. Par exemple, la ligne 68 aurait pu être simplement remplacée par :

```
CNF '(?1\/?2\/?3)
```

avec une déclaration mutuellement récursive de `CNF_Or` et `CNF`, au moyen d'une construction `Recursive ... And ...`. Cependant, on voit bien que `CNF` va déstructurer le $\vee$ avec `?1` et `?2\/ ?3`. Il simplifie ensuite `?1` (déjà simplifié) puis `?2\/ ?3` en déstructurant le $\vee$ en `?2` et `?3`. Il les simplifie alors aussi (alors que, de même que `?1`, ils ont déjà été simplifiés) puis appelle `CNF_Or` (à ce moment, on a fait l'équivalent du `(CNF_Or '( ?2\/ ?3))` de la ligne 68). Enfin, on appelle à nouveau `CNF_Or` avec `?1` et le précédent résultat (ce qui revient à faire le `CNF_Or '( ?1\/t)` de la ligne 68). Ainsi, on a bien l'équivalent de la ligne 68, mais il y a bien exactement 5 appels à `CNF` inutiles.

Il ne reste plus qu'à écrire la tactique qui remplace la proposition `~P->False` en `P'->False`, où `P'` est la cnf de `~P` :

```
83  Tactic Definition MakeCNF :=
84    Match Context With
85    | [|- ?1->False] ->
86      Let t=(CNF (NNF (ImplyNF ?1))) In
87      Cut t->False;[Tauto|Idtac].
```

L'idée de `MakeCNF` consiste à calculer la cnf de `~P` (ligne 86) et à l'introduire par une coupure. Pour prouver que `(P'->False)->~P->False`, il suffit alors d'utiliser la tactique `Tauto`, puisque toutes les transformations effectuées lors de la mise en forme cnf sont purement intuitionnistes.

Élimination des conjonctions

Il s'agit là d'éliminer toutes les conjonctions de `P'`, que l'on supposera déjà introduit dans le contexte local :

```
89  Recursive Tactic Definition Splits :=
90    Try Match Context With
91    | [id:?1/\?2 |- ?] -> Elim id;Intros;Clear id;Splits.
```

Le principe de `Splits` consiste à filtrer tous les $\wedge$ (à la racine) en hypothèses (ligne 91) et à les casser. De même que pour la tactique `PropIntros` (voir section 7.3.4), le `Try` (ligne 90) permet d'éviter d'échouer si la cnf est dégénérée, c'est-à-dire sans aucun $\wedge$.

7.3.6 Simplification

Nous allons avoir besoin de fonctions de simplification par rapport à `True` et `False`, notamment lors des instantiations des clauses unitaires ou des séparations. Par ailleurs, comme nous avons décidé de traiter aussi les constantes `True` et `False` (voir section 7.3.3), ces fonctions serviront également à simplifier initialement l'ensemble de clauses. De même que pour `CNF` (voir section 7.3.5), cette simplification est stratifiée selon les connecteurs logiques (restants) :

```
93   Meta Definition SimplifOr trm :=
94     Match trm With
95     | [True\/?] -> True
96     | [?\/True] -> True
97     | [False\/?1] -> ?1
98     | [?1\/False] -> ?1
99     | _ -> trm.
100
101  Meta Definition SimplifNot trm :=
102    Match trm With
103    | [~True] -> False
104    | [~False] -> True
105    | _ -> trm.
106
107  Recursive Meta Definition Simplif trm :=
108    Match trm With
109    | [?1\/?2] ->
110      Let t1=(Simplif ?1)
111      And t2=(Simplif ?2) In
112      SimplifOr 't1\/t2
113    | [~?1] ->
114      Let t=(Simplif ?1) In
115      SimplifNot '~t
116    | _ -> trm.
```

La simplification s'effectue selon les connecteurs logiques $\vee$ et $\neg$ (les $\wedge$ ont été éliminés par la tactique `Splits`, définie précédemment dans la section 7.3.5). Dans les fonctions `SimplifOr` et `SimplifNot`, on reconnaît les règles de simplification du $\vee$ (lignes 95 à 98) et du $\neg$ (lignes 103 et 104), complètement isomorphes à leurs tables de vérité (voir section 7.3.1). La fonction `Simplif` appelle ces fonctions selon l'opérateur filtré en ayant au préalable simplifié les sous-termes (lignes 109 à 115).

Comme nous l'avons dit plus haut, nous nous proposons de traiter le cas où la proposition initiale contient `True` ou `False`. Pour ce faire, nous allons utiliser, en préliminaire, les fonctions de simplification que nous venons de décrire, avant d'appeler la procédure de Davis-Putnam-Logemann-Loveland. En effet, il faudra toujours simplifier avant d'appeler la procédure lorsqu'il y aura potentiellement des occurrences de `True` et de `False`, que ce soit au départ ou après des instantiations (clauses unitaires ou séparations). Cela se justifie dans la mesure où ces simplifications pourront soit restreindre l'ensemble de clauses (élimination des clauses `True`), soit apporter la solution (obtention de clauses `False`). Ainsi, un invariant est que la procédure de Davis-Putnam-Logemann-Loveland sera systématiquement appelée avec un ensemble de clauses, où `True` n'apparaît pas et si `False` apparaît dans une clause alors cette clause est réduite à `False`. La simplification initiale se fait de la manière suivante :

```
118 Recursive Tactic Definition InitSimplif :=
119   Match Context With
120   | [id:C[True] |- ?] ->
121     Let t=Inst C[True] In
122     (Match t With
123     | [True] -> Clear id;Try InitSimplif
124     | _ ->
125       Let nt=(Simplif t) In
126       Cut nt;[Intro after id;Clear id;Try InitSimplif|Tauto])
127   | [id:C[False] |- ?] ->
128     Let t=Inst C[False] In
129     (Match t With
130     | [False] -> Fail 1
131     | _ ->
132       Let nt=(Simplif t) In
133       Cut nt;[Intro after id;Clear id;Try InitSimplif|Tauto]).
```

En ligne 120, on essaie de filtrer `True` en tant que sous-terme. Ensuite, on instantie (ligne 121) de manière à obtenir la clause filtrée et on vérifie si cette clause est réduite à `True`. Si c'est le cas, elle ne sert à rien (puisqu'on cherche `False`) et on l'élimine (ligne 123). Dans le cas contraire (ligne 124), on la simplifie (ligne 125) et on lui substitue la clause simplifiée (ligne 126). Pour montrer que la clause simplifiée implique la clause initiale, on utilise la tactique `Tauto`, dans la mesure où les simplifications effectuées par `Simplif` sont toutes intuitionnistes. Quant à False, de même, on essaie, en ligne 127, de la filtrer en tant que sous-terme. On l'instantie alors (ligne 128) pour obtenir la clause qui a été filtrée et on vérifie si elle est pas réduite à `False`. Si la clause est exactement `False`, alors c'est la solution et il ne faut la retirer. L'idée est donc d'échouer avec `Fail 1` (ligne 130), de manière à sortir de la boucle de backtracking du `Match` (ligne 129) et faire échouer la règle du `Match Context` (ligne 119). Le backtracking du `Match Context` permet alors d'aller explorer les prochaines hypothèses. Si au lieu d'échouer, on avait naïvement rappelé `InitSimplif`, le `Match Context` aurait filtré la même hypothèse et l'évaluation aurait été exactement la même, ce qui aurait fait boucler la tactique. Dans le cas où la clause initiale n'est pas réduite à `False` (ligne 131), le principe est complètement similaire au traitement de la constante `True`.

7.3.7 Règles de la procédure

Nous allons maintenant décrire le code correspondant aux différentes étapes de la procédure de Davis-Putnam-Logemann-Loveland (voir section 7.3.2).

Ensemble de clauses vide

Si la proposition à montrer est invalide, la procédure sera appelée, à un certain point de l'évaluation, avec un ensemble de clauses vide. L'idée est donc d'écrire une tactique qui échoue si le contexte local est vide. Cependant, même si toutes les clauses ont été éliminées, le contexte local ne sera potentiellement jamais vraiment vide puisque la proposition peut contenir des variables propositionnelles, qui sont introduites, au préalable, dans le contexte local (voir la tactique `PropIntros`, section 7.3.4). Cette tactique doit donc faire abstraction des variables propositionnelles du contexte local dans son test :

```
135 Tactic Definition Empty :=
```

```
136     Match Context With
137     | [_:?1 |- ?] ->
138       (Match ?1 With
139        | [Prop] -> Fail 1
140        | _ -> Idtac).
```

Le `Match Context` (ligne 136) possède un unique motif (ligne 137) qui filtre toutes les hypothèses. Si le contexte local est vide, il y a échec direct. Dans le cas contraire, il faut tester si l'hypothèse filtrée est une variable propositionnelle. Si c'est le cas (ligne 139), il faut échouer avec `Fail 1`, de manière à sortir de la boucle de backtracking du `Match` (ligne 138) et à faire échouer la règle du `Match Context` qui va aller étudier la prochaine hypothèse. Si l'hypothèse n'est une variable propositionnelle (ligne 140), alors le contexte local n'est pas vide et on sort avec `Idtac`.

Conclusion

Si la proposition à prouver est valide, toutes les branches de l'arbre d'évaluation se termineront sur des feuilles, où le contexte local contient la clause `False`. Il s'agit donc d'écrire une tactique qui cherche `False` dans le contexte local :

```
142   Tactic Definition Conclude :=
143     Match Context With
144     | [id:False |- ?] -> Exact id.
```

En ligne 144, on filtre les hypothèses de type `False` et on applique alors la tactique `Exact` qui prend en argument un terme de type convertible à la conclusion (ici, on a même une égalité syntaxique).

Élimination des tautologies

Il s'agit d'éliminer toutes les clauses qui contiennent à la fois A et $\neg A$, où A est une variable propositionnelle :

```
146   Recursive Tactic Definition GrepLit lit trm :=
147     Match trm With
148     | [lit] -> Idtac
149     | [lit\/?1] -> Idtac
150     | [?\/?1] -> GrepLit lit ?1.
151
152   Recursive Tactic Definition IsTauto trm :=
153     Match trm With
154     | [?1\/?2] ->
155       (Match ?1 With
156        | [~?3] -> First [GrepLit ?3 ?2|IsTauto ?2]
157        | _ -> First [GrepLit '~?1 ?2|IsTauto ?2]).
158
159   Recursive Tactic Definition Tautology :=
160     Match Context With
161     | [id:?1 |- ?] -> IsTauto ?1;Clear id;Try Tautology.
```

La tactique `GrepLit` (ligne 146) permet de dire si `lit` appartient à la clause `trm` ou non. Si `lit` est un littéral de la clause `trm`, alors elle rend `Idtac`, sinon elle échoue[25]. La tactique `IsTauto` vérifie si la clause `trm` est une tautologie. Pour cela, elle isole le premier littéral de la clause (ligne 154). Si ce littéral est de la forme $\neg A$ (ligne 156), où A est une variable propositionnelle, alors on doit chercher A dans le reste de la clause (appel à `GrepLit`). Si cette recherche échoue, on vérifie si le reste de la clause est une tautologie (appel récursif `IsTauto`). Réciproquement, si le littéral est réduit à une variable propositionnelle A (ligne 157), alors on recherche $\neg A$ dans le reste de la clause (toujours avec `GrepLit`). S'il y a échec, de même que dans le premier cas, on étudie le reste de la clause (appel de `IsTauto`). Enfin, la tactique `Tautology` (ligne 159) vérifie si toutes les clauses du contexte local sont des tautologies. Pour cela, elle appelle `IsTauto` (ligne 161). En cas de succès, la clause est éliminée par `Clear` et on recommence pour étudier d'autres clauses en rappelant `Tautology`. En cas d'échec, le backtracking du `Match Context` permet d'aller filtrer la prochaine clause. Il est à noter que `Tautology` échoue s'il n'y a aucune tautologie dans le contexte local et rend `Idtac` (assuré par le `Try` de la ligne 161) s'il y en a au moins une[26].

Élimination des clauses pures

Dans cette phase, il faut éliminer toutes les clauses qui contiennent un littéral A (resp. $\neg A$), où A est une variable propositionnelle, tel que $\neg A$ (resp. A) n'apparaît dans aucune clause[27] :

```
163 Tactic Definition GrepClauses id :=
164   Match Context With
165   | [_:?1 |- ?] -> GrepLit id ?1.
166
167 Recursive Tactic Definition ClearPureClause id :=
168   Try Match Context With
169   | [idh:[id] |- ?] -> Clear idh;ClearPureClause id.
170
171 Tactic Definition PureClause lvar :=
172   Match lvar With
173   | [(cons ? ?1 ?2)] ->
174     First [GrepClauses ?1;GrepClauses '~?1|ClearPureClause ?1;Clear ?1];
175       Try (PureClause ?2).
```

La tactique `GrepClauses` (ligne 63) vérifie que le littéral `id` est au moins dans une des clauses du contexte local. Elle utilise, pour cela, la tactique `GrepLit` (ligne 165), décrite

[25] De manière naïve, on aurait pu implanter `GrepLit` de la manière suivante :

```
Tactic Definition GrepLit lit trm :=
   Match trm With
   | [[lit]] -> Idtac.
```

Mais dans le cas où l'on recherche le littéral A, où A est une variable propositionnelle, on pourrait filtrer A en tant que sous-terme de $\neg A$, ce qui ne correspond pas au littéral recherché. Il faut donc écrire la tactique suivant la structure d'une clause, à savoir une disjonction n-aire.

[26] L'information d'échec est ici complètement pertinente et permet de savoir si des tautologies ont été éliminées. On aurait pu faire rendre systématiquement `Idtac` à `Tautology` (en déplaçant le `Try` de la ligne 161 devant le `Match context` de la ligne 160), mais l'appelant aurait été obligé d'utiliser `Progress` pour avoir cette information, ce qui est, sans doute, moins pratique.

[27] En fait, on pourrait même préciser dans aucune autre clause, puisque si c'est le cas, on a alors une tautologie et elle a déjà due être éliminée.

précédemment dans l'élimination des tautologies. La tactique `ClearPureClause` (ligne 167) élimine toutes les clauses pures contenant le littéral pur `id`. Enfin, la tactique `PureClause` élimine toutes les clauses pures du contexte local. L'idée est de raisonner sur la liste des variables propositionnelles représenté par le paramètre `lvar`. Pour une variable propositionnelle donnée A, on cherche, au moyen de `GrepClauses` (ligne 174), le littéral A dans les clauses, puis $\neg A$. En cas de succès des deux recherches, A et $\neg A$ ne sont purs dans aucune clause et on se rappelle récursivement pour traiter les autres variables (ligne 175). Si l'une ou l'autre des recherches a échoué, alors, soit A, soit $\neg A$ est pur dans toutes les clauses où il apparaît. On élimine alors (ligne 174) toutes les clauses contenant A avec `ClearPureClause` (ligne 174), ainsi que la variable elle-même avec `Clear`[28].

Instantiation des clauses unitaires

Il s'agit de trouver, dans l'ensemble de clauses S, une clause de la forme A (resp. $\neg A$), où A est une variable propositionnelle, et de l'instantier en $S[\top/A]$ (resp. $S[\bot/A]$). Avant de traiter les clauses unitaires, il nous faut définir l'instantiation d'une variable propositionnelle. Cela consiste à donner la valeur `True` ou `False` à la variable choisie, de substituer cette valeur à la variable dans toutes les clauses, puis de simplifier les clauses modifiées. La variable peut alors être éliminée de la liste des variables propositionnelles. On a donc les définitions suivantes :

```
177  Tactic Definition MakeInst trm := Elim (classic (trm<->True));Intro.
178
179  Tactic Definition ElimTrue :=
180    Repeat
181      Match Context With
182      | [id:True |- ?] -> Clear id.
183
184  Recursive Tactic Definition Instantiation var val :=
185    Match Context With
186    | [id:C[var] |- ?] ->
187      Let t=Inst C[val] In
188      Let nt=(Simplif t) In
189      Cut nt;[Intro after id;Clear id;Instantiation var val|Tauto]
190    | [id:?<->? |- ?] -> Clear id;Clear var;ElimTrue.
```

La tactique `MakeInst` (ligne 177) permet de justifier les instantiations qui sont effectuées sur l'ensemble de clauses. Plus précisément, elle prend un littéral `trm`, représentant la clause unitaire, puis applique le tiers exclus (lemme `classic`, ligne 177) pour faire apparaître deux cas `trm<->True` (instantiation à `True`) et `~(trm<->True)` (équivalent à `Trm<->False`, instantiation à `False`). La tactique `ElimTrue` (ligne 179) élimine toutes les clauses réduites à `True` (elles sont, en effet, inutiles car on cherche à prouver `False`). La tactique `Instantiation` (ligne 184) prend une variable propositionnelle `var` et l'instantie par la valeur `val` (normalement `True` ou `False`) dans toutes les clauses du contexte local. Pour cela, en ligne 186,

[28] Il peut y avoir aussi le cas (un peu pathologique) où la variable n'apparaît pas du tout dans les clauses. Ceci peut arriver lorsque l'utilisateur définit initialement des variables propositionnelles qu'il n'utilise pas dans la proposition à montrer. Par exemple :

$$\forall_{\mathcal{P}} A.\forall_{\mathcal{P}} B.\forall_{\mathcal{P}} C.A \wedge B \rightarrow A$$

Ce n'est pas un problème dans la mesure où `ClearPureClause` n'échoue jamais et `Clear` pourra alors éliminer cette variable.

on filtre les clauses dont `var` est un sous-terme. Ensuite, on effectue l'instantiation avec `val` (ligne 187), on simplifie le terme obtenu (ligne 188), puis on la substitue à l'ancienne clause (non instantiée) (ligne 189). Pour montrer que la clause instantiée et simplifiée implique bien la clause originale, il suffit d'utiliser `Tauto` (ligne 189), puisque l'instantiation et la simplification sont prouvables de manière intuitionniste. Toutefois, cela n'est possible que si l'on suppose, à ce stade, qu'il y a une hypothèse de la forme `trm<->True` (préalablement introduite) permettant de justifier l'instantiation. La règle de la ligne 190 permet justement d'éliminer cette hypothèse, qui ne sert qu'à justifier l'instantiation, et, puisque toutes les instantiations ont été faites, on peut appeler `ElimTrue`, pour éliminer les éventuelles clauses `True` générées.

L'instantiation des clauses unitaires se fait alors de la manière suivante :

```
192  Tactic Definition UnitClauseVar var :=
193    Match Context With
194    | [_:var |- ?] -> MakeInst var;[Instantiation var True|Tauto]
195    | [_:~var |- ?] -> MakeInst '~var;[Instantiation var False|Tauto].
196
197  Recursive Tactic Definition UnitClause lvar :=
198    Match lvar With
199    | [(nil ?)] -> Fail
200    | [(cons ? ?1 ?2)] -> (UnitClauseVar ?1) Orelse (UnitClause ?2).
```

La tactique `UnitClauseVar` (ligne 192) prend en argument une variable propositionnelle `var` et vérifie s'il y a des clauses unitaires selon `var`. Il y a alors deux cas possibles. Si la clause unitaire est de la forme `var` (ligne 194), on sépare en deux branches (avec `MakeInst`). Dans la première, le lemme `var<->True` est introduit dans le contexte local pour justifier les instantiations, puis on effectue les instantiations à `True` et les simplifications (avec `Instantiation`). Dans la deuxième, on a le lemme `(var<->True)` (équivalent à `var<->False`) et on peut trivialement conclure par `Tauto`, puisque `var` appartient à l'ensemble de clauses. Si la clause unitaire est de la forme `~var` (ligne 195), c'est complètement similaire. Les lemmes introduits sont `~var<->True` (équivalent à `var<->False`) et `~(~var<->True)` (équivalent à `var<->True`). Les instantiations se font à `False` et la tactique `Tauto` peut conclure car on a un lemme équivalent à `~var<->False` en hypothèses, ainsi que la clause `~var`. La tactique `UnitClause` (ligne 197) est chargée de traiter au plus une clause unitaire du contexte local. Pour ce faire, elle utilise la liste des variables propositionnelles `lvar` (ligne 197) et traite les clauses unitaires selon chaque variable de cette liste avec `UnitClauseVar` (ligne 200). Il est à noter qu'avec le `Orelse` de la ligne 200, dès qu'une clause unitaire a pu être instantiée, on s'arrête. En effet, cette instantiation a modifié l'ensemble de clauses et, potentiellement, cet ensemble peut être vide, la solution a pu être générée ou des clauses pures (à éliminer) ont pu être créées.

Séparation

Cette étape est le pire des cas (lorsque l'on ne peut pas échouer ou conclure et lorsqu'aucune autre simplification ne peut plus être effectuée) et consiste à séparer l'ensemble de clauses S, selon une variable propositionnelle A choisie, en deux instantiations $S[\top/A]$ et $S[\perp/A]$. L'instantiation d'une variable propositionnelle ayant déjà été définie pour traiter les clauses unitaires, la séparation s'exprime donc directement comme suit :

```
202  Tactic Definition MakeSplit lvar :=
203    Match lvar With
```

```
204      | [(nil ?)] -> Fail
205      | [(cons ? ?1 ?)] ->
206        MakeInst ?1;[Instantiation ?1 True|Instantiation ?1 False].
```

La tactique `MakeSplit` (ligne 202) prend la liste des variables propositionnelles `lvar` en argument pour choisir une variable selon laquelle séparer. Si la liste est vide (ligne 204), on ne peut faire aucune séparation et on échoue avec `Fail`. Si la liste contient au moins une variable (ligne 205), on effectue la séparation selon cette variable. L'heuristique pour choisir la variable de séparation est donc simple et consiste à choisir la première variable de la liste. La séparation se fait au moyen de `MakeInst` (ligne 206), définie précédemment et qui introduit les lemmes permettant de justifier les instantiations. Enfin, les instantiations et les simplifications sont effectuées, par `Instantiation` (ligne 206), dans chaque branche issue de la séparation, l'une avec `True` et l'autre avec `False`.

7.3.8 Tactique principale

Maintenant que les différentes étapes de la méthode de Davis-Putnam-Logemann-Loveland ont été définies, nous pouvons donner le code de la procédure :

```
208  Recursive Tactic Definition ProofProc :=
209    Empty;Conclude Orelse
210      ((Repeat Progress (PureClause (PropVarList ())));
211        ((UnitClause (PropVarList ()))
212         Orelse (MakeSplit (PropVarList ())));ProofProc).
213
214  Tactic Definition CTauto :=
215    Unfold iff;PropIntros;Apply NNPP;Red;MakeCNF;Intro;Splits;
216      Try InitSimplif;Try Tautology;ProofProc.
```

La tactique `ProofProc` (ligne 208) correspond exactement aux étapes 2 à 6 de la procédure, décrites section 7.3.2. On essaie d'abord les cas d'arrêts (ligne 209) avec `Empty` pour savoir si le contexte est vide (invalide) ou `Conclude` pour voir si `False` est dans le contexte (valide). Si les cas d'arrêts sont infructueux, on élimine toutes les clauses pures avec `PureClause` (ligne 210). Puis, on essaie d'instantier une clause unitaire avec `UnitClause` (ligne 211). Si cela marche, on recommence en rappelant `ProofProc` (ligne 212). Sinon, on tente de séparer selon une variable propositionnelle avec `MakeSplit` (ligne 212). Soit c'est possible et on réitère avec `ProofProc` (ligne 212), soit il n'y a plus variables et `ProofProc` échoue. La tactique `CTauto` (ligne 214) prépare l'appel au noyau récursif de la méthode (`ProofProc`). En ligne 215, on transforme les expressions de la forme $A \leftrightarrow B$, où A et B sont des propositions, en $A \rightarrow B \wedge B \rightarrow A$ avec `Unfold iff`. Ensuite, on introduit les quantifications sur les variables propositionnelles avec `PropIntros`. Si la proposition obtenue en conclusion est `P`, `Apply NNPP` permet de la transformer en `~~P`, sachant que `NNPP` est une forme de tiers exclus, à savoir $\forall_{\mathcal{P}} P.\neg\neg P \rightarrow P$. La tactique `Red` permet de déplier le `~` (constante de `Coq`) de tête pour obtenir `~P->False`, à prouver. `MakeCNF` produit la nouvelle conclusion `P'->False`, où `P'` est la forme cnf de `~P`. `Intro` permet d'introduire la forme cnf dans le contexte local et `Splits` la déstructure pour obtenir un ensemble. En ligne 216, on simplifie par les constantes `True` et `False` avec `InitSimplif`, avant d'éliminer les tautologies avec `Tautology`. Enfin, on peut rentrer dans la boucle itérative de la procédure en appelant `ProofProc`.

7.3.9 Exemples

Nous allons maintenant donner quelques exemples de tautologies propositionnelles classiques et montrer le résultat de l'application de la tactique CTauto que nous venons de décrire. Tous les exemples ont été réalisés dans la version 7.1 de Coq (en natif). La machine utilisée est un Intel Pentium III à 600 MHz, sous Linux Mandrake 7.2. On suppose que toutes les définitions précédentes ont déjà été évaluées par le toplevel de Coq.

```
Coq < Goal (A:Prop)A\/~A.
1 subgoal

  ============================
   (A:Prop)A\/~A

Unnamed_thm < Time CTauto.
Subtree proved!
Finished transaction in 0 secs (0.04u,0s)

Coq < Goal (A,B:Prop)~~((A\/B)->(A\/B)).
1 subgoal

  ============================
   (A,B:Prop)~~(A\/B->A\/B)

Unnamed_thm < Time CTauto.
Subtree proved!
Finished transaction in 0 secs (0.13u,0s)

Coq < Goal (A,B,C:Prop)(B/\C)\/(~A/\(B->~C))\/A.
1 subgoal

  ============================
   (A,B,C:Prop)B/\C\/~A/\(B->~C)\/A

Unnamed_thm < Time CTauto.
Subtree proved!
Finished transaction in 0 secs (0.27u,0s)

Coq < Goal (A,B,C,D:Prop)
Coq <      ~((A/\(C->~B)/\D)\/((A/\~D)/\(~C\/~B)))->A->D->B/\C.
1 subgoal

  ============================
   (A,B,C,D:Prop)~(A/\(C->~B)/\D\/(A/\~D)/\(~C\/~B))->A->D->B/\C

Unnamed_thm < Time CTauto.
Subtree proved!
Finished transaction in 17 secs (16.3u,0.07s)
```

7.3.10 Remarques

Le code de la tactique `CTauto` est significativement plus long (216 lignes) que celui de `Tauto'` et `IsoProve`. Cela s'explique dans la mesure où les différentes étapes de la méthode de Davis-Putnam-Logemann-Loveland manipulent beaucoup le contexte local, et ce, de manière non triviale, en ajoutant un certain nombre de contraintes. Pour gagner un peu de compacité, il faudrait certainement des motifs de plus haut niveau, avec, par exemple, la possibilité de dire que l'on veut filtrer une hypothèse avec le motif p_1 en interdisant qu'une hypothèse puisse être filtrée avec le motif p_2 (en créant d'éventuelles dépendances entre p_1 et p_2). Avec un tel système, le filtrage de clauses pures deviendrait bien plus immédiat. Par ailleurs, comme il s'agit d'une méthode sémantique, il est nécessaire de gérer les affectations partielles, ce qui est, syntaxiquement, plus complexe.

Cependant, compte-tenu de la procédure à implanter, on peut considérer ce code comme suffisamment compact, et on peut raisonnablement penser qu'un code équivalent en Objective Caml serait au moins trois fois plus long. On peut également observer que, comme pour les tactiques précédentes, le code est complètement lisible et donc facilement maintenable.

En ce qui concerne les performances, pour les trois premiers exemples, elles s'avèrent être très bonnes, mais, dès que la taille de la proposition (typiquement, le nombre de connecteurs) augmente, elles peuvent devenir très moyennes. La phase particulièrement critique est la mise en forme cnf, comme on peut le voir dans la session suivante :

```
Coq < Goal (A,B,C,D:Prop)
Coq <      ~((A/\(C->~B)/\D)\/((A/\~D)/\(~C\/~B)))->A->D->B/\C.
1 subgoal

  ============================
   (A,B,C,D:Prop)~(A/\(C->~B)/\D\/(A/\~D)/\(~C\/~B))->A->D->B/\C

Unnamed_thm < Time Unfold iff;PropIntros;Apply NNPP;Red;MakeCNF.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  D : Prop
  ============================
   (~A\/C\/~D)/\(~A\/B\/~D)/\(~A\/D\/C)/\(~A\/D\/B)/\A/\D/\(~B\/~C)
   ->False
Finished transaction in 12 secs (12.19u,0s)
```

La mise en forme cnf prend, en effet, plus des deux-tiers du temps global dans l'application de `CTauto`. Cela semble plutôt normal, dans la mesure où, si la mise en forme nnf est linéaire en le nombre de connecteurs de la proposition, la distribution du $\vee$ par rapport au $\wedge$ est exponentielle. Cela tend à conforter l'idée que les méthodes sémantiques sont particulièrement inefficaces, même si la méthode de Davis-Putnam-Logemann-Loveland est l'une des plus rapides que l'on connaisse aujourd'hui.

Chapitre 8

La tactique `Field`

Une autre utilisation non triviale de $\mathcal{L}_{tac}$, plus significative que les automatisations vues au chapitre 7, a été effectuée pour coder la tactique `Field`, qui se propose d'automatiser les preuves d'égalités sur les corps commutatifs. La principale motivation était de pouvoir faciliter les preuves utilisant les nombres réels et, par là-même, le développement de la librairie des nombres réels[1], bien qu'en principe, cette tactique peut s'appliquer à tout corps commutatif. L'idée de l'algorithme consiste à se débarrasser des inverses afin de pouvoir se ramener à l'utilisation de la procédure de décision déjà existante sur les anneaux abéliens (`Ring`). L'élimination des inverses se fait de manière complètement réflexive et la réflexion est réalisée au moyen de $\mathcal{L}_{tac}$, ce qui permet de remarquer l'adéquation de ce nouveau langage de tactiques pour ce type de développements.

La tactique `Field` a été développée dans la version V7 de `Coq` et fait partie de la distribution du système. Ce travail a été réalisé en collaboration avec Micaela Mayero et a fait l'objet d'une publication [24] aux JFLA'2001, de laquelle ce chapitre s'inspire en partie.

8.1 Motivations

La théorie des nombres réels dans le système `Coq` est axiomatique[2] et les preuves se font à grand renfort de réécritures[3], ce qui fait grossir la taille des scripts ainsi que les termes preuves. La tactique `Ring` qui permet de décider d'égalités sur les anneaux abéliens a permis de résoudre partiellement ce problème et se révèle très utile dans les preuves ne faisant pas intervenir l'inverse. On peut l'utiliser dans des théorèmes auxiliaires triviaux servant à compléter la théorie comme :

$$\forall x, y \in \mathbb{R}. x + y = 0 \to y = (-x)$$

où pas moins de cinq réécritures sont nécessaires pour résoudre sans la tactique `Ring`. Mais, on peut aussi l'utiliser dans des théorèmes non triviaux où l'inverse n'intervient pas, comme le théorème des 3 intervalles [58].

Cependant, `Ring` ne règle pas le cas des égalités avec inverses qui, si elles ne sont pas difficiles à montrer, n'en restent pas moins très fastidieuses à résoudre. Ces preuves sont

[1] Cette librairie fait partie de la distribution du système `Coq`.

[2] Ce choix provient d'un souci non seulement de simplicité mais aussi de rapidité dans le développement de la théorie.

[3] Une construction des nombres réels ne changerait pas ce phénomène car $\mathbb{R}$ ne serait pas non plus un type inductif sur lequel on pourrait calculer. Voir, par exemple, [42] pour s'en rendre compte.

d'autant plus pénibles à construire qu'elles ralentissent de manière conséquente le développement de la théorie des nombres réels. En effet, les théorèmes sur les limites et les dérivées font systématiquement intervenir des égalités avec des inverses dans les "preuves ε"[4] lorsque l'on veut composer, additionner, ... Ces égalités sont relativement triviales et alourdissent considérablement les preuves fondamentales de limites et de dérivées. Un exemple typique est celui de la dérivée de l'addition où l'on doit montrer que la somme des dérivées est égale à la dérivée de la somme. Pour ce faire, on prend deux fonctions f et g ainsi que leurs dérivées en x_0, c'est-à-dire $f'(x_0)$ et $g'(x_0)$. Par définition, les deux dérivées peuvent s'exprimer comme suit :

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$g'(x_0) = \lim_{x \to x_0} \frac{g(x) - g(x_0)}{x - x_0}$$

En utilisant le théorème d'addition des limites, on obtient directement :

$$f'(x_0) + g'(x_0) = \lim_{x \to x_0} \left(\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} \right)$$

En utilisant la définition de la limite, on a pour tout domaine D de $\mathbb{R}$:

$$\forall \varepsilon > 0,\ \exists \alpha > 0,\ \forall x \in D \backslash x_0,\ \text{si } |x - x_0| < \alpha \text{ alors}$$

$$\left| \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} - (f'(x_0) + g'(x_0)) \right| < \varepsilon$$

Maintenant, il suffit de montrer l'égalité suivante :

$$\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} = \frac{f(x) + g(x) - (f(x_0) + g(x_0))}{x - x_0}$$

pour conclure que $f'(x_0) + g'(x_0)$ et $(f + g)'(x_0)$ coïncident. Cette dernière égalité est clairement triviale mais la preuve formelle l'est beaucoup moins[5]. Il faut d'abord réduire au même dénominateur (4 réécritures) puis montrer l'égalité sur les numérateurs (6 réécritures). L'utilisation de `Ring` sur les numérateurs économise des réécritures et au total, cette égalité nécessite 4 réécritures + 1 tactique (`Ring`). Il est ainsi clair que ce genre d'égalité doit être résolue directement par une tactique, à la fois pour un gain de temps considérable dans les développements, mais aussi de concision dans les scripts où l'on a plutôt envie de rendre implicites de telles preuves.

[4] On appelle preuves ε, les preuves utilisant la définition explicite de la limite sous la forme : $\forall \varepsilon > 0.\exists \alpha \ldots$. Dans la littérature anglaise, ces preuves sont plutôt connues sous la dénomination de preuves ε/δ.

[5] Il ne s'agit pas ici d'entamer le leitmotiv bien connu que les preuves formelles sont bien plus difficiles que les preuves que l'on peut trouver dans les meilleurs ouvrages de mathématiques mais de mettre en évidence une lacune d'automatisation qui, si elle venait à être comblée, pourrait permettre certaines "imprécisions" dans le sens où l'utilisateur n'aurait plus à montrer, à la main, certaines propriétés considérées comme triviales.

8.2 Algorithme

8.2.1 Principe

L'idée de l'algorithme est de minimiser les opérations de simplification de manière à se ramener le plus tôt possible à l'utilisation de la procédure de décision sur les anneaux abéliens (`Ring`). Cela signifie qu'il faut éliminer tous les inverses intervenant dans l'égalité qu'il s'agit de résoudre.

Pour ce faire, considérons les étapes suivantes :

1. Transformer les expressions $x - y$ en $x + (-y)$ et x/y en $x * 1/y$.
2. Chercher tous les inverses apparaissant dans l'égalité pour en faire un produit.
3. Distribuer totalement à gauche et à droite de l'égalité, excepté dans les inverses.
4. Associer à droite chaque monôme, excepté dans les inverses[6].
5. Multiplier à gauche et à droite par le produit d'inverses, que l'on a construit précédemment, en générant la condition que tous les inverses doivent être non nuls.
6. Distribuer seulement le produit sur la somme de monômes à gauche et à droite sans réassocier à droite.
7. Éliminer les inverses des monômes en utilisant la règle de corps $x.1/x = 1,\;$ si $x \neq 0$ et en permutant les éléments du monôme si nécessaire, c'est-à-dire s'il reste des inverses et que la règle de corps ne peut pas s'appliquer[7].
8. Recommencer le processus à partir de l'étape 2, s'il reste encore des inverses.

La dernière étape, qui consiste à réitérer le processus, s'explique par le fait qu'il peut y avoir d'autres inverses dans les inverses et ce, dans des expressions pouvant être compliquées. Pour éviter la réitération, une idée serait de se lancer dans une simplification directe en utilisant la règle $1/1/x = x,\;$ si $x \neq 0$. Toutefois, l'expérience a montré que le codage de cette simplification était plutôt complexe et générait un lemme de correction difficile. Par ailleurs, les expressions devant être différentes de 0 n'étaient pas exactement les mêmes que celles nécessaires pour l'élimination des inverses dans les monômes. On pouvait certes les déduire mais le lemme de correction correspondant à l'élimination des inverses devenait alors plus compliqué à montrer. On a donc tout avantage à se limiter à la règle $x.1/x = 1,\;$ si $x \neq 0$ qui tend à simplifier grandement l'algorithme et ce pour une perte d'efficacité négligeable en pratique[8].

Après ces étapes, nous obtenons une expression débarrassée de tous ses inverses et il suffit d'appeler `Ring` pour conclure.

8.2.2 Exemple

Afin de voir comment la procédure fonctionne, nous allons donner un petit exemple en détaillant les étapes de preuve. Étant donné x et y, deux variables d'un corps commutatif, on se propose de montrer l'égalité suivante :

$$x * (\frac{1}{x} + \frac{x}{x+y}) = (-\frac{1}{y}) * y * (-(\frac{x*x}{x+y}) - 1)$$

[6] Cette étape est clairement non nécessaire mais elle permet un gain d'efficacité en évitant un double appel récursif pour toutes les fonctions manipulant ces expressions.

[7] Il n'est pas nécessaire ici de vérifier que $x \neq 0$ car la condition a déjà été générée lors de la multiplication par le produit de tous les inverses.

[8] On estime, en effet, que les expressions contenant des empilements d'inverses d'inverses seront plutôt rares.

On commence par transformer les moins binaires et les divisions :

$$x * (\frac{1}{x} + x * \frac{1}{x+y}) = (-\frac{1}{y}) * y * (-(x*x) * \frac{1}{x+y} + (-1))$$

On construit le produit d'inverses que l'on appellera p :

$$p = x * ((x+y) * (y * (x+y)))$$

On distribue totalement à gauche et à droite sauf dans les inverses :

$$x * \frac{1}{x} + x * (x * \frac{1}{x+y}) = (-1) * \frac{1}{y} * y * ((-1) * (x*x) * \frac{1}{x+y}) + (-1) * \frac{1}{y} * y * (-1)$$

On associe à droite chaque monôme sauf dans les inverses :

$$x * \frac{1}{x} + x * (x * \frac{1}{x+y}) =$$

$$(-1) * (\frac{1}{y} * (y * ((-1) * (x * (x * \frac{1}{x+y}))))) + (-1) * (\frac{1}{y} * (y * (-1)))$$

On multiplie à gauche et à droite par p en générant la condition de correction :

$$(x * ((x+y) * (y * (x+y)))) * (x * \frac{1}{x} + x * (x * \frac{1}{x+y})) =$$

$$(x * ((x+y) * (y * (x+y)))) * ((-1) * (\frac{1}{y} * (y * ((-1) * (x * (x * \frac{1}{x+y})))))+$$

$$(-1) * (\frac{1}{y} * (y * (-1))))$$

Avec $x * ((x+y) * (y * (x+y))) \neq 0$.
On distribue ce produit sur les monômes sans réassocier à droite :

$$(x * ((x+y) * (y * (x+y)))) * (x * \frac{1}{x})+$$

$$(x * ((x+y) * (y * (x+y)))) * (x * (x * \frac{1}{x+y})) =$$

$$(x * ((x+y) * (y * (x+y)))) * (((-1) * (\frac{1}{y} * (y * ((-1) * (x * (x * \frac{1}{x+y}))))))+$$

$$(x * ((x+y) * (y * (x+y)))) * ((-1) * (\frac{1}{y} * (y * (-1))))$$

On élimine les inverses en permutant si nécessaire :

$$((x+y) * (y * ((x+y)))) * x + (x * (y * (x+y))) * (x * x) =$$

$$(x * (x+y)) * ((-1) * (y * ((-1) * (x * x))))+$$

$$(x * ((x+y) * (x+y))) * ((-1) * (y * (-1)))$$

On obtient alors une égalité sur les anneaux abéliens que `Ring` sait résoudre.

8.2.3 Remarques

La preuve que cet algorithme décide bien des égalités sur les corps commutatifs modulo certaines preuves d'inégalités n'a pas été formalisée. Il semble clair, cependant, qu'il est correct dans la mesure où l'on n'utilise que des axiomes de corps. On peut également se convaincre de la terminaison de la procédure puisque le nombre d'inverses décroît strictement à chaque itération.

Par ailleurs, il est important de souligner que la démarche ne vise pas à résoudre le problème global de décision sur les corps commutatifs dont on ne sait pas, *a priori*, s'il est décidable ou non. En effet, on ne cherche pas à prouver les conditions sur les inverses qui sont laissées à l'utilisateur. Ainsi, on se place dans une optique où l'inverse est une fonction totale.

Dans un souci de généralité, cette méthode a pour vocation de traiter tous les corps commutatifs. Si l'objectif avait été seulement d'automatiser les preuves utilisant les nombres réels, l'approche aurait été bien différente et on aurait certainement opté pour des algorithmes résolvant au premier ordre tels que, entre autres, la méthode de Tarski[9] [83], l'algorithme de Kreisel-Krivine[10] [49] ou la décomposition cylindrique de Collins [16].

Toujours dans cette optique plus générale, on peut citer le travail du projet Fundamental Theorem of Algebra (Herman Geuvers, Freek Wiedijk, Jan Zwanenburg, Randy Pollack et Henk Barendregt), avec, dans le cadre d'une axiomatisation constructive des nombres réels en Coq, le codage de la tactique réflexive `Rational` [33], traitant des égalités similaires à celles que nous nous proposons de résoudre. L'approche présentée ici se démarque essentiellement du fait de choix différents dans la formalisation des nombres réels. Tout d'abord, le fait que la fonction inverse soit totale permet de faire une réflexion totale des expressions de $\mathbb{R}$, ce qui n'est pas le cas dans `Rational`, où tout inverse contient aussi la preuve que le dénominateur est non nul. La réflexion doit donc aussi être partielle, ce qui rend le processus plus complexe. Enfin, nous considérons l'égalité de Leibniz, ce qui permet à l'utilisateur d'appliquer des tactiques de réécriture à n'importe quel prédicat alors que dans `Rational`, l'égalité est plus large (setoïde) et il est nécessaire de prouver des lemmes de compatibilité afin de passer au contexte.

8.3 Implantation

Comme nous l'avons dit précédemment, l'implantation de cette procédure de décision sur les corps commutatifs, que nous avons appelée `Field`, a été réalisée dans la version V7 de Coq, afin de pouvoir profiter des nouvelles possibilités du langage de tactiques $\mathcal{L}_{tac}$. Le développement a été intégré au système et fait partie de la distribution. Le lecteur intéressé pourra consulter les sources du système[11] et, plus particulièrement, le répertoire suivant :

~/V7/contrib/field/

Comme nous l'avons également vu, la tactique a été développée de manière réflexive et, avant d'entrer dans les détails de l'implantation, nous allons rappeler brièvement en quoi consiste un codage par réflexion.

[9] Cet algorithme ne fonctionne qu'en logique classique [31] mais ce n'est pas gênant dans la mesure où il en est de même pour les nombres réels en Coq à cause de l'axiome d'ordre total.

[10] Kreisel et Krivine se sont aussi intéressés à un algorithme dans d'autres structures telles que les corps algébriquement clos, les anneaux de Boole séparables, ...

[11] Les sources du système sont disponibles à l'adresse : http ://coq.inria.fr/.

8.3.1 À propos de la réflexion

Pour coder `Field`, il y a globalement deux choix possibles. Un codage explicite en utilisant la réécriture ou un codage par réflexion en utilisant la réduction. Le codage explicite (approche à la Edinburgh LCF) est très coûteux du fait de l'utilisation de la réécriture qui prend du temps mais aussi et surtout de la place dans le terme preuve[12]. Le codage par réflexion est une alternative complètement satisfaisante par rapport à ces deux critères. En effet, les réécritures sont remplacées par des phases de réduction plus efficaces et la taille du terme preuve est de l'ordre de celle du but à résoudre. Par ailleurs, on peut formaliser clairement la correction globale de la tactique ainsi que sa complétude (même si nous ne l'avons pas fait ici) alors que dans l'approche explicite, c'est bien plus difficile, voire impossible.

Le principe de la réflexion est le suivant : soit un langage C des termes concrets (typiquement un type quelconque) et un langage A des termes abstraits (typiquement un type inductif). Comme on ne peut pas manipuler les termes du langage C comme on voudrait (on ne peut pas filtrer), l'idée est de le réfléchir dans le langage A qui lui est isomorphe. Une première phase, appelée métaification [13] par Samuel Boutin [11], consiste donc à traduire les termes de C vers les termes de A. Plus précisément, cela consiste, pour un terme c de C, à trouver le terme a de A tel que $(f\ v\ a) = c$, où f est la fonction d'interprétation de A vers C (codable dans Coq), v est une liste d'associations contenant les parties de C que l'on ne réfléchit pas (atomes) et $=$ est l'égalité de Leibniz. On peut se passer de la liste d'associations à condition que l'égalité sur les atomes soit décidable car on a généralement besoin de comparer les termes de A. La métaification s'assimile exactement à une phase d'analyse syntaxique comme on pourrait la trouver dans un langage de programmation.

Ensuite, on peut coder la fonction t de transformation des termes de A. Pour l'utiliser, il suffit de prouver un lemme de correction[14] de la forme :

$$\forall a \in A.(f\ v\ (t\ a)) = (f\ v\ a)$$

Enfin, après avoir appliqué ce lemme de correction, il suffit de réduire totalement (`Compute`) pour transformer le terme abstrait (de A) et revenir à un terme concret (de C). Pour pousser l'analogie avec les langages de programmation, on pourrait voir ces deux étapes comme une phase d'évaluation suivie d'une phase de *pretty-print*.

On peut résumer la situation au moyen du schéma de la figure 8.1. Pour une description complète de la réflexion, on pourra se reporter à [11] et [38].

8.3.2 Codage de la tactique

Nous allons maintenant entrer dans les détails de l'implantation de l'algorithme donné précédemment.

Paramétrisation

La tactique `Field` doit pouvoir fonctionner pour tout corps commutatif. Elle est donc paramétrée par une théorie de corps commutatif, c'est-à-dire un ensemble et des axiomes sur cet ensemble lui donnant une structure de corps commutatif.

[12] La taille du terme preuve est un point auquel il faut être très sensible car il n'est pas rare de rencontrer des scripts de preuves corrects pour lesquels on ne peut pas construire le terme preuve, faute de mémoire suffisante.

[13] Dans la littérature, cette phase est aussi appelée quotation ou réification.

[14] Il est intéressant de voir ici que dans le processus de réflexion, la tactique et la preuve de sa correction sont indissociables.

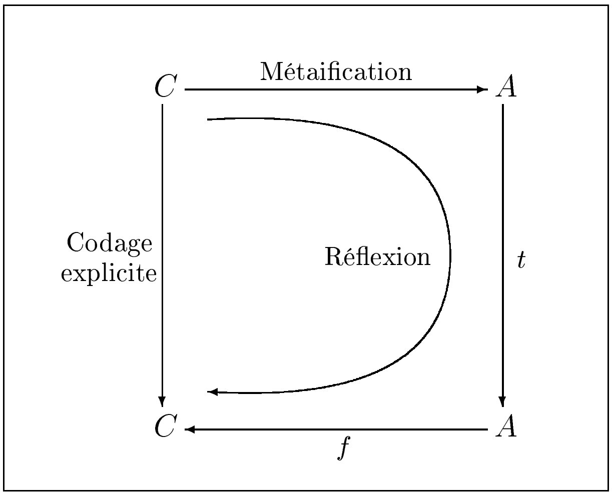


FIG. 8.1 – Principe de la réflexion.

Pour effectuer cette paramétrisation, nous avons choisi d'utiliser une structure d'enregistrement (`Record` en `Coq`), qui permet de contenir à la fois l'ensemble et ses axiomes. Nous définissons ainsi un type corps commutatif, appelé `Field_Theory`, de la manière suivante :

```
1   Record Field_Theory : Type :=
2   { A : Type;
3     Aplus : A -> A -> A;
4     Amult : A -> A -> A;
5     Aone : A;
6     Azero : A;
7     Aopp : A -> A;
8     Aeq : A -> A -> bool;
9     Ainv : A -> A;
10    Aminus : (option A);
11    Adiv : (option A);
12    RT : (Ring_Theory Aplus Amult Aone Azero Aopp Aeq);
13    Th_inv_def : (n:A)~(n==Azero)->(Amult (Ainv n) n)==Aone
14  }.
```

En ligne 1, on a l'ensemble support et de la ligne 3 à 11, ce sont ses opérateurs. En lignes 10 et 11, les deux champs sont optionnels car ils correspondent aux constantes représentant le moins binaire et la division, qui s'expriment respectivement en fonction du plus et de l'opposé et en fonction de la multiplication et de l'inverse. En ligne 12, on munit l'ensemble d'une structure d'anneau au moyen de `Ring_Theory`[15], qui lui aussi est un enregistrement et qui contient, entre autres, tous les axiomes d'anneau. Enfin, en ligne 13, on définit l'axiome de simplification de l'inverse, qui permet de passer d'une structure d'anneau à une structure de corps.

Remarque 8.3.1 (Choix de la sorte pour l'ensemble support) *Nous avons choisi de mettre l'ensemble support dans* `Type` *de manière à être le plus général possible. On permet,*

[15]`Ring_Theory` fait partie de la formalisation de la tactique `Ring` et se trouve dans le fichier Ring_theory.v dans le répertoire ~/V7/contrib/ring/.

par exemple, l'instantiation avec les nombres réels, qui, par souci de cohérence[16]*, sont dans* Type*. Mais, grâce à la cumulativité, on peut aussi traiter des corps commutatifs dans* Set *ou même* Prop *dans la mesure où ces deux sortes sont aussi considérées comme des univers, c'est-à-dire des* Type*. Pour plus de détails sur cette spécificité sur le système de types de* Coq*, se référer à [84] et [90].*

Structure de réflexion

Dans le cas de `Field`, les termes concrets sont exactement les expressions du corps commutatif (par exemple, des expressions de type `R`, dans le cas des nombres réels). Les termes abstraits doivent *refléter* les opérateurs qui permettent de donner une structure de corps commutatif et se définissent très simplement par un type inductif que nous avons appelé `ExprA` :

```
Inductive ExprA : Set :=
| EAzero : ExprA
| EAone  : ExprA
| EAplus : ExprA -> ExprA -> ExprA
| EAmult : ExprA -> ExprA -> ExprA
| EAopp  : ExprA -> ExprA
| EAinv  : ExprA -> ExprA
| EAvar  : nat -> ExprA.
```

Les variables sont des expressions du corps commutatif quelconques pour lesquelles on ne peut pas, de fait, décider de l'égalité de manière générale. On les remplace donc par des indices entiers (`nat`) pour pouvoir décider l'égalité entre variables et on associe à un terme une liste d'associations entre les indices et des expressions du corps commutatif.

Métaification

Pour traduire les expressions du corps commutatif vers `ExprA` (métaification), il faut utiliser le métalangage de Coq. Jusqu'à la V7 exclue, le seul moyen était de coder cette traduction dans Objective Caml (le langage d'implantation et le métalangage de Coq, [54]) en utilisant un fichier ML que l'on compilait avec le système et que l'on pouvait importer dans un toplevel bytecode de Coq. Ce protocole était un peu lourd à mettre en œuvre[17], d'autant que le processus de métaification est extrêmement simple. Dans la V7, $\mathcal{L}_{tac}$ permet de se libérer de ce genre de contraintes, notamment, dans ce cas, au moyen du filtrage sur les termes de Coq. Par ailleurs, un autre atout intéressant est, qu'étant intégré au toplevel de Coq, il est possible de faire tourner le code au moyen d'un toplevel compilé en natif sans perdre la modularité[18] du sytème.

[16]En effet, on aurait plus tendance à mettre les nombres réels dans `Set`. Toutefois, cette sorte est purement intuitionniste et le modèle de `Set` est la *réalisibilité*. En particulier, le tiers exclu ne peut pas être réalisé dans `Set`, ce qui explique qu'il ne faut pas l'intégrer à une quelconque théorie sous peine de la rendre incohérente. Dans l'axiomatisation des nombres réels, il y a l'axiome d'ordre total, qui, du fait que l'on soit dans `Set`, impose la décidabilité de l'égalité. Hors, on ne peut pas, *a priori*, trouver de modèle réalisant cet axiome, c'est-à-dire où l'égalité est décidable sur les nombres réels. Cet axiome s'exprime très facilement en fonction du tiers exclu (qui, lui non plus, n'est pas réalisable) et peut alors se qualifier de classique, ce qui est incompatible avec `Set`. Ceci explique pourquoi l'axiomatisation des nombres réels a été *montée* dans `Type`. Pour plus de détails sur le sujet, on pourra consulter [59].

[17]En effet, il faut se procurer Objective Caml, Camlp4, compiler les sources de Coq, coder la métaification en comprenant la structure abstraite des termes Coq et enfin compiler le fichier en question.

[18]En effet, le chargement dynamique de fichiers bytecode dans un exécutable natif n'est pas encore très standard et le chargement dynamique de fichiers natifs dans du natif n'est, quant à lui, clairement pas com-

Dans `Field`, pour métaifier, on utilise d'abord une tactique appelée `BuildVarList`, qui construit la liste d'associations des expressions du corps commutatif non reflétés avec leurs indices entiers, telle que deux expressions égales (au sens de l'égalité de Leibniz) aient le même indice. Cette liste est ensuite passée à la fonction d'interprétation du corps commutatif dans `ExprA`, définie de la manière suivante :

```
1  Recursive Tactic Definition interp_A FT lvar trm :=
2    Let AT     = Eval Compute in (A FT)
3    And AzeroT = Eval Compute in (Azero FT)
4    And AoneT  = Eval Compute in (Aone FT)
5    And AplusT = Eval Compute in (Aplus FT)
6    And AmultT = Eval Compute in (Amult FT)
7    And AoppT  = Eval Compute in (Aopp FT)
8    And AinvT  = Eval Compute in (Ainv FT) In
9    Match trm With
10   | [(AzeroT)] -> EAzero
11   | [(AoneT)] -> EAone
12   | [(AplusT ?1 ?2)] ->
13     Let e1 = (interp_A FT lvar ?1)
14     And e2 = (interp_A FT lvar ?2) In
15     '(EAplus e1 e2)
16   | [(AmultT ?1 ?2)] ->
17     Let e1 = (interp_A FT lvar ?1)
18     And e2 = (interp_A FT lvar ?2) In
19     '(EAmult e1 e2)
20   | [(AoppT ?1)] ->
21     Let e = (interp_A FT lvar ?1) In
22     '(EAopp e)
23   | [(AinvT ?1)] ->
24     Let e = (interp_A FT lvar ?1) In
25     '(EAinv e)
26   | [?1] ->
27     Let idx = (Assoc ?1 lvar) In
28     '(EAvar idx).
```

où `Assoc` est une tactique qui donne l'indice entier correspondant à l'expression du corps commutatif dans la liste d'associations (`lvar`).

En ligne 1, le paramètre `FT` est un corps commutatif (de type `Field_Theory`). De la ligne 2 à la ligne 8, on extrait, de `FT`, l'ensemble ainsi que ses opérateurs[19], excepté le moins binaire et la division (champs `Aminus` et `Adiv`). En effet, on suppose, à ce niveau-là, que les éventuelles constantes correspondant au moins binaire et à la division ont déjà été expansées (dans une passe préalable). Enfin, des lignes 9 à 28, on reproduit, dans `ExprA`, la structure

---

pris. La commande `coqmktop` permet de créer un toplevel "customisé" éventuellement en natif en indiquant une liste de fichiers à inclure au moment de l'édition de liens. Toutefois, le processus est purement statique et on perd la possibilité d'appeler d'autres tactiques qui n'ont pas été "linkées" au moment du `coqmktop`.

[19] En `Coq`, les types enregistrements ne sont pas primitifs et utilisent les types inductifs. Un type enregistrement déclare ainsi un type inductif à un seul constructeur contenant tous les champs de l'enregistrement. De plus, les labels des champs sont déclarés comme des constantes fonctionnelles prenant en argument un objet du type inductif déclaré et qui le déstructurent pour obtenir les contenus des champs correspondants. Ainsi, accéder à un champ n'est pas immédiat et doit passer par une phase de réduction (d'où la série de `Eval Compute in` sur `FT`).

de l'expression du corps commutatif et, pour le reste (cas variable, lignes 26 à 28), il suffit de rechercher l'indice associé à l'expression dans la liste produite par `BuildVarList`.

Construction du multiplicateur

Pour simplifier, il nous faut, au préalable, construire un produit de facteurs sans doublons inutiles (doublons qui apparaîtront entre monômes après distribution[20]), constitué des inverses de l'égalité (on ne prend pas en compte les inverses dans les inverses qui seront traités dans d'autres passes de `Field`). Pour ce faire, on a le choix entre utiliser une fonction de `Coq` ou une tactique que l'on peut plus facilement écrire dans la V7. Pour des raisons d'aisance de programmation, nous avons opté pour une tactique puisque l'on est moins limité, entre autres, dans la récursivité. Le multiplicateur nous est donc donné par la tactique `GiveMult`, utilisant la tactique `RawGiveMult`, qui donne la liste des inverses :

```
1   Recursive Tactic Definition RawGiveMult trm :=
2     Match trm With
3     | [(EAinv ?1)] -> '(cons ExprA ?1 (nil ExprA))
4     | [(EAopp ?1)] -> (RawGiveMult ?1)
5     | [(EAplus ?1 ?2)] ->
6       Let l1 = (RawGiveMult ?1)
7       And l2 = (RawGiveMult ?2) In
8       (Union l1 l2)
9     | [(EAmult ?1 ?2)] ->
10      Let l1 = (RawGiveMult ?1)
11      And l2 = (RawGiveMult ?2) In
12      Eval Compute in (app ExprA l1 l2)
13    | _ -> '(nil ExprA).
14
15  Tactic Definition GiveMult trm :=
16    Let ltrm = (RawGiveMult trm) In
17    '(mult_of_list ltrm).
```

où `nil`, `cons` et `app` sont les constructeurs et la concaténation des listes polymorphes. En ligne 8, `Union` est une tactique qui concatène deux listes en éliminant les doublons (éléments communs aux deux listes). On remarquera, à ce propos, que l'on élimine bien que les doublons inutiles dans la mesure où `Union` n'est utilisée que pour supprimer les doublons provenant de deux monômes, c'est-à-dire, d'expressions séparées par l'addition (opérateur `EAplus`). En ligne 17, `mult_of_list` est une fonction `Coq` qui rend un produit associé à droite à partir d'une liste d'expressions de `ExprA`. Le `Eval Compute in` de la ligne 12 permet de réduire les `app` d'une liste pour obtenir une liste canonique (constituée uniquement de `nil` et `cons`) pouvant être correctement filtrée (par `mult_of_list`).

Distributivité et associativité

Il s'agit maintenant de distribuer totalement (sauf dans les inverses) et d'associer à droite (par rapport à l'addition et à la multiplication) dans les membres de l'égalité. Ces deux fonctions se font obligatoirement dans `Coq` car l'idée est de passer du terme initial au

[20] En effet, il peut y avoir des doublons utiles au sens où ils sont nécessaires pour simplifier un monôme. Par exemple, pour le monôme $\frac{1}{x^2}$, le produit est x^2 et le doublon de x est nécessaire pour simplifier ensuite le monôme. Par contre, dans l'expression $\frac{1}{x}+\frac{1}{x}$, le produit peut être x^2, mais le doublon de x est ici inutile et on peut se contenter de prendre x comme produit pour simplifier.

terme transformé via la réduction de Coq et un lemme de correction à prouver pour chaque fonction.

La distributivité ne peut pas être codée directement et facilement dans Coq. En effet, les conditions de garde assurant la normalisation forte obligent à découper le problème de manière à faire des appels récursifs respectant la décroissance de la mesure (ordre sous-terme). Les fonctions peuvent donc sembler un peu compliquées mais il s'agit surtout de respecter ces conditions syntaxiques. Pour distribuer, l'idée est de distribuer d'abord tous les moins unaires `EAopp`. Ensuite, on distribue totalement dans les sous-termes et, pour le cas de la multiplication `EAmult`, on distribue d'abord à gauche puis à droite. Nous ne donnerons pas ici les fonctions en question qui ne présentent pas un intérêt particulier. Le lecteur intéressé pourra se reporter au code source. Pour utiliser la fonction de distributivité `distrib`, on a prouvé le lemme de correction suivant :

```
Lemma distrib_correct:
  (T:Field_Theory; e:ExprA; lvar:(list (Sprod (A T) nat)))
    (interp_ExprA T lvar (distrib e))==(interp_ExprA T lvar e).
```

où `T` est le corps commutatif initial (avec ses axiomes), `interp_ExprA` la fonction d'interprétation de `ExprA` vers le corps commutatif (écrite en Coq), `lvar` la liste d'associations des variables et `Sprod` le produit cartésien entre un terme dans `Type` et un terme dans `Set`.

L'associativité ne pose pas de problèmes dans son codage moyennant quelques précautions. De même que pour `distrib`, nous ne donnerons pas le code de la fonction d'associativité, nommée `assoc`, et il nous a fallu prouver le lemme de correction suivant :

```
Lemma assoc_correct:
  (T:Field_Theory; e:ExprA; lvar:(list (Sprod (A T) nat)))
    (interp_ExprA T lvar (assoc e))==(interp_ExprA T lvar e)
```

Après avoir appliqué `distrib` et `assoc`, on obtient à gauche et à droite de l'égalité à prouver, deux termes qui sont des sommes de monômes associés à droite.

Multiplier les membres de l'égalité

Pour pouvoir effectuer la simplification des inverses, on multiplie ensuite par le multiplicateur qui a été construit précédemment (produit de tous les inverses excepté ceux qui sont dans d'autres inverses). Pour ce faire, il suffit de montrer le lemme suivant sur les expressions de `ExprA`, qui a directement son équivalent dans le corps commutatif reflété :

```
Lemma mult_eq:
  (T:Field_Theory; e1,e2,a:ExprA; lvar:(list (Sprod (A T) nat)))
    ~(interp_ExprA T lvar a)==(Azero T)->
      (interp_ExprA T lvar (EAmult a e1))==
      (interp_ExprA T lvar (EAmult a e2))->
        (interp_ExprA T lvar e1)==(interp_ExprA T lvar e2).
```

Ce lemme permet de générer le but (que l'utilisateur devra prouver) que le multiplicateur doit être non nul. Ceci équivaut à dire que tous ses facteurs doivent être nuls ce qui est cohérent dans la mesure où l'on est sûr de devoir simplifier ces expressions.

Une fois le produit effectué, on distribue ce produit sur les monômes sans réassocier à droite, ce qui est trivialement fait par une fonction Coq appelée `multiply` dont le lemme de correction est complètement similaire à ceux donnés précédemment pour la distributivité et l'associativité.

Élimination des inverses

L'élimination des inverses se fait monôme par monôme. À ce stade, un monôme est un produit du multiplicateur et d'une expression qui est un produit associé à droite (monôme obtenu après la phase de distributivité et d'associativité). L'idée est donc de parcourir le produit du multiplicateur et de repérer les inverses correspondants dans le reste du monôme pour les simplifier. Pour assurer la correction de la simplification, on va paramétrer par un produit de facteurs (supposé non nul) et, avant de simplifier, on vérifiera que ce produit est égal au multiplicateur.

Concrètement, le travail est effectué par les fonctions Coq suivantes :

```
1   Definition monom_simplif [a,m:ExprA] : ExprA :=
2     Cases m of
3     | (EAmult a' m') =>
4       (Cases (eqExprA a a') of
5        | (left _) => (monom_simplif_rem a m')
6        | (right _) => m
7        end)
8     | _ => m
9     end.
10
11  Fixpoint inverse_simplif [a,e:ExprA] : ExprA :=
12    Cases e of
13    | (EAplus e1 e2) =>
14      (EAplus (monom_simplif a e1) (inverse_simplif a e2))
15    | _ => (monom_simplif a e)
16    end.
```

où monom_simplif_rem, en ligne 5, prend deux produits de facteurs en arguments et parcourt le premier pour supprimer d'éventuels inverses dans le second.

On remarque que, comme prévu, inverse_simplif, en ligne 11, simplifie monôme par monôme et que monom_simplif, en ligne 1, vérifie bien que le premier produit d'un monôme correspond au multiplicateur (passé en paramètre).

Le lemme de correction d'élimination des inverses s'exprime comme suit :

```
1   Lemma inverse_correct:
2     (T:Field_Theory; e,a:ExprA; lvar:(list (Sprod (A T) nat)))
3       ~(interp_ExprA T lvar a)==(Azero T)->
4         (interp_ExprA T lvar (inverse_simplif a e))==
5         (interp_ExprA T lvar e).
```

Ce lemme est paramétré par un produit de facteurs (paramètre a) que l'on suppose non nul, en ligne 3, de manière à ce que les éventuelles simplifications réalisées par inverse_simplif soient correctes.

La tactique globale

La tactique Field combine toutes les phases que nous venons de voir. Elle s'exprime directement dans le langage de tactiques de la V7 comme suit :

```
Tactic Definition Field_Gen FT :=
  Let AplusT = Eval Compute in (Aplus FT) In
```

```
3     Unfolds FT;
4     Match Context With
5     | [|- ?1==?2] ->
6       Let lvar = (BuildVarList FT '(AplusT ?1 ?2)) In
7       Let trm1 = (interp_A FT lvar ?1)
8       And trm2 = (interp_A FT lvar ?2) In
9       Let mul = (GiveMult '(EAplus trm1 trm2)) In
10      Cut [ft:=FT][vm:=lvar]
11          (interp_ExprA ft vm trm1)==(interp_ExprA ft vm trm2);
12      [Compute;Auto
13      |Intros;(ApplySimplif ApplyDistrib);(ApplySimplif ApplyAssoc);
14       (Multiply mul);[(ApplySimplif ApplyMultiply);
15         (ApplySimplif (ApplyInverse mul));
16         (Let id = GrepMult In Clear id);Compute;
17         First [(InverseTest FT);Ring|(Field_Gen FT)]|Idtac]].
```

En ligne 2, la tactique `Unfolds` permet de déplier les moins binaires et les divisions, s'ils ont été fournis dans le corps commutatif `FT`. Ensuite, en ligne 6, on crée la liste d'associations des variables (dans `lvar`) avec `BuildVarList` (en sommant les deux membres de l'égalité de manière à tenir compte des variables des deux termes), pour interpréter, en lignes 7 et 8, les deux membres de l'égalité, ce qui donne deux termes `trm1` et `trm2` de `ExprA`. En ligne 9, le multiplicateur `mul` est donné par `GiveMult` (en prenant soin, comme avec `BuildVarList`, de sommer les deux termes pour tenir compte des inverses des deux membres de l'égalité). Le `Cut`[21] des lignes 10 et 11 permet alors d'obtenir un but faisant intervenir les termes de `ExprA`. Par la suite, on peut leur appliquer les différentes transformations dont nous avons parlé précédemment au moyen de tactiques, toujours définies à toplevel. `ApplyDistrib`, en ligne 13, applique la distributivité, `ApplyAssoc`, en ligne 13, l'associativité, `Multiply`, en ligne 14, la multiplication par le multiplicateur à gauche et à droite de l'égalité, `ApplyMultiply`, en ligne 14, la distribution du multiplicateur et `ApplyInverse`, en ligne 15, l'élimination des inverses. `ApplySimplif` permet d'appliquer la tactique à gauche et à droite de l'égalité pour les tactiques qui travaillent sur un terme. On se débarrasse, en ligne 16, de l'hypothèse que l'interprétation du multiplicateur doit être non nulle (on n'en a plus besoin puisqu'à ce stade, l'élimination des inverses a déjà été faite) au moyen de `GrepMult`, qui rend le nom de cette hypothèse pouvant être ainsi effacées (`Clear`). Enfin, en ligne 17, on teste s'il reste encore des inverses dans les termes de l'égalité grâce à la tactique `InverseTest` qui, soit ne fait rien s'il ne reste pas d'inverses permettant ainsi l'appel à `Ring`, soit échoue dans le cas contraire impliquant une nouvelle application de `Field`.

Instantiation de la tactique

Pour utiliser `Field` avec une théorie donnée, disons `FT_C`, il suffit d'appeler la tactique précédente comme suit :

```
Field_Gen FT_C.
```

Pour aller plus vite, l'utilisateur peut même définir la tactique suivante :

[21]Les deux `let-in` (avec `ft` et `vm`) dans l'expression introduite par la coupure permettent de réduire considérablement la taille du terme preuve. En effet, étant de suite introduits dans le contexte local, tout le reste de la preuve est factorisée par les deux expressions correspondantes, d'où un gain de place important sachant que ces expressions sont certes de taille moyenne mais avec de nombreuses occurences.

```
Tactic Definition FieldC := Field_Gen FT_C.
```

et appeler directement `FieldC` par la suite.

Toutefois, ce n'est pas très flexible et il pourrait y avoir d'éventuelles collisions dans les noms choisis pour appeler la tactique (tout le monde souhaitant probablement utiliser le nom `Field`). C'est pourquoi nous avons opté pour un système similaire à la tactique `Ring` (on gagne, de surcroît, en homogénéité). Ainsi, on utilise une table (de hash) stockant les différentes théories avec, comme clé, l'ensemble support. Il n'y a alors plus qu'une seule tactique, appelée `Field`, qui, de l'égalité à résoudre, devine l'ensemble support, puis récupère la théorie correspondante dans la table pour enfin utiliser `Field_Gen` avec. Cependant, tout ceci ne peut pas se faire à toplevel et ce, essentiellement à cause de la table, qui est un trait éminemment impératif, de fait, non disponible dans le langage de Coq. On est donc contraint, pour cette partie, de *descendre* au niveau d'Objective Caml, mais le code en question est plutôt petit (une centaine de lignes), ce qui le rend finalement assez peu significatif sur le développement global (d'autant qu'il s'agit de fournir uniquement une facilité supplémentaire).

Ainsi, l'instantiation se fait par la commande toplevel suivante :

```
Add Field A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl.
```

où `A` est l'ensemble support, `Aplus`, `Amult`, `Aone`, `Azero`, `Aopp`, `Ainv`, sont ses opérateurs, `Aeq` est une égalité de Leibniz semi-décidable à valeurs dans `bool`[22] utilisée par `Ring`, `Rth` contient les axiomes d'anneau de `A` (voir le type `Ring_Theory`) et `Tinvl` est l'axiome de corps sur l'inverse (de type `(n :A)~(n==Azero)->(Amult (Ainv n) n)==Aone`).

En option, on peut éventuellement préciser le moins binaire ou la division comme suit :

```
Add Field A Aplus Amult Aone Azero Aopp Aeq Ainv Rth Tinvl
          with minus:=Aminus div:=Adiv.
```

où `Aminus` et `Adiv` sont les constantes représentant respectivement le moins binaire et la division.

Cette commande a pour effet de construire la théorie du corps commutatif correspondante (de type `Field_Theory`) et de la stocker dans une table avec `A` comme clé. S'il n'y a pas de théorie d'anneau déclarée (commande `Add Ring`), `Add Field` la déclare, ce qui permet d'utiliser `Ring` ensuite et surtout d'économiser une ligne de commande[23]. Il suffit alors d'appeler simplement `Field`, qui, comme on l'a dit précédemment, se charge de récupérer, à partir de l'égalité à résoudre, le `A` correspondant et peut ensuite retrouver la théorie liée à cette clé pour appeler `Field_Gen`.

Remarque 8.3.2 (Appel à `Field_Gen`) *La commande* `Add Field` *n'est pas incompatible avec l'appel direct à la tactique* `Field_Gen`*, qui peut continuer à être utilisée normalement.*

Pour voir un exemple d'instantiation, on pourra se référer à celui des nombres réels, qui se situe dans le fichier Rbase.v (répertoire ~/V7/theories/Reals/).

[22] Si on a `(Aeq x y)=true` alors on a `x=y`, mais si on a `(Aeq x y)=false`, on peut quand même avoir `x=y`.

[23] En effet, un corps commutatif est aussi un anneau. Toutefois, on peut décider de garder quand même `Add Ring` si on utilise certaines spécificités de cette commande (voir [84]).

8.4 Exemples

Nous donnons ici quelques exemples, accompagnés du temps d'exécution. Ces exemples sont tirés de preuves faisant partie ou allant faire partie du développement[24] des nombres réels où l'utilisation de cette tactique sera la bienvenue (autant par commodité que dans le but d'alléger les preuves).

Les tests sont effectués sur un Intel Pentium III à 600 MHz sous Linux Mandrake 7.2 et la version (V7) de Coq utilisée a été compilée en natif. En effet, comme nous l'avons dit précédemment, l'utilisation du langage de tactiques de Coq pour coder `Field` s'adapte parfaitement à la compilation native. L'appel à `Ring` ne pose aucun problème puisque le code ML correspondant est linké par défaut dans la version native.

8.4.1 Exemple 1

L'exemple que nous donnons ici appartient à une famille d'égalités que nous pouvons qualifier de simples. Néanmoins, ce type d'égalités revient assez souvent dans les preuves et l'accumulation de toutes ces petites preuves devient rapidement fastidieuse et fini par produire un terme preuve plus important qu'il ne devrait. Pour cette raison, il est intéressant de pouvoir utiliser `Field` fréquemment, tout comme `Ring` afin de minimiser le nombres de réécritures.

Nous considérons, par exemple, les parties de preuves présentes de manière récurrente telles que : $b = \frac{b}{a} * a$.

```
Welcome to Coq 7.1 (September 2001)

Coq < Goal (a,b:R)``b == b*(1/a)*a``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  a : R
  b : R
  ============================
   ``a <> 0``
Finished transaction in 0 secs (0.42u,0s)
```

Sans aucune réécriture, il ne nous reste plus qu'à prouver que $a \neq 0$, propriété que nous devions impérativement prouver même dans le cas où nous procédions par réécritures.

8.4.2 Exemple 2

Nous voulons prouver que $\frac{\varepsilon}{2+2} + \frac{\varepsilon}{2+2} = \frac{\varepsilon}{2}$. Cette opération est utilisée dans la preuve concernant la multiplication des limites (`limit_mul`). La preuve du but énoncé ci-dessous fait environ 25 lignes de Coq, alors qu'après l'application de `Field`, il nous reste à prouver uniquement que $2 + 2$ et 2 sont non nuls[25].

```
Coq < Goal (eps:R)``eps*1/(2+2)+eps*1/(2+2) == eps*1/2``.
```

[24] Plus précisément, deux fichiers sont principalement concernés : `Rlimit.v` et `Rderiv.v`.

[25] Ceci peut être fait au moyen de la tactique `DiscrR`, écrite aussi à toplevel avec $\mathcal{L}_{tac}$, qui permet de montrer de deux expressions à base de `R0`, `R1`, `Rplus`, `Rmult` et `Ropp`, sont distinctes.

```
Unnamed_thm < Intro. Time Field.
1 subgoal

  eps : R
  ============================
   ``(2+2)*2 <> 0``
Finished transaction in 1 secs (1.08u,0s)
```

Comme dit précédemment, pour pouvoir simplifier par $2+2$ et par 2, nous utilisons le fait que $2+2 \neq 0$ et $2 \neq 0$. Ces deux conditions sont générées sous la forme d'un unique sous-but traduisant ces deux propriétés : $(2+2)*2 \neq 0$. En effet, si un produit est non nul alors chacun de ses facteurs est également non nul.

8.4.3 Exemple 3

Revenons sur l'exemple cité en introduction, tiré de la preuve concernant l'addition des dérivées :

```
Coq < Goal (f,g:(R->R); x0,x1:R)
Coq <   ``((f x1)-(f x0))/(x1-x0)+((g x1)-(g x0))/(x1-x0) ==
Coq <     ((f x1)+(g x1)-((f x0)+(g x0)))/(x1-x0)``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  f : R->R
  g : R->R
  x0 : R
  x1 : R
  ============================
   ``x1+ -x0 <> 0``
Finished transaction in 2 secs (1.84u,0.02s)
```

Il ne reste plus qu'à prouver que $x1 - x0 \neq 0$, ce qui est une hypothèse du lemme d'addition des dérivées.

8.4.4 Exemple 4

Nous nous intéressons ici aux preuves concernant les séries entières, utilisées pour définir les fonctions transcendantes (en cours de développement) telles que l'exponentielle, sinus ou cosinus.

Considérons, par exemple, l'application du critère de d'Alembert à la fonction exponentielle. Rappelons, avant tout, les définitions d'une série entière, de la fonction exponentielle ainsi que l'énoncé du critère de d'Alembert :

Une série entière réelle est une série de la forme : $\sum_{n=0}^{+\infty} a_n.x^n$.

La fonction exponentielle e^x peut se définir ainsi : $\sum_{i=0}^{+\infty} \frac{x^i}{i!}$.

Une forme du critère de d'Alembert est : si $\left|\frac{a_{n+1}}{a_n}\right| \xrightarrow[n \to +\infty]{} 0$ alors $\exists l,\ \sum_{i=0}^{+\infty} a_n.x^n \ \to l$.

Une manière de définir l'exponentielle est donc d'appliquer le critère de d'Alembert avec $a_n = \frac{1}{n!}$ et il faut alors montrer que $\left|\frac{\frac{1}{(n+1)!}}{\frac{1}{n!}}\right| \xrightarrow[n \to +\infty]{} 0$.

Dans ce but, nous pouvons montrer l'égalité suivante et l'appliquer ultérieurement dans la preuve avec $a = (S\ n)$ et $b = n!$:

$$\frac{\frac{1}{a.b}}{\frac{1}{b}} = \frac{1}{a}$$

```
Coq < Goal (a,b:R)``a <> 0``->``b <> 0``->``1/(a*b)/(1/b) == 1/a``.

Unnamed_thm < Intros. Time Field.
2 subgoals

  a : R
  b : R
  H : ``a <> 0``
  H0 : ``b <> 0``
  ============================
   ``b <> 0``

subgoal 2 is:
 ``a*b*(1*/b*a) <> 0``
Finished transaction in 2 secs (1.29u,0s)
```

Nous remarquons la génération de deux nouveaux sous-buts. Conformément à l'algorithme utilisé, la première passe génère le sous-but 2 tandis que la seconde génère le sous-but 1. En effet, la première passe commence par multiplier par $a * b * \frac{1}{b} * a$ et, après simplifications, il reste alors un $\frac{1}{b}$. La seconde passe multiplie donc par b. Cela équivaut alors à montrer $a * b \neq 0$, $\frac{1}{b} \neq 0$, $a \neq 0$ et $b \neq 0$, ce qui se déduit des hypothèses.

8.4.5 Exemple 5

Enfin, nous pouvons donner l'exemple de la section 8.2.2, qui n'a pas de sens particulier mais qui est un bon test pour `Field` :

```
Coq < Goal (x,y:R)``x*(1/x+x/(x+y)) == -1/y*y*(-(x*x)/(x+y)-1)``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  x : R
  y : R
  ============================
   ``x*((x+y)*y) <> 0``
Finished transaction in 1 secs (1.46u,0s)
```

8.4.6 Observations

À la vue de ces quelques exemples, la principale observation concerne le temps d'exécution de la tactique. En effet, sur certains exemples, nous pouvons constater des performances assez moyennes[26].

Après quelques tests rapides, nous avons éliminé des sources potentielles d'inefficacité et isolé quelques causes probables. La perte de temps a essentiellement lieu au sein de fonctions Coq et n'est donc pas due au métalangage. Les fonctions chargées de distribuer tous les termes (exponentielle en le nombre de nœuds de l'expression distribuée) afin d'obtenir des monômes ainsi que la fonction de simplification des inverses semblent être les principales mises en cause. En particulier, l'algorithme de simplification des inverses pourrait être modifié en effectuant un tri préalable du multiplicateur et des monômes, ce qui améliorait la complexité moyenne[27].

Une autre source d'inefficacité concerne le codage des types enregistrements (voir la note 19 en bas de page 149) qui oblige l'utilisateur à effectuer une phase réduction pour accéder au contenu des champs. L'inefficacité est donc complètement proportionnelle au nombre d'accès à des champs d'enregistrements. Dans le code de `Field`, cette opération est effectuée à de nombreuses reprises (pour extraire des informations du corps commutatif de type `Field_Theory`, voir la section 8.3.2) et un gain de temps non négligeable pourrait être obtenu avec des enregistrements primitifs, où l'accès au contenu des champs est *immédiat*.

8.5 Discussion et extensions

8.5.1 Synthèse

La tactique `Field` se veut générale et traite tous les corps commutatifs. En particulier, elle contribue grandement au développement de la théorie des nombres réels en Coq. Elle permet une économie de temps précieux ainsi qu'un gain non négligeable de concision dans les scripts de preuves. Par ailleurs, étant intégralement réflexive, elle permet la construction de termes preuves plus petits que dans l'approche directe en utilisant les réécritures. L'utilisateur peut maintenant se désintéresser de certaines parties de preuves comme il le ferait dans une preuve informelle.

D'un point de vue plus technique, `Field` est un bon test pour $\mathcal{L}_{tac}$, non pas en ce qui concerne l'implantation mais plutôt le type de situations où il peut être utile. En regardant le cas de `Field`, il semblerait que les tactiques réflexives soient typiquement le genre de tactiques que l'on souhaite écrire à toplevel. En effet, seule la métaification nécessite l'utilisation, comme son nom l'indique, du métalangage. Étant particulièrement simple à concevoir et nécessitant du filtrage sur les termes Coq (d'un type non inductif), il est clair que la métaification tombe dans le contexte de ce langage de tactiques qui possède tous les opérateurs de filtrage nécessaires et dont l'objectif est d'automatiser des petites parties de preuves.

8.5.2 Travaux futurs

Dans un futur proche, une étude un peu plus poussée des performances de la tactique doit être effectuée. En effet, comme nous l'avons vu précédemment, certains cas mettent

[26]Cela peut sembler paradoxal dans la mesure où un codage réflexif est censé être efficace. Toutefois, il faudrait comparer avec un codage explicite (par réécritures) pour savoir s'il y a un problème dans la réflexion ou si l'algorithme utilisé possède une complexité élevée.

[27]La complexité dans le pire des cas resterait, par contre, la même. En effet, prenons un multiplicateur a et un monôme m, avec respectivement a_n et m_n composantes. Le pire des cas pour le premier algorithme est que a et m n'aient aucune composante commune. Dans ce cas, on a une complexité en $\theta(a_n.m_n)$. Avec un tri, le pire des cas, c'est aussi lorsqu'il n'y a aucune composante commune et que $\forall c_a \in a,\ \forall c_m \in m,\ c_a < c_m$. La complexité est alors toujours $\theta(a_n.m_n)$.

en évidence des sources d'inefficacité et il est important de savoir s'il est possible de les contrôler ou non.

Une extension intéressante serait d'avoir une option similaire à celle de `Ring` où l'on peut lui donner un terme, lequel est normalisé et remplacé dans le but courant. On pourrait faire de même avec `Field` qui simplifierait tous les inverses du terme avant de le remplacer dans le but courant. Étant donné une égalité, on est sûr de pouvoir simplifier tous les inverses, mais, pour un terme, il se peut que des inverses ne se simplifient pas et le test d'arrêt de `Field` devra être différent dans ce cas.

Enfin, on peut voir ce travail comme s'inscrivant dans une optique plus globale qui est de créer, à terme, une tactique plus puissante capable de gérer aussi les inéquations linéaires comme le fait la tactique `Omega` pour les entiers naturels et relatifs. En réalité, c'est déjà partiellement le cas avec la tactique `Fourier`, qui utilise `Field`, mais qui est exclusivement dédiée aux nombres réels.

Chapitre 9

Outils pour $\mathcal{L}_{tac}$

Dans ce chapitre, nous allons détailler l'environnement de programmation qui a été conçu pour $\mathcal{L}_{tac}$. En particulier, nous nous intéresserons à l'interfaçage avec Objective Caml [54] (à la fois langage d'implantation et métalangage de Coq), car, comme on a pu le voir dans le chapitre 8, $\mathcal{L}_{tac}$ est limité et il est parfois nécessaire de recourir à Objective Caml, tout en profitant de $\mathcal{L}_{tac}$. Par ailleurs, nous décrirons l'utilisation d'un *debugger* spécifique à $\mathcal{L}_{tac}$ et des exemples montreront comment il s'avère nécessaire dans certaines situations, notamment pour des tactiques qui peuvent potentiellement backtracker.

9.1 Interfaçage avec Objective Caml

9.1.1 Motivations

Comme nous l'avons dit précédemment, $\mathcal{L}_{tac}$ est un niveau supplémentaire de méta-langage permettant de pourvoir aux petites automatisations que l'utilisateur viendrait à effectuer dans le toplevel de Coq. En aucun cas, il n'est destiné à se substituer au métalangage complet de Coq, à savoir Objective Caml, même si, dans le chapitre 7, on a pu voir des exemples non triviaux utilisant exclusivement $\mathcal{L}_{tac}$. Ainsi, étant à toplevel, $\mathcal{L}_{tac}$ est limité et cela semble raisonnable d'accepter ses limitations, non pas comme un manque de complétude, mais plutôt comme une symbiose avec Objective Caml, qui est sollicité lorsque la complexité de la tactique à réaliser l'exige.

Cependant, il est clair qu'il est nécessaire de pouvoir utiliser $\mathcal{L}_{tac}$ au niveau d'Objective Caml. Tout d'abord, parce que l'utilisateur peut avoir commencé son développement en $\mathcal{L}_{tac}$ et s'apercevoir qu'il a besoin de caractéristiques plus *profondes* du système, que ce soit pour achever le codage ou le généraliser. Dans cette situation, il est souhaitable qu'il puisse porter tel quel son développement réalisé en $\mathcal{L}_{tac}$, afin de pouvoir le continuer, sans perte de temps, en Objective Caml. Une deuxième raison est de pouvoir bénéficier, en Objective Caml, du pouvoir syntaxique procuré par $\mathcal{L}_{tac}$. En effet, comme Objective Caml est aussi le langage d'implantation de Coq, l'utilisateur peut directement appeler les primitives de $\mathcal{L}_{tac}$, mais, en court-circuitant l'interprétation habituelle des tactiques à toplevel, il perd la syntaxe et est obligé de fournir aux primitives des informations internes dont il n'avait jamais eu à se soucier auparavant.

Réciproquement, l'utilisation directe ou indirecte de code Objective Caml dans $\mathcal{L}_{tac}$ semble aussi complètement souhaitable et, sans doute, plus intéressante. Se servir de $\mathcal{L}_{tac}$ en Objective Caml est effectivement important, mais, sans le processus inverse, il ne s'agit là que d'une simple inclusion syntaxique de code. Pour être suffisamment flexible, il faut également pouvoir agir sur les parties écrites en $\mathcal{L}_{tac}$ au moyen de code Objective Caml.

Pour réaliser une telle interface, il est nécessaire de modifier la syntaxe d'Objective Caml. Pour ce faire, on a le choix entre étendre la syntaxe ou utiliser une quotation. Pour comprendre comment la décision doit être prise, rappelons ce que sont une extension de syntaxe et une quotation.

9.1.2 Préliminaires : extensions de syntaxe et quotations

Pour réaliser les extensions de syntaxe (notamment dynamiques) et les quotations en Coq, nous utilisons Camlp4 [77, 76], un outil annexe à Objective Caml. Camlp4 se définit comme un Pré-Processeur-Pretty-Printer[1] pour Objective Caml. Il permet essentiellement de définir des grammaires, d'effectuer des extensions de syntaxe (d'Objective Caml, mais aussi des grammaires de l'utilisateur) et de réaliser facilement des quotations.

Extensions de syntaxe

Il s'agit de pouvoir étendre la syntaxe existante d'une certaine grammaire. Comme Camlp4 fournit un *parser* (analyseur grammatical) pour Objective Caml, il est alors très facile d'étendre la syntaxe d'Objective Caml. Par exemple, on peut se proposer d'avoir une syntaxe pour les fonctions plus proche de la notation mathématique usuelle, qui permet d'écrire $x \mapsto x + 1$ pour la fonction successeur. Pour ce faire, voici la session interactive suivante :

```
1   > ocaml -I `camlp4 -where`
2           Objective Caml version 3.02
3
4   # #load "camlp4o.cma";;
5           Camlp4 Parsing version 3.02
6
7   # #load "pa_extend.cmo";;
8   # #load "q_MLast.cmo";;
9   # open Pcaml;;
10  # EXTEND
11      expr: LEVEL "expr1"
12        [ [ i = LIDENT ; "|->"; e = expr ->
13            let p = <:patt< $lid:i$ >> in <:expr< fun $p$ -> $e$ >> ] ];
14    END;;
15  - : unit = ()
16  # let my_succ = x |-> x+1;;
17  val my_succ : int -> int = <fun>
18  # my_succ 1;;
19  - : int = 2
```

La ligne 1 permet de lancer le toplevel d'Objective Caml en lui indiquant (avec l'option `-I`) où se trouve la bibliothèque de Camlp4 (répertoire rendu par la commande `camlp4 -where`). En ligne 4, on charge Camlp4 et, notamment, le parser qui permet d'analyser des expressions d'Objective Caml dans leur syntaxe habituelle. En ligne 7, on charge le module qui va permettre d'étendre la grammaire précédemment importée. En ligne 8, on charge le module des

[1] D'où le p4 dans Camlp4.

quotations pour Objective Caml, qui permettront de continuer à utiliser une syntaxe concrète tout en fournissant directement des AST's[2] d'Objective Caml. Nous verrons plus précisément comment ces quotations fonctionnent dans la section suivante, qui leur est consacrée. En ligne 9, on rend les entrées de la grammaire d'Objective Caml (codée avec Camlp4) accessibles, de manière à pouvoir les étendre.

Des lignes 10 à 14, on réalise l'extension de syntaxe, au moyen de la structure `EXTEND ... END`. Plus particulièrement, en ligne 11, `expr` est l'entrée (des expressions) que l'on souhaite étendre, et on lui indique aussi que cela doit être réalisé au niveau (`LEVEL`) `expr1` (niveau où se trouve, entre autres, `function`). En ligne 12, il s'agit du motif de filtrage de l'analyse grammaticale. `LIDENT` est une entrée prédéfinie qui parse des identificateurs commençant par une minuscule. `"|->"` doit être filtré tel quel en tant que mot-clé, respectant cependant les règles du *lexer* (analyseur lexical), qui ne peut pas être modifié dynamiquement. En ligne 13, on doit rendre un AST qui correspond à ce qu'on a parsé, à savoir l'AST d'une fonction. Comme on ne connaît pas, *a priori*, la structure des AST's d'Objective Caml (qui peut sans doute se révéler relativement complexe) et que l'on souhaite pouvoir continuer à utiliser la syntaxe concrète d'Objective Caml, tout en y insérant les objets filtrés en ligne 12, on utilise un système de quotations qui permet de se brancher sur les différentes entrées de la grammaire. On peut ainsi former un *pattern* avec l'expression `<:patt< $lid:i$ >>`, où les `$...$` spécifie une antiquotation (on verra dans la section suivante de quoi il s'agit exactement) et permettent l'insertion d'AST's sous la forme d'expressions Objective Caml, ici c'est l'AST correspondant à l'identificateur filtré, qu'on estampille, au passage (avec `lid`), comme commençant avec une minuscule. Avec ce pattern (variable `p`), on peut alors former l'AST complet avec la quotation correspondant à `expr` en utilisant la syntaxe normale de `fun`[3].

Remarque 9.1.1 (Grammaires dynamiques) *À ce stade, on peut noter qu'en ligne 15,* `EXTEND ... END` *rend le type* `unit`*, c'est-à-dire que l'extension de syntaxe réalise un effet de bord. Le processus est donc complètement dynamique. Il ne s'agit pas de prendre une grammaire et d'en rendre une version étendue à recompiler, mais de changer une grammaire en place, modifiant, par la suite, le langage reconnu par cette grammaire. On peut ainsi profiter, de suite, de l'extension de syntaxe réalisée. C'est cet aspect, outre le fait que les extensions de syntaxe puissent être effectuées n'importe où, qui distingue profondément* Camlp4 *d'autres outils d'analyse grammaticale complètement statiques comme* Yacc.

De même et au delà de l'utilisation des quotations, cette possibilité de créer des grammaires dynamiquement modifiables est la raison essentielle de l'utilisation de Camlp4 *en* Coq. *L'utilisateur peut ainsi étendre très facilement la syntaxe de* Coq*, ce qui, de ce point de vue, le rend plus flexible que d'autres outils d'aide à la preuve.*

Pour tester l'extension réalisée, on définit, en ligne 16, une autre fonction successeur (`my_succ`) en utilisant la nouvelle syntaxe et on observe, en ligne 17, qu'elle est acceptée. Enfin, en lignes 18 et 19, une application de cette fonction montre qu'elle s'évalue correctement.

Camlp4 permet d'étendre une certaine grammaire, mais, de manière annexe, on peut également modifier une grammaire en éliminant des règles. Par exemple, si l'on décide que

[2]Un AST (*Abstract Syntax Tree*) est un objet d'un type abstrait représentant la structure d'expressions exprimées dans une syntaxe concrète. La phase d'analyse syntaxique permet de passer d'une syntaxe concrète à une syntaxe abstraite, i.e., d'expressions concrètes à des AST's.

[3]Remarquons ici que l'on est obligé d'empaqueter l'identificateur au moyen de la quotation des patterns (`patt`), puisque la règle de `fun` l'exige et qu'il faut respecter les règles de formation des AST's (le typage permet de vérifier que les AST's insérés sont corrects).

la notation $x \mapsto x + 1$ doit être la seule autorisée dans notre Objective Caml modifié, on utilisera les commandes suivantes :

```
20  # let match_case = Grammar.Entry.find expr "match_case";;
21  val match_case : Obj.t Grammar.Entry.e = <abstr>
22  # let fun_def = Grammar.Entry.find expr "fun_def";;
23  val fun_def : Obj.t Grammar.Entry.e = <abstr>
24  # DELETE_RULE
25      expr: "function"; OPT "|"; LIST1 match_case SEP "|"
26    END;;
27  - : unit = ()
28  # DELETE_RULE
29      expr: "fun"; patt LEVEL "simple"; fun_def
30    END;;
31  - : unit = ()
32  # let my_succ2 = function x -> x+1;;
33  Toplevel input:
34  # let my_succ2 = function x -> x+1;;
35                              ^^
36  Parse error: ';;' expected after [phrase] (in [top_phrase])
37  # let my_succ2 = fun x -> x+1;;
38  Toplevel input:
39  # let my_succ2 = fun x -> x+1;;
40                       ^
41  Parse error: [labeled_patt] expected after 'fun' (in [expr])
```

Les déclarations des lignes 20 et 22 permettent de rendre accessibles deux entrées privées (`match_case` et `fun_def`) intervenant dans l'entrée `expr`. Ensuite, des lignes 24 à 26, avec la commande `DELETE_RULE ... END`, on supprime la règle correspondant à `function` dans l'entrée `expr`. Pour connaître, la liste des symboles de la règle en question, on peut afficher la liste des règles de l'entrée `expr` avec la fonction `Grammar.Entry.print`. Des lignes 28 à 30, on fait de même pour `fun`. Enfin, on peut s'apercevoir, des lignes 32 à 41, qu'il n'est plus possible de définir de fonctions avec `function` ou `fun`. Éventuellement, il est possible de faire de même avec les déclarations implicites de fonctions, de la forme `let f a1 ... an = e`.

Quotations

Une quotation est une partie de code spécialement parenthésée (avec `<<...>>`) qui est traitée par une fonction utilisateur particulière appelée expanseur de quotation. Ces quotations ne peuvent être qu'en position d'expression ou de motif, ce qui les limite assez fortement par rapport aux extensions de syntaxe.

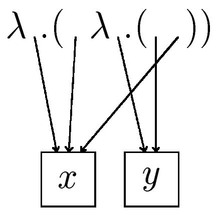
Historiquement, les quotations ont été introduites dans les premières versions de ML [34], pour représenter les propositions de la logique de LCF, sous une forme concrète. Elles ont ensuite suivi l'évolution de Edinburgh ML (INRIA, Cambridge), et étaient encore présentes dans Caml [89]. Elles ont alors été implantées dans une variante de Caml Light [53, 57], avant d'être disponibles dans Objective Caml par l'intermédiaire de Camlp4.

Comme exemple d'utilisation des quotations, nous nous proposons de coder le λ-calcul pur et, plus particulièrement, de lui fournir une syntaxe concrète directement utilisable dans des programmes Objective Caml. Définissons d'abord la structure des λ-termes :

```
> ocaml -I `camlp4 -where`
```

```
2          Objective Caml version 3.02
3
4   # type term =
5       | Var of string
6       | Binder of term ref
7       | Lambda of term ref * term
8       | App of term * term;;
```

En lignes 5, 6 et 7, on définit respectivement les variables, les lieurs, le λ et l'application. On peut remarquer l'indirection qui est effectuée sur les lieurs et le λ. Cela a pour objectif d'améliorer la représentation usuelle à base d'indices de De Bruijn. En effet, au moment de la construction du terme, on fait en sorte que le λ et le ou les lieurs associés partagent la même adresse. Ces références pointent alors sur un terme représentant la variable liée. Par exemple, le terme $\lambda x.(x\ \lambda y.(y\ x))$ sera représenté de la manière suivante :



Ainsi, dans cette représentation, si l'on souhaite β-réduire $(\lambda x.t\ u)$, il suffit d'affecter u à la référence représentant la variable x et de rendre le terme t. On évite alors la capture de variables due à la substitution (comme dans la représentation de De Bruijn), mais, surtout, la substitution est réalisée en $\theta(1)$, tandis qu'avec les indices de De Bruijn, elle s'effectue en $\theta(n)$, où n est la taille de t. Le seul inconvénient de cette représentation est qu'elle n'est pas canonique. En effet, après β-réduction, il reste des lieurs qui, en réalité, ne sont plus des lieurs, et qui pointent sur des termes éventuellement plus compliqués que des variables. Si on remplace ces pseudo-lieurs par leurs contenus, on obtient alors un terme complètement équivalent. Ces différentes représentations du même terme ne sont pas particulièrement problématiques, mais il faut en tenir compte dans toute fonction travaillant sur cette structure de termes.

Ensuite, la prochaine étape est de définir une grammaire qui va reconnaître la syntaxe concrète que l'on souhaite utiliser pour les λ-termes. Pour ce faire, nous utiliserons Camlp4[4] qui permet également de créer des grammaires :

```
# #load "camlp4o.cma";;
        Camlp4 Parsing version 3.02

# #load "pa_extend.cmo";;
# #load "q_MLast.cmo";;
# let gram = Grammar.create (Plexer.make ());;
val gram : Grammar.g = <abstr>
# let term = Grammar.Entry.create gram "term";;
val term : '_a Grammar.Entry.e = <abstr>
# let tpat = Grammar.Entry.create gram "tpat";;
val tpat : '_a Grammar.Entry.e = <abstr>
```

[4]Ce n'est pas obligatoire et on peut choisir d'opter, à ce stade, pour d'autres outils d'analyse syntaxique.

```
12  # EXTEND
13      term:
14        [ [ "["; x = LIDENT; "]"; t = term ->
15              <:expr<Lambda (ref (Var $str:x$),$t$)>>
16          | "("; t1 = term; t2 = term; ")" -> <:expr<(App $t1$ $t2$)>>
17          | x = LIDENT -> <:expr<Var $str:x$>>
18          | "'"; x = LIDENT -> <:expr<$lid:x$>> ] ];
19      tpat:
20        [ [ "["; x = LIDENT; "]"; t = tpat -> <:patt<Lambda ($lid:x$,$t$)>>
21          | "("; t1 = tpat; t2 = tpat; ")" -> <:patt<(App $t1$ $t2$)>>
22          | "@"; x = LIDENT -> <:patt<Var $lid:x$>>
23          | "!"; x = LIDENT -> <:patt<Binder $lid:x$>>
24          | x = LIDENT -> <:patt<$lid:x$>> ] ];
25    END;;
26  - : unit = ()
```

En ligne 1, on charge le parser Objective Caml de Camlp4. Ensuite, en ligne 4 et 5, on charge respectivement le module d'extension de syntaxe et le module des quotations pour Objective Caml. En ligne 6, on crée la grammaire des λ-termes en utilisant le lexer par défaut, à savoir celui d'Objective Caml utilisé par Camlp4 (il sera compatible avec la syntaxe que l'on souhaite). En ligne 9 et 11, on crée deux entrées de grammaire pour les λ-termes, une pour les expressions et une pour les motifs[5]. Enfin, des lignes 12 à 25, on fait l'extension de syntaxe (comme on a pu le voir dans la section précédente).

Dans l'entrée `term`, on peut remarquer, en ligne 18, la dernière règle, qui permet de parser des expressions de la forme `'x`, où `x` est un identificateur commençant par une minuscule, et de renvoyer un AST représentant directement cet identificateur. C'est ce qu'on appelle une antiquotation, c'est-à-dire, la possibilité de spécifier des expressions (ici, seulement un identificateur, mais cela pourrait être des expressions plus complexes) qui seront parsées dans le contexte de l'expression qui utilise la quotation et non plus dans celui de la quotation elle-même. Dans notre cas, l'identificateur représenté par `x` devra donc être connu au moment où la quotation est expansée. Dans les quotations prédéfinies d'Objective Caml (module `q_MLast.cmo`), les antiquotations sont parenthésées par `$...$` et peuvent être des expressions quelconques (sauf d'autres quotations) ou des identificateurs avec des labels. Les labels sont utilisés lorsque la syntaxe concrète n'est pas suffisante pour savoir quel AST construire. Par exemple, en ligne 15, on souhaite que `x` soit inséré en tant que `string` (label `str`), tandis qu'en ligne 18, on veut que ce soit un identificateur commençant par une minuscule (label `lid`). On pourra consulter le tutoriel [77] et le manuel de référence [76] de Camlp4, pour plus de détails sur ces quotations pour Objective Caml.

En ce qui concerne l'entrée `tpat`, il faut distinguer trois types de variables : les variables libres, les lieurs (qui pointent sur des variables) et les variables de filtrage. Ainsi, en ligne 22, les variables libres sont préfixées par `@` et, en ligne 23, les lieurs par `!`, tandis qu'en lignes 20 et 24, les variables de filtrage sont exprimées telles quelles comme dans les motifs d'Objective Caml.

Avant de définir la quotation, nous devons écrire la fonction qui remplace les variables liées (représentées par le constructeur `Var`, à l'issue du parsing) par des lieurs (constructeur `Binder`), partageant la même adresse que le λ (constructeur `Lambda`) correspondant[6] :

[5] En effet, il est très utile d'avoir aussi une syntaxe concrète dans les motifs, d'autant plus que la majorité des fonctions sur les termes seront codées par filtrage.

[6] Cette phase n'a pas pas pu être réalisée en une passe, avec le parsing, car cela aurait nécessité de pouvoir mettre des actions (sémantiques) dans les membres gauches des règles, ce qui n'est pas autorisé. À noter

```
1  # let put_binder =
2      let rec pbr env = function
3        | (Var x) as t ->
4          (try Binder (List.assoc t env) with | Not_found -> t)
5        | Lambda (r,t) -> Lambda (r,pbr ((!r,r)::env) t)
6        | App (t1,t2) -> App (pbr env t1,pbr env t2)
7        | t -> t in
8      pbr [];;
9  val put_binder : term -> term = <fun>
```

L'algorithme est très simple : dès qu'en ligne 5, on filtre un λ, on empile l'adresse correspondante dans l'environnement (variable `env`), puis, en lignes 3 et 4, dans le cas variable, si la variable est dans l'environnement, on met un lieur, sinon c'est une variable libre. En ligne 6, c'est le cas de l'application, il suffit de passer au contexte, tandis qu'en ligne 7, c'est le cas du lieur, on renvoie le terme tel quel car il n'y a, à ce stade, pas encore de lieurs (on pourrait aussi lever une exception pour le signaler).

Nous pouvons maintenant définir la quotation pour les expressions et les motifs :

```
1  # let term_exp s =
2      let ast = Grammar.Entry.parse term (Stream.of_string s)
3      and loc = (0,0) in
4      <:expr<put_binder $ast$>>;;
5  val term_exp : string -> MLast.expr = <fun>
6  # let term_pat s = Grammar.Entry.parse tpat (Stream.of_string s);;
7  val term_pat : string -> MLast.patt = <fun>
8  # Quotation.add "term" (Quotation.ExAst (term_exp, term_pat));;
9  - : unit = ()
10 # Quotation.default := "term";;
11 - : unit = ()
```

En ligne 1 à 4, il s'agit de l'expanseur pour les expressions. En ligne 1, on parse l'expression de la quotation selon l'entrée `term` de la grammaire précédemment définies, puis, en ligne 4, on lui ajoute un appel à la fonction `put_binder`, pour mettre les lieurs. Il est important de remarquer qu'à ce stade, rien n'est évalué, mais on ne fait que construire un nœud d'application sous la forme d'un AST, qui sera interprété ou compilé plus tard (dans une autre phase). En ligne 3, on est obligé de spécifier une location (pour les erreurs), car on forme un nouveau nœud d'AST[7], et, comme nous ne sommes pas intéressés sur une localisation précise des erreurs sur cette quotation, nous donnons une location factice `(0,0)`. En ligne 6, c'est l'expanseur pour les motifs, qui ne fait que parser selon l'entrée `tpat`. Enfin, en ligne 8, on déclare la quotation (pour les expressions et les motifs) et, en ligne 10, on la désigne comme quotation par défaut, c'est-à-dire qu'on pourra utiliser le parenthésage partiel `<<...>>` plutôt que la version complète `<:term<...>>` (toujours utilisable cependant).

Comme exemple d'utilisation de cette quotation, à la fois pour les expressions et les motifs, définissons la normalisation par β-réduction :

que ce n'est pas non plus possible en OcamlYacc, mais seulement en Yacc pur.

[7] On peut remarquer que, dans les grammaires que nous avons vues, la création de nouveaux nœuds d'AST, en utilisant les quotations prédéfinies d'Objective Caml, ne nécessitent pas de donner de locations, puisqu'elles sont implicitement fournies par les membres gauches des règles. On peut, cependant, redéfinir ces locations si on estime que la location par défaut est imprécise. Par exemple, dans la règle d'antiquotation, si on a `'x` et que `x` n'est pas connu au moment de l'expansion, on peut avoir envie que l'erreur soit signalée sur `x`, plutôt que sur `'x` en entier, comme c'est le cas actuellement. Pour plus de détails sur les locations, voir [77, 76].

```
1  # let rec beta_red = function
2      | <<([x]y z)>> -> x:=z; beta_red y
3      | <<@x>> as t -> t
4      | <<!x>> as t -> x:=beta_red !x;t
5      | <<[x]t>> -> let nt = beta_red t in <<[x]'nt>>
6      | <<(t1 t2)>> ->
7        let nt1 = beta_red t1
8        and nt2 = beta_red t2 in
9        <<('nt1 'nt2)>>;;
10 val beta_red : term -> term = <fun>
```

En ligne 2, le motif représente un β-redex et, comme on l'a vu précédemment, la réduction consiste simplement à affecter l'argument (`z`) à l'adresse correspondant au λ (`x`) et, comme il faut normaliser, on rapplique `beta_red` au corps du λ (`y`). En ligne 3, c'est le cas variable, qui est déjà en forme normale, on rend donc le terme tel quel. En ligne 4, c'est un lieur et comme, on ne sait pas, *a priori*[8], s'il pointe sur une variable (cela peut alors être un vrai lieur) ou sur un terme plus complexe (dû à une substitution antérieure), il faut normaliser son contenu et rendre le lieur. En ligne 5, c'est un λ, on doit alors réduire le corps (`t`) et rendre le λ avec le corps normalisé (`nt`). On voit, ici, l'utilisation de l'antiquotation (`'nt`), qui permet d'insérer telle quelle la variable `nt` dans la quotation (la variable `nt` est alors bien connue dans l'expression utilisant la quotation, ici, un `let...in`). Enfin, de la ligne 6 à 9, on traite l'application, de même que pour le λ, en passant au contexte et en utilisant deux antiquotations (`'nt1` et `'nt2`). On peut remarquer que cette fonction peut potentiellement boucler, car le λ-calcul pur n'est pas normalisant de manière générale.

Avant de voir des exemples d'utilisation, il nous faut définir une fonction permettant de *pretty-printer* les termes, afin d'avoir une meilleure représentation concrète lors de l'affichage :

```
1  # let rec printer = function
2      | <<@x>> -> x
3      | <<!r>> -> printer !r
4      | <<[r]t>> -> "["^(printer !r)^"]"^(printer t)
5      | <<(t1 t2)>> -> "("^(printer t1)^" "^(printer t2)^")";;
6  val printer : term -> string = <fun>
```

En ligne 2, le terme est une variable, qui est affichée telle quelle. En ligne 3, c'est lieur et il suffit d'afficher son contenu. En ligne 4, c'est un λ, on l'affiche conformément aux règles de parsing données précédemment, en regardant bien le contenu de l'adresse correspondant au λ (normalement une variable). En ligne 5, c'est l'application, dont le traitement, comme pour le λ, obéit à la syntaxe que l'on s'est fixée.

Voici quelques petits exemples de normalisation :

```
# print_endline (printer (beta_red <<([x]x y)>>));;
y
- : unit = ()
# print_endline (printer (beta_red <<([x]([y](y z) x) ([z](x z) y))>>));;
((x y) z)
- : unit = ()
```

[8]Pour savoir, *a posteriori*, si un lieur est un vrai lieur ou non, il faudrait mémoriser les adresses des λ *traversés* et regarder si on trouve l'adresse du lieur dans ces adresses. Si l'adresse est trouvée alors c'est un vrai lieur, sinon il a été affecté (substitution) et peut donc être supprimé (remplacé par son contenu).

```
7  # let delta = <<[x](x x)>>;;
8  val delta : term =
9    Lambda
10    ({contents=Var "x"},
11     App (Binder {contents=Var "x"}, Binder {contents=Var "x"}))
12 # print_endline (printer (beta_red <<('delta 'delta)>>));;
13 Stack overflow during evaluation (looping recursion?).
```

En lignes 1 et 2, puis 4 et 5, on peut voir deux exemples d'utilisation sur des termes fortement normalisants. En ligne 7, on définit le terme $\Delta = \lambda x.(x\ x)$, bien connu en λ-calcul pur, puisqu'appliqué à lui-même, en lignes 12 et 13, il n'est pas normalisant.

Domaines d'application et discussion

De manière générale, les extensions de syntaxe sont plus puissantes que les quotations. En effet, comme on l'a vu précédemment, les quotations sont limitées aux expressions et aux motifs, alors que les extensions de syntaxe peuvent potentiellement modifier toutes les entrées de la grammaire d'Objective Caml et, de ce fait, agir plus *profondément* sur le parser (malgré cette limitation, les quotations peuvent parfaitement être utilisées dans l'exemple d'extension de syntaxe sur une nouvelle notation fonctionnelle, puisqu'on agit uniquement sur les expressions). Ainsi, dans l'exemple des λ-termes, on aurait pu choisir d'étendre la syntaxe, plutôt que d'utiliser une quotation. Cependant, il aurait fallu faire attention à ne pas rentrer en conflit avec d'autres règles du niveau des expressions ou des motifs. Il aurait peut-être même fallu préfixer par un mot-clé spécifique pour pouvoir parser les λ-termes conformément à la syntaxe concrète choisie. Dans le système de quotations, le problème n'apparaît pas puisque l'expression est spécialement parenthésée et l'expanseur branche directement sur l'entrée concernée.

On s'aperçoit donc que, bien qu'un fossé technique semble opposer les extensions de syntaxe et les quotations, il apparaît que les deux systèmes ont, en pratique, leurs domaines de prédilection. Les extensions de syntaxe seront plus avantageusement utilisées pour modifier du code (exécutable), tandis que les quotations pourront servir à fournir une syntaxe concrète à des données (représentées typiquement par un type inductif).

Dans le cadre de l'interfaçage de $\mathcal{L}_{tac}$ avec Objective Caml, utiliser les tactiques en Objective Caml ne semble pas être une extension de code, mais plutôt des données que l'on souhaite utiliser dans leur syntaxe habituelle et qui s'apparentent naturellement à des AST's (de Coq). Par ailleurs, la réalisation d'une extension de syntaxe poserait quelques problèmes techniques. Tout d'abord, dans cette optique, on serait plutôt amené à rendre, dans la règle de grammaire correspondante, une tactique, c'est-à-dire une fonction. Or, pour ce faire, il faut appeler l'interpréteur de tactiques (qui fonctionnent sur des AST's), ce qui semble un peu étrange pour un parser. Un autre problème concerne l'utilisation d'expressions Objective Caml dans $\mathcal{L}_{tac}$, qui ne peut être fait qu'en rajoutant une entrée dans la grammaire des tactiques. Cela obligerait à créer un nœud spécial dans les AST's de Coq, ce qui, d'une certaine manière, parasite cette structure. Enfin, il est clair qu'avec une telle extension de syntaxe, il serait particulièrement difficile de distinguer les parties propres à $\mathcal{L}_{tac}$ de celles d'Objective Caml.

Avec une quotation, les problèmes précédents ne se posent pas. La quotation fournira un AST de Coq, que l'on pourra manipuler sous cette forme, notamment en appelant, par la suite, l'interpréteur de tactiques. L'utilisation d'Objective Caml dans $\mathcal{L}_{tac}$ se fera par antiquotations (seulement un identificateur) représentant des fonctions qui seront *cachées* dans l'AST. Quant à la distinction visuelle entre les deux métalangages, elle sera pleinement assurée par le parenthésage spécifique des quotations (`<<...>>`).

9.1.3 Quotation pour $\mathcal{L}_{tac}$

On ne va décrire que partiellement le codage de cette quotation. Notamment, on sera amené à faire référence à des fichiers de la version V7.1 de Coq, que l'on localisera relativement au répertoire d'installation des sources.

Description générale

La grammaire des tactiques est définie dans les fichiers `parsing/g_tactic.ml4` (tactiques primitives) et `parsing/g_ltac.ml4` ($\mathcal{L}_{tac}$). Cet entrée est ensuite utilisée pour définir la quotation sur les tactiques, appelée `tactic`, dans le fichier `parsing/q_coqast.ml4`. Ce n'est pas la quotation par défaut (qui est celle sur les termes de Coq, à savoir `constr`) et doit donc être préfixée à chaque appel. Cette quotation rend des AST's de Coq avec le type `Coqast.t`, défini dans les fichiers `parsing/coqast.ml[i]`, mais comme c'est une quotation, ses expanseurs doivent rendre des AST's d'Objective Caml. Ceci est fait, entre autres, avec la fonction `expr_of_ast`, qui sert aussi à d'autres quotations et qui est définie dans `parsing/q_coqast.ml4`. Dans cette phase de traduction, on reconnaît les antiquotations qui sont des identificateurs préfixés par `$` et qui doivent être de type `Coqast.t`.

Utiliser $\mathcal{L}_{tac}$ en Objective Caml

À ce stade, on peut déjà écrire des tactiques Coq au niveau Objective Caml, tout en utilisant leur syntaxe habituelle (disponible à toplevel). Par exemple, si l'on souhaite écrire, en Objective Caml, une tactique qui cherche, dans le contexte local, une hypothèse sous la forme $A \wedge B$, où $\wedge$ est la conjonction sur les propositions, et l'élimine (pour donner deux hypothèses de types A et B), on peut le réaliser comme suit :

```
> coqtop.byte
Welcome to Coq 7.1 (September 2001)

Coq < Drop.
        Objective Caml version 3.01

        Camlp4 Parsing version 3.01

# #use "include";;
# let break_and =
    let loc = dummy_loc in
    <:tactic<
      Match Context With
      | [ id : ?/\? |- ? ] -> Elim id>>;;
val break_and : Coqast.t =
  Match Context With
  | [ id : ?/\? |- ? ] -> Elim id
# open Tacinterp;;
# add_tacdef (id_of_string "BreakAnd") break_and;;
BreakAnd is defined
- : unit = ()
# go ();;
```

```
23
24   Coq < Goal (A,B:Prop)A/\B->A.
25
26   Unnamed_thm < Intros;BreakAnd.
27   1 subgoal
28
29     A : Prop
30     B : Prop
31     H : A/\B
32     ============================
33      A->B->A
```

En ligne 1, on lance la version bytecode de Coq. C'est, en effet, la seule possibilité pour utiliser la commande `Drop` de la ligne 4, que nous avons vu dans la section 6.2 du chapitre 6, et qui permet de lancer un toplevel Objective Caml (parsé par Camlp4). Ce toplevel permet de charger dynamiquement d'autres modules compilés en bytecode (pouvant être des tactiques écrites en Objective Caml), mais il sert aussi de debugger pour des problèmes ponctuels pour lesquels on souhaite un accès rapide. D'ailleurs, en ligne 9, la première commande sert à installer une série de pretty-printers à ces fins. Elle *ouvre* aussi certains modules et définit la commande (go) qui permet de regagner le toplevel de Coq. Des lignes 10 à 14, on définit notre tactique. Pour ce faire, on utilise la quotation des tactiques, parenthésée par `<:tactic<...>>` et où la syntaxe est exactement celle de $\mathcal{L}_{tac}$. En ligne 11, comme on a pu le voir dans la section précédente dans l'expanseur de quotation pour les expressions de λ-termes, nous donnons une location factice (`dummy_loc`[9]), car nous ne sommes pas intéressés par une localisation précise des erreurs sur cette quotation. On peut remarquer, au passage, des lignes 16 à 17, les effets du pretty-printer sur les tactiques, qui a été chargé précédemment. Ensuite, en ligne 18, on ouvre le module d'interprétation des tactiques, afin de pouvoir, en ligne 19, déclarer notre tactique comme définition toplevel sous le nom `BreakAnd`, avec lequel elle sera invoquée. En ligne 22, on revient au toplevel de Coq afin de tester la tactique que l'on vient d'enregistrer. En ligne 24, on crée un nouveau but qui contiendra un $\wedge$ en hypothèses et, on peut ainsi, en ligne 26, introduire les produits pour appeler enfin `BreakAnd`, dont le résultat, des lignes 27 à 33, s'avère concluant.

Ce petit exemple montre que la première partie de nos motivations de la section 9.1.1 est complètement effective, à savoir, qu'il nous est possible d'utiliser $\mathcal{L}_{tac}$ en Objective Caml. Cependant, la réciproque s'avère plus complexe, dans la mesure où il s'agit d'insérer du code Objective Caml dans les AST's de Coq, et où rien n'est, *a priori*, prévu pour ce faire.

Utiliser Objective Caml en $\mathcal{L}_{tac}$

Pour pouvoir écrire du code Objective Caml dans un script $\mathcal{L}_{tac}$, le moyen naturel est d'utiliser les antiquotations. Comparées aux antiquotations des quotations prédéfinies de Camlp4 pour créer des AST's d'Objective Caml, les antiquotations de la quotation `tactic` sont plus primitives puisqu'elles sont limitées à un identificateur préfixé par `$` (alors qu'avec celles de Camlp4, on pouvait mettre des expressions Objective Caml, en les parenthésant par `$...$`). Cependant, cette limitation n'est que d'ordre pratique, puisqu'un `let...in` permet de s'affranchir complètement de cette contrainte et on peut donc, de même, utiliser toute expression Objective Caml. Ainsi, le problème ne concerne pas la classe syntaxique des expressions Objective Caml autorisées, mais plutôt le type de ces expressions. En effet, étant insérées dans la quotation `tactic`, ces parties de code devront impérativement rendre des

[9]Cet identificateur s'évalue bien sur `(0,0)`.

AST's de Coq, c'est-à-dire de type `Coqast.t`. Cette contrainte de typage peut sembler forte, mais, en réalité, elle est facilement satisfiable au moyen de la quotation `tactic` elle-même.

Toutefois, en pratique, on s'aperçoit que ce n'est pas encore suffisant. En particulier, ces parties de code ne peuvent pas interagir avec l'environnement d'interprétation car, elles n'ont aucune information sur leur contexte d'évaluation. Par exemple, si on a le script suivant :

```
Match Context With
| [|- ?1] -> $is_prod.
```

où `is_prod` est une tactique qui renvoie `Idtac`, si la conclusion du but courant, lié à la métavariable `?1`, est un produit et `Fail`, sinon. Le problème est que `is_prod` ne peut pas accéder à la métavariable `?1` et, de fait, à la conclusion du but[10], afin d'effectuer son test.

Ainsi, pour que les parties de code Objective Caml soient réellement utilisables avec $\mathcal{L}_{tac}$, elles doivent être systématiquement paramétrées avec l'environnement d'interprétation, qui leur sera donné lors de leur évaluation. Le type de cet environnement est `interp_sign`, défini dans les fichiers `proofs/tacinterp.ml[i]`. Nous ne donnerons pas de détails sur les composantes de cet environnement, dont on pourra faire abstraction pour la suite. Les parties de code Objective Caml à insérer devront donc être de type `interp_sign -> Coqast.t`, ce qui pose un nouveau problème de typage puisque les objets désignés par antiquotations ne peuvent être que de type `Coqast.t`.

Pour résoudre cette contrainte de typage, une première solution serait de créer un nouveau constructeur dans le type `Coqast.t`, disons `Ocaml`, de type `interp_sign -> Coqast.t`, afin de préfixer les parties de code Objective Caml à insérer. Mais, cela poserait un problème d'ordre sémantique, puisque des informations concernant l'interprétation *parasiteraient* la structure des AST's. L'interprétation fonctionnant sur des AST's, on serait même amené à fusionner les fichiers d'AST's avec les fichiers d'interprétation, ce qui, même si c'est techniquement possible, aurait tendance à mélanger des notions parfaitement distinctes et ce, au moins chronologiquement, dans la mesure où l'analyse syntaxique a *normalement* lieu avant l'évaluation.

Cette première solution étant *sémantiquement* incorrecte, la deuxième idée, pour laquelle nous avons opté, a été de *court-circuiter* le typage d'Objective Caml. En effet, cette possibilité est offerte par le type `Coqast.t` lui-même, avec un nœud particulier appelé `Dynamic`, de type `loc * Dyn.t`, où `loc` est le type des locations habituelles de Camlp4 et où `Dyn.t` est une sorte de type *magique*, qui permet d'y ranger des objets de tous types (d'où son nom, puisque le type de ces objets est, pour ainsi dire, dynamique). Le type `Dyn.t` est défini dans les fichiers `lib/dyn.ml[i]` et constitue une sorte d'interface *protégée* (pour être exact, on devrait plutôt dire *un peu plus protégée*) au module `Obj` d'Objective Caml. Ce module d'Objective Caml est non documenté et pour cause, il permet de passer outre le typage du système et expose, par là-même, l'utilisateur non averti à des évaluations hasardeuses. En général, on utilisera `Obj` dans des situations où l'on estime que le typeur est trop strict et oblige à effectuer un codage coûteux pour passer le typage (c'est typiquement notre cas, ici), ou lorsqu'il ne semble pas y avoir, *a priori*, de solutions typables à un problème donné. Dans tous les cas, le développeur doit assurer lui-même la correction de l'exécution des parties de code recourant à ce module.

Pour détourner le typage, deux fonctions sont essentiellement utilisées, à savoir `Obj.repr`, de type `'a -> Obj.t`, qui permet d'enfermer n'importe quelle expression, et `Obj.magic`, de type `'a -> 'b`, qui permet, entre autres, de récupérer les contenus d'expressions de type `Obj.t`. Par exemple, avec ces fonctions, on peut créer des listes hétérogènes et les manipuler :

[10] En fait, elle a d'autres moyens d'y accéder, en appelant certaines fonctions Objective Caml, mais, dans cette situation, on réalise un codage direct et on ne profite plus des fonctionnalités de $\mathcal{L}_{tac}$.

```
1  > ocaml
2          Objective Caml version 3.02
3
4  # let l = [Obj.repr 1; Obj.repr [1; 2]];;
5  val l : Obj.t list = [<abstr>; <abstr>]
6  # 1 + (Obj.magic (List.hd l));;
7  - : int = 2
8  # List.map succ (Obj.magic (List.nth l 1));;
9  - : int list = [2; 3]
10 # List.map succ (Obj.magic (List.hd l));;
11 Segmentation fault
```

En ligne 4, grâce à `Obj.repr`, on crée une liste (`l`) apparemment homogène, avec des éléments de type `Obj.t`, mais en réalité, le premier élément est un entier (`1`) et le deuxième une liste d'entiers (`[1 ; 2]`). En ligne 6, avec `Obj.magic`, on peut récupérer l'entier correspondant au premier élément de `l`, puis l'utiliser comme tel. En ligne 8, on peut faire de même avec le deuxième élément de `l`, qui est une liste. Enfin, en ligne 10, on peut utiliser le premier élément de `l` comme une liste, puisque `Obj.magic` renvoie un type polymorphe et qu'il se contraint naturellement à être une liste d'entiers (de par son contexte). Toutefois, cet élément est, en réalité, un entier et, le typage ayant été trompé à tort, l'exécution échoue.

D'un point de vue implantation, ces deux fonctions `Obj.repr` et `Obj.magic` ne sont autres que des fonctions identités, `Obj.t` ne définissant pas une structure particulière. Elles ne sont là que pour détourner le typage et le petit exemple précédent montre que leur utilisation peut s'avérer aussi pratique que dangereuse.

Le module `Dyn` de Coq permet au développeur de manipuler le module `Obj` avec un contrôle supplémentaire. En effet, si l'on injecte dans `Obj.t` (ce n'est, pour le moment, pas ce que l'on cherche à faire) des objets hétérogènes, il sera difficile de savoir de quel objet il s'agit lorsqu'on le récupérera. L'idée est donc de taguer les objets abstraits, ainsi `Dyn.t` se définit comme `string * Obj.t`. Deux fonctionnalités sont alors exportées : `Dyn.create`, qui prend un tag en argument, enregistre le tag (pour éviter les redéfinitions) et rend la fonction d'injection ainsi que la fonction de récupération, puis `Dyn.tag`, qui rend le tag d'un objet de type `Dyn.t`. Grâce à ce système de tags, on peut *cacher* des objets de différents types dans `Dyn.t` et assurer leur correcte interprétation par la suite.

Ainsi, pour stocker les parties de code Objective Caml, de type `interp_sign->Coqast.t`, on crée, dans le fichier `tacinterp.ml`, une nouvelle entrée avec le tag `"tactic"` et, au passage, on écrit aussi une fonction permettant de construire directement le nœud d'AST :

```
open Dyn

let ((tactic_in : (interp_sign -> Coqast.t) -> Dyn.t),
     (tactic_out : Dyn.t -> (interp_sign -> Coqast.t))) = create "tactic"

let tacticIn t = Dynamic (dummy_loc,tactic_in t)
let tacticOut = function
  | Dynamic (_,d) ->
    if (tag d) = "tactic" then
      tactic_out d
    else
      anomalylabstrm "tacticOut" [<'sTR "Dynamic tag should be tactic">]
  | ast ->
```

```
    anomalylabstrm "tacticOut"
      [<'sTR "Not a Dynamic ast: "; print_ast ast>]
```

où la fonction `anomalylabstrm` (définie dans `lib/util.ml[i]`) permet de lever une exception (ici `Util.Anomaly`), en spécifiant le nom de la fonction qui a levée l'exception, ainsi que le message d'erreur. Les fonctions rendues par `Dyn.create` étant complètement polymorphes, on spécifie les types de `tactic_in` et `tactic_out`, de manière à ce que le développeur puisse connaître le type des objets stockés dans `Dyn.t`. Ce n'est pas obligatoire et peut être vu comme un souci de maintenir un code minimalement documenté. La fonction `tacticIn` crée l'AST avec une location factice, car il n'y a pas de localisation d'erreurs plus précise que sur tout l'AST. La fonction `tacticOut` déstructure l'AST (qui doit être un nœud `Dynamic`), vérifie que le tag est bien `"tactic"` et rend alors la fonction stockée dans l'objet de type `Dyn.t`.

On peut donc maintenant insérer certaines parties de code Objective Caml dans des AST's de Coq. Reste à l'interpréteur de tactiques de reconnaître ces parties pour les évaluer correctement. Ceci est fait dans la fonction principale d'interprétation `val_interp` (dans le fichier `tacinterp.ml`), qui filtre l'AST à évaluer :

```
| Dynamic(_,t) ->
  let tg = (tag t) in
  if tg = "tactic" then
    let f = (tactic_out t) in val_interp ist (f ist)
  else ...
```

où `ist` est l'environnement d'interprétation (de type `interp_sign`). On peut remarquer que l'on vérifie le tag du `Dynamic` (avec `tag`), de manière à être sûr de l'objet que l'on doit évaluer[11]. Ensuite, il suffit de récupérer l'objet en question (avec `tactic_out`), de lui donner l'environnement d'interprétation `ist` (c'est une fonction de type `interp_sign -> Coqast.t`), puis d'interpréter le résultat (avec `val_interp`).

Bien qu'un pas en avant ait été fait en rendant possible l'utilisation de certaines fonctions Objective Caml (d'un type donné) dans $\mathcal{L}_{tac}$, ce n'est pas encore entièrement satisfaisant. En effet, ces fonctions Objective Caml ont accès à l'environnement d'interprétation, qui contient des données déjà interprétées par l'évaluateur, donc proches du type des valeurs de l'interpréteur, à savoir `Tacinterp.value` (type rendu, entre autres, par `val_interp`). Ces données peuvent être manipulées et, il est, ensuite, naturel de vouloir les réinjecter dans les scripts de $\mathcal{L}_{tac}$. De même que précédemment, se pose alors un problème de typage, puisque le type `Tacinterp.value` est incompatible avec le type `Coqast.t` et qu'il n'est pas non plus concevable, pour des raisons plus sémantiques, de créer un nouveau nœud d'AST contenant des informations relatives à l'interprétation. L'idée est donc de procéder exactement de la même façon que pour les fonctions de type `interp_sign -> Coqast.t`, en utilisant un nouveau type de `Dyn.t`, tagué `"value"` (car, cette fois, il s'agit de stocker des objets de type `Tacinterp.value`). Ainsi, dans le fichier `tacinterp.ml`, on complète les déclarations concernant les objets de `Dyn.t`, tagués `"tactic"`, par :

```
let ((value_in : value -> Dyn.t),
     (value_out : Dyn.t -> value)) = create "value"

let valueIn t = Dynamic (dummy_loc,value_in t)
```

[11] Le tag `"tactic"` est, pour l'instant, le seul, mais cette vérification est une mesure de précaution en vue d'éventuelles extensions futures (dans les objets stockés dans `Dyn.t`), qui, de surcroît, pourraient être réalisées à d'autres fins que pour l'évaluation.

```
let valueOut = function
  | Dynamic (_,d) ->
    if (tag d) = "value" then
      value_out d
    else
      anomalylabstrm "valueOut" [<'sTR "Dynamic tag should be value">]
  | ast ->
    anomalylabstrm "valueOut"
      [<'sTR "Not a Dynamic ast: "; print_ast ast>]
```

L'interpréteur doit aussi être étendu comme suit :

```
| Dynamic(_,t) ->
  let tg = (tag t) in
  if tg = "tactic" then
    let f = (tactic_out t) in val_interp ist (f ist)
  else if tg = "value" then
    value_out t
  else ...
```

Lorsque l'on reconnaît le tag `"value"`, il suffit, en effet, de récupérer la valeur (avec `value_out`) et de la rendre telle quelle (puisqu'elle a déjà été interprétée).

Une autre évolution importante, et un peu orthogonale à l'interpréteur de tactiques, serait de pouvoir insérer des termes Coq interprétés (de type `constr`, dans `kernel/term.ml[i]`) dans des termes sous forme d'AST's (toujours de type `Coqast.t`). Pour ce faire, on crée, de même que précédemment, un troisième tag, appelé `"constr"` et on exporte les fonctions habituelles `constr_in`, `constr_out`, `constrIn`, puis `constrOut` (voir fichiers `pretyping/pretyping.ml[i]` et `parsing/astterm.ml[i]`). Si cette déclaration de tag n'est pas effectuée au niveau de l'interpréteur de tactiques, c'est parce qu'il n'interprète pas directement les termes, mais appelle une fonction séparée lorsqu'il rencontre l'AST d'un terme. Ainsi, c'est cette fonction qui est modifiée afin d'évaluer correctement ces nouveaux nœuds d'AST (les changements concernent essentiellement la fonction `Pretyping.pretype`). Toutefois, pour plus de souplesse, on étend aussi l'interpréteur de tactiques, afin d'injecter ces AST's dynamiques, tagués `"constr"`, dans le type `value`. Si l'utilisateur a déjà *empaqueté* son terme avec `constrIn`, il ne sera alors pas obligé de définir une seconde version avec `tacticIn`, afin de pouvoir l'insérer, en tant qu'expression de tactiques, dans un script de $\mathcal{L}_{tac}$. L'extension proprement dite se fait de la manière suivante :

```
open Pretyping

| Dynamic(_,t) ->
  let tg = (tag t) in
  if tg = "tactic" then
    let f = (tactic_out t) in val_interp ist (f ist)
  else if tg = "value" then
    value_out t
  else if tg = "constr" then
    VArg (Constr (constr_out t))
  else ...
```

où `Constr` est un constructeur du type `tactic_arg` (dans `proofs/proof_type.ml[i]`), représentant les arguments primitifs des tactiques, et `VArg`, un constructeur du type `value`.

Ainsi, avec ces trois tags, on atteint un niveau acceptable de flexibilité pour insérer du code Objective Caml dans des scripts $\mathcal{L}_{tac}$. On peut tester ce nouveau système en se proposant d'écrire une tactique qui réalise une η-expansion sur une hypothèse donnée du contexte local :

```
> coqtop.byte
Welcome to Coq 7.1 (September 2001)

Coq < Drop.
        Objective Caml version 3.01

        Camlp4 Parsing version 3.01

# #use "include";;
# open Retyping;;
# open Tacinterp;;
# let eta_expansion ist =
    let hyp  = valueIn (List.assoc (id_of_string "id") ist.lfun)
    and prd  = constrIn (List.assoc 1 ist.lmatch)
    and arg  = constrIn (List.assoc 2 ist.lmatch) in
    let targ = constrIn (get_assumption_of ist.env ist.evc
                 (get_type_of ist.env ist.evc (constrOut arg))) in
    let ntm  = <:constr<([_:$targ]($prd _) $arg)>> in
    <:tactic<Refine [x:=$ntm]?;Cut x;Cbv Delta;
                [Clear x;Clear $hyp;Intro $hyp|Auto]>>;;
val eta_expansion : Tacinterp.interp_sign -> Coqast.t = <fun>
# let make_eta_exp =
    let t_eta_expansion = tacticIn eta_expansion in
    <:tactic<
      Match Context With
      | [ id:(?1 ?2) |- ? ] -> $t_eta_expansion>>;;
val make_eta_exp : Coqast.t =
  Match Context With
  | [ id : (?1 ?2) |- ? ] -> <dynamic [tactic]>
# add_tacdef (id_of_string "MakeEtaExp") make_eta_exp;;
MakeEtaExp is defined
- : unit = ()
# go();;

Coq < Parameters A:Set;P,Q:A->Prop.
A is assumed
P is assumed
Q is assumed

Coq < Goal (a:A)(P a)->(Q a).

Unnamed_thm < Intros. MakeEtaExp.
1 subgoal

  a : A
```

```
46      H : ([_:A](P _) a)
47      ============================
48       (Q a)
```

En ligne 1, on lance la version bytecode de Coq, afin de pouvoir utiliser, en ligne 4 et de même que précédemment, la commande `Drop`, qui permet d'accéder à un toplevel Objective Caml, parsé par Camlp4. En ligne 9, on interprète le fichier `include`, qui, comme on a pu le voir, *configure* l'environnement de développement. On *ouvre*, ensuite, en lignes 10 et 11, les modules `Retyping`, qui permet de retyper des termes déjà interprétés[12], et `Tacinterp`, qui, entre autres, exporte les nouvelles fonctions d'injection et de déstructuration des AST's dynamiques, dont nous avons parlé (il nous sera aussi nécessaire pour déclarer la nouvelle définition de tactique).

Des lignes 12 à 21, on a la fonction `eta_expansion`, qui, comme son nom l'indique, réalise la η-expansion. À ce stade, on fait plusieurs suppositions : l'hypothèse est de la forme $(P\ a)$, le P et le a sont respectivement liés aux métavariables 1 et 2, dans le contexte des métavariables (champ `Tacinterp.interp_sign.lmatch`), et le nom de l'hypothèse est lié à `id` dans le contexte des variables (champ `Tacinterp.interp_sign.lfun`). Ces suppositions se justifieront par la suite, sachant que l'hypothèse sera capturée et déstructurée par filtrage (`Match Context`). En ligne 12, on récupère le nom de l'hypothèse (liée à `id`) du contexte des variables (`ist.lfun`, avec `ist` de type `Tacinterp.interp_sign`) et on l'empaquette dans un AST dynamique `"value"`, car on l'utilisera dans un script $\mathcal{L}_{tac}$. En lignes 14 et 15, on récupère le P et le a en les réinjectant aussi dans un AST dynamique, de tag `"constr"` cette fois, afin de pouvoir les utiliser, plus loin, dans une quotation `constr`. En lignes 16 et 17, on type l'argument a, de manière à pouvoir donner le type du λ du η-redex à créer et, au passage, on l'injecte en AST dynamique `"constr"`. En ligne 18, on crée l'expression η-expansée avec un λ anonyme[13] (symbole `_`) et ce, très facilement au moyen de la quotation `constr` (on la laisse sous forme d'AST, car on l'utilise dans la partie $\mathcal{L}_{tac}$ qui suit). En ligne 19 et 20, on effectue le remplacement de l'hypothèse d'origine par l'hypothèse η-expansée, avec le même nom. On peut remarquer que l'on ne fait pas directement une coupure, avec `Cut`, car cette tactique réduit le terme introduit, ce qui fait que l'on se retrouve avec l'hypothèse d'origine. Pour arriver tout de même à faire accepter ce terme, l'astuce consiste à utiliser la tactique `Refine` (qui ne réduit pas), pour introduire un `let-in` dans le contexte local. Il suffit, ensuite, de faire la coupure avec l'identificateur du `let-in` (ici `x`), et de le déplier.

Une fois que l'on a la fonction de η-expansion, il reste à récupérer l'hypothèse proprement dite pour la lui *appliquer*. Ceci est fait, de la ligne 22 à 26, par la fonction `make_eta_exp`. Plus particulièrement, on transforme d'abord, en ligne 23, la fonction `eta_expansion` en AST, de manière à pouvoir l'utiliser dans la quotation `tactic`. Puis, des lignes 24 à 26, on passe dans le langage de $\mathcal{L}_{tac}$, pour récupérer l'hypothèse voulue avec un `Match Context`, et on appelle la fonction de η-expansion avec l'AST correspondant, qui a été précédemment construit.

Enfin, on enregistre la tactique, en ligne 30, sous le nom `MakeEtaExp`, avant de retourner au toplevel de Coq, en ligne 33. On déclare alors, en ligne 35, un ensemble `A` de `Set`, ainsi que deux prédicats, `P` et `Q`, sur cet ensemble. En ligne 40, on crée un nouveau but, qui, après introduction des produits, désigne l'hypothèse de type `(P a)` pour l'application de

[12] C'est une phase de retypage car, lors de l'interprétation, c'est-à-dire du passage de `Coqast.t` à `constr`, les termes sont typés. Ainsi, un terme `constr` issu de l'interprétation peut être considéré comme sûr. C'est une caractéristique nouvelle de la version V7 de Coq, et elle s'inscrit complètement dans le contexte de l'effort drastique, qui a pu être réalisé, afin de restructurer, rigoureusement et de manière interne, l'ensemble du code.

[13] Les λ anonymes sont utiles si l'on souhaite se passer de donner un nom au lieur et, de fait, éviter une éventuelle capture de variable non désirée.

`MakeEtaExp`, en ligne 42. On remarque ainsi, de la ligne 43 à 48, que la tactique s'exécute avec succès et construit bien un η-redex autour de `P`.

9.1.4 Exemples d'utilisation

En plus des exemples donnés précédemment dans la section 9.1.3, nous allons décrire brièvement deux autres applications de la quotation dédiée à $\mathcal{L}_{tac}$, qui, de plus, sont distribuées avec la version V7 de Coq. Nous utiliserons toujours la même convention de référencement des fichiers sources.

La tactique `Tauto`

Comme nous l'avons vu dans le chapitre 7, la tactique `Tauto` a été entièrement réécrite en utilisant $\mathcal{L}_{tac}$, et remplace désormais l'ancien code (en Objective Caml pur) disponible dans les versions V6 et antérieures. On a pu remarquer qu'il y avait de nombreux avantages à ce remplacement, dont la concision, la lisibilité, le gain de performances ou la portabilité. Toutefois, la version donnée dans le chapitre 7 n'est pas entièrement compatible avec la version d'origine, qui, notamment, pouvait résoudre modulo isomorphismes des connecteurs logiques[14]. Au nom de la *backward-compatibility*, il a donc fallu adapter le script de $\mathcal{L}_{tac}$ pour traiter cette caractéristique. Cependant, l'extension ne peut pas être réalisée directement avec $\mathcal{L}_{tac}$, puisque cela nécessite de pouvoir accéder aux informations caractérisant un type inductif[15], ce qui est trop *profond* pour le métalangage du toplevel. L'idée est donc de passer le code $\mathcal{L}_{tac}$ en Objective Caml, au moyen de la quotation `tactic`, et d'effectuer les quelques modifications nécessaires pour appeler les fonctions Objective Caml, dont on a besoin pour reconnaître les isomorphismes de connecteurs.

Nous n'allons pas donner le code complet de la version modifiée de `Tauto`, le lecteur intéressé pouvant aller consulter le fichier source `tactics/tauto.ml4`. À titre d'exemple, regardons comment la définition de tactique `Axioms` doit être changée pour accepter les isomorphismes de connecteurs :

```
let is_unit ist =
  if (is_unit_type (List.assoc 1 ist.lmatch)) then
    <:tactic<Idtac>>
  else
    failwith "is_unit"

let is_empty ist =
  if (is_empty_type (List.assoc 1 ist.lmatch)) then
    <:tactic<ElimType ?1;Assumption>>
  else
    failwith "is_empty"

let axioms ist =
  let t_is_unit = tacticIn is_unit
  and t_is_empty = tacticIn is_empty in
  <:tactic<
    Match Context With
    |[|- ?1] -> $t_is_unit;Constructor
```

[14]On appellera connecteurs (logiques), les constantes $\top$, $\bot$, $\wedge$ et $\vee$.

[15]Tous les connecteurs sont codés par des types inductifs.

```
19        |[_:?1 |- ?] -> $t_is_empty
20        |[_:?1 |- ?1] -> Auto>>
```

Des lignes 1 à 5, il s'agit de la fonction `is_unit`, qui vérifie que le type lié à la métavariable 1 est isomorphe à $\top$, et ne fait rien (`Idtac`) si c'est le cas (cela permet d'effectuer un traitement suite à cet appel), ou échoue (`failwith`) sinon. La vérification de l'isomorphisme de type est effectuée, en Objective Caml, par la fonction `is_unit_type` (voir fichiers `tactics/hipattern.ml[i]`). Sur un principe similaire, des lignes 7 à 11, on définit la fonction `is_empty`, qui vérifie que le type lié à la métavariable 1 (et supposé être en hypothèse) est isomorphe à $\perp$, et résoud si c'est le cas, ou, échoue sinon. De même, la vérification proprement dite est effectuée par la fonction annexe `is_empty_type` (définie aussi dans les fichiers `tactics/hipattern.ml[i]`). Enfin, ces deux fonctions permettent de définir la fonction `axioms`, des lignes 13 à 20. En particulier, en lignes 14 et 15, ces fonctions sont transformées en AST's dynamiques (tag `"tactic"`), pour être utilisées dans la partie $\mathcal{L}_{tac}$ qui suit. Dans le script, en ligne 18, on vérifie qu'on a un $\top$ en conclusion en appelant `is_unit`, et on applique alors son constructeur, avec `Constructor` (`Trivial` n'est, en effet, plus assez général). En ligne 19, on cherche un $\perp$ dans le contexte local, ce qui est directement réglé par `is_empty`. Quant au cas de la ligne 20, il est inchangé.

Le reste du code $\mathcal{L}_{tac}$ de `Tauto` a été modifié de manière similaire. Au passage, d'autres changements ont pu être effectués, afin de fixer des heuristiques satisfaisantes concernant le traitement des types dépendants et la δ-réduction (voir fichier `tactics/tauto.ml4`, pour plus de détails). La taille du code a augmenté quelque peu (150 lignes) par rapport au code en $\mathcal{L}_{tac}$ pur (28 lignes), mais reste complètement satisfaisante comparé à la taille du code original (2000 lignes). On conserve donc les avantages de la version $\mathcal{L}_{tac}$ initiale, puisque reposant essentiellement sur ce code, et, en particulier, aucune perte de performances n'a été observée.

La tactique `Field`

Dans le chapitre 8, on a pu s'apercevoir que si l'on désirait un fonctionnement de `Field` aussi flexible que celui de `Ring`, à savoir gérer une table des théories stockées par le type représentant l'ensemble support et avoir une tactique reconnaissant le type des membres de l'égalité pour aller chercher la théorie correspondante, il nous fallait passer en Objective Caml. Dans le cas de `Field`, ce *plongement* se justifie essentiellement par la nécessité de manipuler un objet mutable (la table des théories) et, contrairement à la tactique `Tauto`, il n'y a pas d'interactions complexes entre le code Objective Caml et la partie $\mathcal{L}_{tac}$, qui peut, de ce fait, rester à toplevel (voir fichier `contrib/field/Field_Tactic.v`). Toutefois, même dans le fichier Objective Caml, à savoir `contrib/field/field.ml4`, l'utilisation de $\mathcal{L}_{tac}$ s'avère pratique, notamment pour extraire le type des membres de l'égalité à résoudre ou pour gérer aussi les égalités dans `Set` :

```
let field g =
  let evc = project g
  and env = pf_env g in
  let ist = { evc=evc; env=env; lfun=[]; lmatch=[];
              goalopt=Some g; debug=get_debug () } in
  let typ = constr_of_Constr (interp_tacarg ist
    <:tactic<
      Match Context With
      | [|- (eq ?1 ? ?)] -> ?1
      | [|- (eqT ?1 ? ?)] -> ?1>>) in
```

```
11    let th = VArg (Constr (lookup typ)) in
12    (tac_interp [(id_of_string "FT",th)] [] (get_debug ())
13      <:tactic<
14        Match Context With
15        | [|- (eq ?1 ?2 ?3)] ->
16          Let t = (eqT ?1 ?2 ?3) In
17          Cut t;[Intro;
18            (Match Context With
19            | [id:t |- ?] -> Rewrite id;Reflexivity)|Field_Gen FT]
20        | [|- (eqT ? ? ?)] -> Field_Gen FT>>) g
```

C'est ici le code de la tactique principale, qui, en ligne 1, prend un but en argument. En lignes 2 et 3, on extrait le contexte du but, afin de construire un environnement d'interprétation (pour l'évaluateur de tactiques), en lignes 4 et 5. Ensuite, des lignes 6 à 10, il s'agit de reconnaître le type correspondant à l'ensemble support. On utilise, pour cela, des lignes 7 à 10, un script $\mathcal{L}_{tac}$, qui filtre l'égalité à résoudre (on peut remarquer, au passage, que l'on traite également les égalités dans `Set`[16]), puis, on interprète le résultat en ligne 6. En ligne 11, on recherche la théorie associée à l'ensemble support. Enfin, des lignes 12 à 20, on appelle la tactique générique `Field_Gen` (codée à toplevel dans `contrib/field/Field_Tactic.v`) avec la théorie précédemment récupérée. Là encore, on utilise une partie $\mathcal{L}_{tac}$, afin de faire un traitement spécial si l'égalité est dans `Set`. En effet, `Field` ne fonctionnant que dans `Type`, il ne filtre donc que des égalités dans `Type`, ainsi, des lignes 15 à 19, si on a une égalité dans `Set`, on la transforme, au préalable, en une égalité dans `Type`[17].

C'est la seule utilisation de $\mathcal{L}_{tac}$ dans le fichier `contrib/field/field.ml4`. Le reste concerne essentiellement la gestion de la table des théories et la création de la commande pour ajouter une théorie donnée.

9.2 Debugger de tactiques

9.2.1 Besoins

Avec l'émergence de $\mathcal{L}_{tac}$, c'est-à-dire d'une couche de métalangage disponible directement à toplevel, on peut se demander si les moyens *usuels* servant à débuguer les tactiques sont satisfaisants. Par moyens usuels, on entend typiquement deux types de méthodes. La première est superficielle et, de fait, assez limitée. Il s'agit de la commande Drop de Coq, utilisable uniquement avec une version compilée en bytecode, et que l'on a pu voir, à plusieurs reprises, dans la section 9.1. Cette commande lance un toplevel Objective Caml, parsé par Camlp4, et, une fois le fichier `include` (que l'on peut qualifier de fichier de configuration), on est dans un environnement très acceptable pour entreprendre, entre autres, des expériences de mise au point de tactiques. En particulier, tous les printers de Coq (termes, tactiques, ...) sont chargés et un certain nombre de modules du système sont ouverts[18]. On peut alors utiliser des directives du toplevel comme `trace`, qui permet de tracer l'exécution d'une fonction Objective Caml, en affichant ce qu'elle reçoit en arguments et ce qu'elle rend.

[16] Comme on l'a vu dans le chapitre 8, c'est une conséquence directe de la cumulativité. De manière similaire, on aurait aussi pu traiter la sorte `Prop`, mais cette sorte est la sorte des propositions, alors que l'ensemble support correspond à un type de données, qui, de fait, *habite* plutôt `Set` ou `Type`.

[17] C'est toujours possible, à nouveau grâce à la cumulativité. En effet, si on a `a=b`, avec `a` et `b` de type `Set`, alors on peut dire que `a` et `b` sont aussi de type `Type`, afin de pouvoir écrire que `a==b`.

[18] Les autres modules sont aussi accessibles, puisqu'ayant été linkés, mais tout ce qui les référencie doit être préfixé par le nom du module.

La difficulté consiste alors à constituer un ensemble pertinent de fonctions à tracer, sachant qu'il faut se munir impérativement du code du système et que, même avec les sources, il est parfois compliqué de se faire rapidement une idée sur la sémantique d'une fonction. La tâche peut donc s'avérer ardue si le bug est assez profond, et c'est pourquoi, on utilisera plutôt cette technique dans une optique de débugage rapide.

La deuxième méthode, qui s'offre à l'utilisateur souhaitant traquer un bug de sa tactique, est certes plus invasive, mais aussi plus efficace. Elle consiste à utiliser le debugger d'Objective Caml (`ocamldebug`), puisqu'il s'agit du langage d'implantation de Coq, et surtout, parce c'est le métalangage du système[19] (langage pour coder les tactiques). Pour ce faire, il faut se procurer les sources de Coq et les recompiler ensuite intégralement avec une option spéciale d'Objective Caml (option `-g`). Avec le `Makefile` de Coq, il suffit de taper `make "CAMLDEBUG=-g" world`. Enfin, on lance `ocamldebug` avec, comme argument, la version bytecode de Coq (le debugger ne marche, en effet, pas avec des programmes compilés et linkés en natif). Pour connaître les différentes commandes et fonctionnalités d'`ocamldebug`, on pourra consulter le manuel de référence d'Objective Caml [54].

Ces deux méthodes de débugage sont, d'une certaine manière, contraignantes, car elles nécessitent, au minimum, d'avoir les sources de Coq (potentiellement, même avec `Drop`), et éventuellement Objective Caml (si l'on souhaite utiliser `ocamldebug`). Dans l'optique de $\mathcal{L}_{tac}$, il est clair que l'utilisateur peut être sensiblement surpris de devoir *descendre* si profondément dans le système pour débuguer la tactique qu'il vient d'écrire à toplevel. Il est donc plutôt raisonnable de penser qu'une tactique codée à toplevel doit pouvoir aussi être débuguer à toplevel. C'est essentiellement ce qui a motivé l'élaboration d'un debugger spécifique à $\mathcal{L}_{tac}$, et nous allons décrire son fonctionnement.

9.2.2 Debugger à toplevel

Utilisation

Le debugger développé spécialement pour $\mathcal{L}_{tac}$ n'est, pour l'instant, pas très évolué et se situe plus dans un contexte expérimental, ce qui explique le panel encore assez primitif de commandes disponibles, ainsi que l'absence de documentation dans le manuel de référence de Coq. Le mode debug est activé et désactivé respectivement par les commandes suivantes :

```
Debug On.
Debug Off.
```

Toute expression de tactique est évaluée avec une information de debug (soit *actif*, soit *inactif*). Ainsi, si l'on utilise une définition de tactique (`Tactic Definition` ou `Meta Definition`), le mode debug doit être actif si l'on souhaite pouvoir *tracer* l'évaluation de cette définition lors d'un appel ultérieur. Par ce système, on se rapproche un peu du principe d'`ocamldebug`, où l'on doit compiler les fichiers avec une option spécifique, afin d'*enregistrer* des informations nécessaires au débugage. Par ailleurs, il est important de signaler qu'une définition interprétée avec un mode debug actif sera toujours tracée, même si le mode debug est inactif au moment de son appel. On a donc une sorte de *fermeture*, où c'est le mode de debug de définition qui est pris en compte et non celui d'appel.

Une fois entré dans le mode de debug, plusieurs informations apparaissent lorsqu'on évalue une expression de tactique :

[19] Maintenant, avec l'intégration de $\mathcal{L}_{tac}$, il faudrait dire *un* métalangage du système, mais il s'agit ici de décrire la situation avant, où il n'y avait que les tacticals à toplevel, et où la quasi-totalité des tactiques était codées en Objective Caml.

```
Welcome to Coq 7.1 (September 2001)

Coq < Debug On.

Coq < Goal (P:Prop)P->P.
1 subgoal

  ============================
   (P:Prop)P->P

Unnamed_thm < Intros;Assumption.
Goal:

  ============================
   (P:Prop)P->P

Going to execute:
Intros;
Assumption

TcDebug >
```

De manière minimale, le debugger affiche, s'il existe, le but (sous `Goal`) sur lequel la tactique va s'appliquer (on l'appellera *but courant d'évaluation*[20]), ainsi que l'expression de tactique (sous `Going to execute`) qui va être interprétée (elle sera appelée *évaluation courante*). D'autres informations peuvent éventuellement compléter cet état, suivant l'expression à exécuter. En particulier, si on évalue un `Match Context`, les hypothèses et le goal filtrés sont également affichés, et s'il s'agit d'un terme, le terme interprété est ensuite retourné[21]. Nous verrons un exemple de ces exécutions spécifiques dans la section 9.2.3. Pour savoir quelles sont les commandes du debugger, il suffit de taper `h` après le prompt (`TcDebug >`) :

```
TcDebug > h
Commands: <Enter>=Continue, h=Help, s=Skip, x=Exit

TcDebug >
```

Comme on l'a dit précédemment, ce debugger est encore expérimental et les commandes sont un peu rudimentaires. Toutefois, on peut, d'ores et déjà, effectuer quelques opérations élémentaires, ce qui donne à cet outil une base relativement fonctionnelle. `<Enter>` permet d'accepter l'évaluation courante (sous `Going to execute`) et sera généralement la commande la plus utilisée. `h` affiche le menu des commandes comme nous venons de le faire. `s` permet de *passer* l'évaluation courante. Ainsi, l'expression à évaluer sera bien exécutée, mais sans aucune trace. Enfin, `x` sort de l'évaluation courante, en ce sens qu'il n'exécute rien et laisse le but d'évaluation courante inchangé. Cette commande doit être utilisée pour *quitter* l'évaluation d'une expression destinée à être une tactique et uniquement dans ce cas (sinon, le comportement du debugger n'est pas spécifié).

Ici, dans notre exemple, si l'on tape `<Enter>`, on obtiendra le résultat suivant :

[20] À distinguer du *but courant*, qui se définit par rapport au mode d'édition de preuves.

[21] C'est très utile si le terme contient des arguments implicites ou des métavariables.

```
TcDebug >
Goal:

  ============================
   (P:Prop)P->P

Going to execute:
Intros

TcDebug >
```

Le goal est inchangé, puisque, pour évaluer une séquence (`;`), il faut d'abord le premier membre de la séquence (à savoir `Intros`).

Par contre, si l'on utilise `s`, on aura :

```
TcDebug > s
Subtree proved!

Unnamed_thm <
```

On ne détaille pas l'exécution et le but courant d'évaluation est résolu (cela termine aussi la preuve, car il n'y avait qu'un seul but, qui, de plus, correspondait au but courant d'évaluation).

Enfin, avec `x`, le debugger se comportera comma suit :

```
TcDebug > x
1 subgoal

  ============================
   (P:Prop)P->P

Unnamed_thm <
```

L'évaluation n'est pas effectuée et le but courant d'évaluation est inchangé.

Implantation

Avant de voir un exemple d'utilisation, décrivons brièvement l'implantation de ce debugger pour $\mathcal{L}_{tac}$. Les fichiers concernés sont `proofs/tactic_debug.ml[i]` et `proofs/tacinterp.ml[i]` (nous utilisons toujours le même système de référencement que dans les sections 9.1.3 et 9.1.4).

Une remarque importante, concernant le code, est qu'il a été réalisé de manière complètement fonctionnelle, en ce sens que chaque expression évaluée *transmet* l'information de débugage aux sous-expressions devant être interprétées. Par ailleurs, comme on l'a dit précédemment, on utilise aussi une sorte de système de fermeture et, de ce fait, l'information pertinente concerne le mode de debug au moment de la définition, et non au moment de l'évaluation. Ceci a pour avantage de rendre la situation plus statique, bien que le mode de debug soit un objet mutable. Toutefois, cette évaluation fonctionnelle avec fermetures doit être correctement appréhendée par l'utilisateur, car, en effet, une évaluation peut donner plusieurs branches avec des modes de debug différents, qui dépendent du contexte de de définition, et qui peuvent avoir été modifiés par le debugger (avec la commande `s`). Ainsi, en utilisant des tactiques primitives et les tacticals, il est probable que ce fonctionnement spécifique de débugage puisse être ignoré, mais ce ne sera pas le cas avec les extensions apportées par $\mathcal{L}_{tac}$, pour lesquelles il devra être maîtrisé, sous peine d'avoir des surprises apparemment contre-intuitives.

9.2.3 Un exemple

Comme exemple non trivial, nous avons choisi d'écrire une tactique qui *sature* le contexte local en égalités symétriques. Ainsi, si on a un séquent de la forme $a = b, a = c, c = d \vdash P$, on souhaite rendre $a = b, a = c, c = d, b = a, c = a, d = c \vdash P$. Par ailleurs, on ne veut pas symétriser une égalité dont l'égalité symétrique appartient déjà au contexte local. Pour ce faire, on propose le code $\mathcal{L}_{tac}$ suivant :

```
1   Tactic Definition EqExists t1 t2 :=
2     Match Context With
3     | [_:t1=t2 |- ?] -> Idtac.
4
5   Tactic Definition SymEq :=
6     Repeat
7       (Match Context With
8        | [_:?1=?2 |- ?] ->
9          (EqExists ?1 ?2) Orelse (Cut ?2=?1;[Intro|Auto])).
```

On a deux définitions de tactiques. D'abord, `EqExists`, de la ligne 1 à la ligne 3, qui, étant deux termes (`t1` et `t2`), vérifie si l'égalité de ces deux termes (`t1=t2`) existe déjà dans le contexte local. Puis, `SymEq`, de la ligne 5 à la ligne 9, qui filtre une égalité du contexte local, vérifie si l'égalité symétrique n'existe pas déjà (avec `EqExists`), échoue (membre droit du `Orelse`) si c'est le cas (on va alors chercher la prochaine égalité), ou introduit l'égalité symétrique (membre gauche du `Orelse`) sinon. On réitère le processus (avec `Repeat`) jusqu'à ce que les égalités aient toutes été symétrisées.

Exécutons cette nouvelle tactique pour voir si elle fonctionne (on suppose que les définitions précédentes ont déjà été évaluées dans le toplevel) :

```
Coq < Parameters A:Set;a,b,c,d:A;P:Prop.
A is assumed
a is assumed
b is assumed
c is assumed
d is assumed
P is assumed

Coq < Goal a=b->a=c->c=d->P.
1 subgoal

  ============================
   a=b->a=c->c=d->P

Unnamed_thm < Intros.
1 subgoal

  H : a=b
```

```
  H0 : a=c
  H1 : c=d
  ============================
   P

Unnamed_thm < SymEq.

User Interrupt.

Unnamed_thm <
```

On constate que, non seulement, la tactique `SymEq` ne fonctionne pas, mais que, de surcroît, elle boucle. C'est typiquement un comportement, qui empêche un débugage rapide et manuel. On va donc utiliser le debugger à toplevel pour voir ce qui se passe. Tout d'abord, on active le debugger et on évalue les définitions relatives à `SymEq` :

```
Welcome to Coq 7.1 (September 2001)

Coq < Debug On.

Coq < Tactic Definition EqExists t1 t2 :=
Coq <   Match Context With
Coq <   | [_:t1=t2 |- ?] -> Idtac.
No goal
Going to execute:
Fun t1 t2 -> Match Context With
             | [ _ : t1=t2 |- ? ] -> Idtac

TcDebug >
EqExists is defined

Coq < Tactic Definition SymEq :=
Coq <   Repeat
Coq <     (Match Context With
Coq <      | [_:?1=?2 |- ?] ->
Coq <        (EqExists ?1 ?2) Orelse (Cut ?2=?1;[Intro|Auto])).
No goal
Going to execute:
Repeat (Match Context With
        | [ _ : ?1=?2 |- ? ] ->
          (EqExists ?1 ?2) Orelse (Cut ?2=?1; [ Intro | Auto ]))

TcDebug >
SymEq is defined
```

On peut remarquer que les deux définitions nous font *entrer* dans le debugger, puisqu'il a été précédemment activé (`Debug On`), et que la sémantique d'évaluation d'une définition est d'évaluer le corps de la définition avant d'enrichir l'environnement avec le nom de la définition, associé avec la valeur obtenue. En particulier, avec `EqExist`, on observe qu'une définition avec des arguments s'évalue bien comme une fonction. Enfin, notons que, comme ces évaluations se font en dehors du mode d'édition de preuves, le debugger n'affiche aucun but courant d'évaluation (`No goal`).

Nous pouvons maintenant tester la tactique `SymEq` sur le but précédent (on considère que les paramètres ont déjà été évalués) :

```
Coq < Goal a=b->a=c->c=d->P.
1 subgoal

  ============================
   a=b->a=c->c=d->P

Unnamed_thm < Intros.
Goal:

  ============================
   a=b->a=c->c=d->P

Going to execute:
Intros

TcDebug >
1 subgoal

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Unnamed_thm < SymEq.
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
SymEq

TcDebug >
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

```

```
Going to execute:
Match Context With
| [ _ : ?1=?2 |- ? ] ->
  (EqExists ?1 ?2) Orelse (Cut ?2=?1; [ Intro | Auto ])

TcDebug >
Matched goal --> P
Matched hypothesis --> H: a=b
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
(EqExists ?1 ?2) Orelse (Cut ?2=?1; [ Intro | Auto ])

TcDebug >
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
EqExists ?1
?2

TcDebug >
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
EqExists

TcDebug >
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
```

```
  ============================
   P

Going to execute:
?1

TcDebug >
Evaluated term --> a
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
?2

TcDebug >
Evaluated term --> b
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
Match Context With
| [ _ : t1=t2 |- ? ] -> Idtac

TcDebug >
Matched goal --> P
Matched hypothesis --> H: a=b
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
Idtac

TcDebug >
```

Avant de regarder où se situe le bug, nous pouvons remarquer que certaines évaluations affichent des informations en plus du but courant d'évaluation et de l'évaluation courante. Il s'agit des expressions `Match Context`, qui donnent les hypothèses et le but filtrés, comme en lignes 46 à 53 ou 125 à 131, et des termes, qui sont réaffichés après évaluation, comme en lignes 99 à 103 ou 112 à 116.

Ces informations supplémentaires vont nous être d'une aide précieuse. En effet, tout d'abord, on s'aperçoit, en ligne 53, que l'on travaille sur l'hypothèse de type `a=b`. Ensuite, en lignes 74 à 76, on fait bien l'appel à `(EqExists ?1 ?2)`, avec les métavariables `?1` et `?2`, respectivement liées à `a` et `b`, ce que confirment les lignes 99 à 103 et 112 à 116. Ainsi, de fait, lors de l'appel à `EqExists`, `t1` et `t2` se retrouve aussi liées respectivement à `a` et `b`. Ceci est confirmé, en lignes 125 à 131, par le `Match Context` de `EqExists`, qui filtre à nouveau l'hypothèse de type `a=b`, et non l'hypothèse symétrique. Le problème est donc que les paramètres effectifs de `EqExists` sont dans le mauvais ordre.

Le bug est donc résolu en inversant les paramètres d'appel de `EqExists` dans `EqSym`. Voici donc une nouvelle version de `EqSym` :

```
Tactic Definition SymEq :=
  Repeat
    (Match Context With
     | [_:?1=?2 |- ?] ->
       (EqExists ?2 ?1) Orelse (Cut ?2=?1;[Intro|Auto])).
```

Essayons à nouveau `SymEq`, sans debugger (on considère que les définitions de tactiques et les paramètres ont été évalués) :

```
Coq < Goal a=b->a=c->c=d->P.
1 subgoal

  ============================
   a=b->a=c->c=d->P

Unnamed_thm < Intros.
1 subgoal

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Unnamed_thm < SymEq.

User Interrupt.

Unnamed_thm <
```

Malheureusement, on s'aperçoit que le bug précédent n'était pas le seul et la tactique boucle toujours. On réactive donc le debugger et on retrace l'évaluation de `SymEq` (on reprendra la trace précédente à partie de l'évaluation du `Match Context` de `EqExists`) :

```
1   Goal:
```

```

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
Match Context With
| [ _ : t1=t2 |- ? ] -> Idtac

TcDebug >
Matched goal --> P
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Going to execute:
Cut ?2=?1;
[ Intro | Auto ]

TcDebug > s
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  ============================
   P

Going to execute:
Match Context With
| [ _ : ?1=?2 |- ? ] ->
  (EqExists ?2 ?1) Orelse (Cut ?2=?1; [ Intro | Auto ])

TcDebug >
Matched goal --> P
Matched hypothesis --> H: a=b
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  ============================
```

```
   P

Going to execute:
(EqExists ?2 ?1) Orelse (Cut ?2=?1; [ Intro | Auto ])

TcDebug >
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  ============================
   P

Going to execute:
EqExists ?2
?1

TcDebug >

[...]

Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  ============================
   P

Going to execute:
Match Context With
| [ _ : t1=t2 |- ? ] -> Idtac

TcDebug >
Matched goal --> P
Matched hypothesis --> H2: b=a
Goal:

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  ============================
   P

Going to execute:
Idtac
```

```
102
103   TcDebug >
104   Goal:
105
106     H : a=b
107     H0 : a=c
108     H1 : c=d
109     H2 : b=a
110     ============================
111      P
112
113   Going to execute:
114   Cut ?2=?1;
115   [ Intro | Auto ]
116
117   TcDebug >
```

En lignes 10 à 11, on essaie de trouver l'hypothèse symétrique dans le contexte local, à savoir `b=a`. En ligne 14, on s'aperçoit qu'on arrive à filtrer la conclusion du but courant d'évaluation, mais pas d'hypothèses. Le `Match Context` de `EqExists` échoue donc et on doit évaluer le membre droit du `Orelse` de `EqSym`. Cette évaluation introduit l'égalité symétrique et on ne souhaite pas la détailler (car elle ne fait clairement pas boucler et sa sémantique est facilement vérifiable avec le but courant d'évaluation produit), d'où l'utilisation de la commande `s`, en ligne 27. Le `Repeat` de `EqSym` réitère le processus, puisque la symétrisation a été réussie. On refiltre l'hypothèse `a=b`, en ligne 44, et on doit rappeler `EqExists`, en lignes 68 et 69. Cette fois-ci, il filtre l'égalité symétrique précédemment introduite, en ligne 90, et peut, en ligne 101, exécuter le membre droit de la règle correspondante. Toutefois, on s'aperçoit, en ligne 114 et 115, que l'on va tout de même évaluer le membre droit du `Orelse`, introduisant l'égalité symétrique, à savoir `b=a` à nouveau.

Le problème est donc que l'on n'échoue pas correctement lorsque l'égalité symétrique existe. En effet, si l'égalité n'existe pas, on échoue (`Fail`) et le `Orelse` rattrape l'erreur pour exécuter le membre droit, tandis que si l'égalité existe, on ne fait rien (`Idtac`) et le `Orelse` exécute tout de même le membre droit, puisqu'il n'y a pas eu *progression* (le but est inchangé). Dans le cas où l'égalité existe, il faut faire échouer le `Orelse`. On pourrait utiliser :

```
((EqExists ?2 ?1);Fail) Orelse (Cut ?2=?1;[Intro|Auto])).
```

Toutefois, ce n'est pas suffisant, car le `Fail` serait encore rattrapé par le `Orelse`. Une solution est donc d'échouer *plus fortement*, en augmentant le niveau d'échec, avec, par exemple, `Fail 1`. On peut alors donner une nouvelle version de `SymEq` :

```
Tactic Definition SymEq :=
  Repeat
    (Match Context With
     | [_:?1=?2 |- ?] ->
       ((EqExists ?2 ?1);Fail 1) Orelse (Cut ?2=?1;[Intro|Auto])).
```

Voyons, toujours sur le même exemple et sans debugger, si la correction est suffisante (on suppose que définitions et paramètres ont été évalués) :

```
Coq < Goal a=b->a=c->c=d->P.
1 subgoal

  ============================
   a=b->a=c->c=d->P

Unnamed_thm < Intros.
1 subgoal

  H : a=b
  H0 : a=c
  H1 : c=d
  ============================
   P

Unnamed_thm < SymEq.
1 subgoal

  H : a=b
  H0 : a=c
  H1 : c=d
  H2 : b=a
  H3 : c=a
  H4 : d=c
  ============================
   P

Unnamed_thm <
```

On remarque que la tactique ne boucle plus et que, de plus, elle symétrise bien toutes les égalités du contexte local.

9.2.4 Discussion

Le debugger, que nous venons de décrire, même s'il est encore expérimental, s'avère être tout à fait approprié pour la mise au point de tactiques codées directement à toplevel (en $\mathcal{L}_{tac}$). En effet, étant aussi à toplevel, il permet de se passer complètement de toute connaissance avancée ou non de l'implantation du système `Coq`, ce qui n'était pas le cas dans deux méthodes de débugage (avec `Drop` et `ocamldebug`) présentées dans la section 9.2.1, et qui, comme on l'a dit précédemment, semble plutôt légitime dans l'optique de tactiques utilisant $\mathcal{L}_{tac}$.

Une autre caractéristique intéressante de ce debugger est qu'il est également viable en code natif. En effet, contrairement aux méthodes utilisant `Drop` et `ocamldebug`, on peut parfaitement débuguer avec une version native du toplevel. Même si elle ne faisait pas partie des motivations initiales, cette particularité peut être considérée comme un bonus significatif.

Annexes

Annexe A

Preuves complètes

A.1 PVS

Ces preuves ont été executées sous PVS 2.3 avec XEmacs 21.1 sous Linux RedHat 5.2 sur un Intel Pentium III à 450 MHz.

A.1.1 Preuve expansée

```
eq_nat_dec :

  |-------
{1}   FORALL (m, n: nat): (n = m) OR (NOT (n = m))

Rerunning step: (INDUCT "m")
Inducting on m on formula 1,
this yields  2 subgoals:
eq_nat_dec.1 :

  |-------
{1}   FORALL (n: nat): (n = 0) OR (NOT (n = 0))

Rerunning step: (INDUCT "n")
Inducting on n on formula 1,
this yields  2 subgoals:
eq_nat_dec.1.1 :

  |-------
{1}   (0 = 0) OR (NOT (0 = 0))

Rerunning step: (FLATTEN-DISJUNCT)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.1.1 :

{-1}  (0 = 0)
  |-------
```

```
{1}   (0 = 0)

which is trivially true.

This completes the proof of eq_nat_dec.1.1.

eq_nat_dec.1.2 :

  |-------
{1}   FORALL j:
        (j = 0) OR (NOT (j = 0)) IMPLIES (j + 1 = 0) OR (NOT (j + 1 = 0))

Rerunning step: (SKOLEM!)
Skolemizing,
this simplifies to:
eq_nat_dec.1.2 :

  |-------
{1}   (j!1 = 0) OR (NOT (j!1 = 0)) IMPLIES
       (j!1 + 1 = 0) OR (NOT (j!1 + 1 = 0))

Rerunning step: (FLATTEN-DISJUNCT)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.1.2 :

{-1}  (j!1 = 0) OR (NOT (j!1 = 0))
{-2}  (j!1 + 1 = 0)
  |-------
{1}   (j!1 + 1 = 0)

which is trivially true.

This completes the proof of eq_nat_dec.1.2.


This completes the proof of eq_nat_dec.1.

eq_nat_dec.2 :

  |-------
{1}   FORALL j:
        (FORALL (n: nat): (n = j) OR (NOT (n = j))) IMPLIES
         (FORALL (n: nat): (n = j + 1) OR (NOT (n = j + 1)))

Rerunning step: (SKOLEM!)
Skolemizing,
this simplifies to:
eq_nat_dec.2 :
```

```
  |-------
{1}   (FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))) IMPLIES
       (FORALL (n: nat): (n = j!1 + 1) OR (NOT (n = j!1 + 1)))

Rerunning step: (FLATTEN)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2 :

{-1}  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   FORALL (n: nat): (n = j!1 + 1) OR (NOT (n = j!1 + 1))

Rerunning step: (INDUCT "n")
Inducting on n on formula 1,
this yields  2 subgoals:
eq_nat_dec.2.1 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (0 = j!1 + 1) OR (NOT (0 = j!1 + 1))

Rerunning step: (FLATTEN-DISJUNCT)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.1 :

{-1}  (0 = j!1 + 1)
[-2]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (0 = j!1 + 1)

which is trivially true.

This completes the proof of eq_nat_dec.2.1.

eq_nat_dec.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   FORALL j:
        (j = j!1 + 1) OR (NOT (j = j!1 + 1)) IMPLIES
         (j + 1 = j!1 + 1) OR (NOT (j + 1 = j!1 + 1))

Rerunning step: (SKOLEM!)
Skolemizing,
this simplifies to:
eq_nat_dec.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
```

```
  |-------
{1}   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Rerunning step: (CASE "j!2 = j!1")
Case splitting on
   j!2 = j!1,
this yields  2 subgoals:
eq_nat_dec.2.2.1 :

{-1}  j!2 = j!1
[-2]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
[1]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Rerunning step: (FLATTEN-DISJUNCT)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.2.1 :

[-1]  j!2 = j!1
{-2}  (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1))
{-3}  (j!2 + 1 = j!1 + 1)
[-4]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (j!2 + 1 = j!1 + 1)

which is trivially true.

This completes the proof of eq_nat_dec.2.2.1.

eq_nat_dec.2.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   j!2 = j!1
[2]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Rerunning step: (FLATTEN-DISJUNCT)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.2.2 :

{-1}  (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1))
{-2}  (j!2 + 1 = j!1 + 1)
[-3]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
[1]   j!2 = j!1
```

```
{2}   (j!2 + 1 = j!1 + 1)

which is trivially true.

This completes the proof of eq_nat_dec.2.2.2.


This completes the proof of eq_nat_dec.2.2.


This completes the proof of eq_nat_dec.2.

Q.E.D.


Run time  = 0.85 secs.
Real time = 4.15 secs.
```

A.1.2 Preuve avec stratégies

```
eq_nat_dec :

  |-------
{1}   FORALL (m, n: nat): (n = m) OR (NOT (n = m))

Rule? (spread! (induct "m")
  ((spread! (induct "n")
    ((flatten-disjunct)
     (then (skolem!) (flatten-disjunct))))
   (spread! (then (skolem!) (flatten) (induct "n"))
    ((flatten-disjunct)
     (spread! (then (skolem!) (case "j!2 = j!1"))
       ((flatten-disjunct)
        (flatten-disjunct)))))))
Inducting on m on formula 1,
this yields  2 subgoals:
eq_nat_dec.1 :

  |-------
{1}   FORALL (n: nat): (n = 0) OR (NOT (n = 0))

Inducting on n on formula 1,
this yields  2 subgoals:
eq_nat_dec.1.1 :

  |-------
{1}   (0 = 0) OR (NOT (0 = 0))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
```

```
eq_nat_dec.1.1 :

{-1}  (0 = 0)
  |-------
{1}   (0 = 0)

which is trivially true.

This completes the proof of eq_nat_dec.1.1.

eq_nat_dec.1.2 :

  |-------
{1}   FORALL j:
        (j = 0) OR (NOT (j = 0)) IMPLIES (j + 1 = 0) OR (NOT (j + 1 = 0))

Skolemizing,
this simplifies to:
eq_nat_dec.1.2 :

  |-------
{1}   (j!1 = 0) OR (NOT (j!1 = 0)) IMPLIES
       (j!1 + 1 = 0) OR (NOT (j!1 + 1 = 0))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.1.2 :

{-1}  (j!1 = 0) OR (NOT (j!1 = 0))
{-2}  (j!1 + 1 = 0)
  |-------
{1}   (j!1 + 1 = 0)

which is trivially true.

This completes the proof of eq_nat_dec.1.2.


This completes the proof of eq_nat_dec.1.

eq_nat_dec.2 :

  |-------
{1}   FORALL j:
        (FORALL (n: nat): (n = j) OR (NOT (n = j))) IMPLIES
         (FORALL (n: nat): (n = j + 1) OR (NOT (n = j + 1)))

Skolemizing,
this simplifies to:
eq_nat_dec.2 :
```

```
  |-------
{1}   (FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))) IMPLIES
       (FORALL (n: nat): (n = j!1 + 1) OR (NOT (n = j!1 + 1)))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2 :

{-1}  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   FORALL (n: nat): (n = j!1 + 1) OR (NOT (n = j!1 + 1))

Inducting on n on formula 1,
this yields  2 subgoals:
eq_nat_dec.2.1 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (0 = j!1 + 1) OR (NOT (0 = j!1 + 1))

Postponing eq_nat_dec.2.1.

eq_nat_dec.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   FORALL j:
        (j = j!1 + 1) OR (NOT (j = j!1 + 1)) IMPLIES
         (j + 1 = j!1 + 1) OR (NOT (j + 1 = j!1 + 1))

Postponing eq_nat_dec.2.2.

eq_nat_dec.2.1 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (0 = j!1 + 1) OR (NOT (0 = j!1 + 1))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.1 :

{-1}  (0 = j!1 + 1)
[-2]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (0 = j!1 + 1)

which is trivially true.
```

```
This completes the proof of eq_nat_dec.2.1.

eq_nat_dec.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   FORALL j:
        (j = j!1 + 1) OR (NOT (j = j!1 + 1)) IMPLIES
         (j + 1 = j!1 + 1) OR (NOT (j + 1 = j!1 + 1))

Skolemizing,
this simplifies to:
eq_nat_dec.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Case splitting on
   j!2 = j!1,
this yields  2 subgoals:
eq_nat_dec.2.2.1 :

{-1}  j!2 = j!1
[-2]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
[1]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Postponing eq_nat_dec.2.2.1.

eq_nat_dec.2.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   j!2 = j!1
[2]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Postponing eq_nat_dec.2.2.2.

eq_nat_dec.2.2.1 :

{-1}  j!2 = j!1
[-2]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
[1]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))
```

```
Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.2.1 :

[-1]  j!2 = j!1
{-2}  (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1))
{-3}  (j!2 + 1 = j!1 + 1)
[-4]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   (j!2 + 1 = j!1 + 1)

which is trivially true.

This completes the proof of eq_nat_dec.2.2.1.

eq_nat_dec.2.2.2 :

[-1]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
{1}   j!2 = j!1
[2]   (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1)) IMPLIES
       (j!2 + 1 = j!1 + 1) OR (NOT (j!2 + 1 = j!1 + 1))

Applying disjunctive simplification to flatten sequent,
this simplifies to:
eq_nat_dec.2.2.2 :

{-1}  (j!2 = j!1 + 1) OR (NOT (j!2 = j!1 + 1))
{-2}  (j!2 + 1 = j!1 + 1)
[-3]  FORALL (n: nat): (n = j!1) OR (NOT (n = j!1))
  |-------
[1]   j!2 = j!1
{2}   (j!2 + 1 = j!1 + 1)

which is trivially true.

This completes the proof of eq_nat_dec.2.2.2.


This completes the proof of eq_nat_dec.2.2.


This completes the proof of eq_nat_dec.2.

Q.E.D.


Run time  = 0.83 secs.
Real time = 10.65 secs.
```

A.2 HOL

Ces preuves ont été executées sous HOL Light 1.0, avec Caml Light 0.74 sous Digital Unix DEC OSF/1 4.0C sur une station Digital Alpha PWS 500 (500 Mhz).

A.2.1 Preuve complète

```
#g `!(n:num) (m:num). (n=m) \/ ~(n=m)`;;
it : goalstack = 1 subgoal (1 total)
`!n m. (n = m) \/ ~(n = m)`


#e INDUCT_TAC;;
it : goalstack = 2 subgoals (2 total)
`!m. (SUC n = m) \/ ~(SUC n = m)`

 0 [`!m. (n = m) \/ ~(n = m)`]
`!m. (0 = m) \/ ~(0 = m)`


#e INDUCT_TAC;;
it : goalstack = 2 subgoals (3 total)
`(0 = SUC m) \/ ~(0 = SUC m)`

 0 [`(0 = m) \/ ~(0 = m)`]
`(0 = 0) \/ ~(0 = 0)`


#e DISJ1_TAC;;
it : goalstack = 1 subgoal (3 total)
`0 = 0`


#e ARITH_TAC;;
it : goalstack = 1 subgoal (2 total)
`(0 = SUC m) \/ ~(0 = SUC m)`

 0 [`(0 = m) \/ ~(0 = m)`]

#e DISJ2_TAC;;
it : goalstack = 1 subgoal (2 total)
`~(0 = SUC m)`

 0 [`(0 = m) \/ ~(0 = m)`]

#e ARITH_TAC;;
it : goalstack = 1 subgoal (1 total)
`!m. (SUC n = m) \/ ~(SUC n = m)`

 0 [`!m. (n = m) \/ ~(n = m)`]
```

```
#e INDUCT_TAC;;
it : goalstack = 2 subgoals (2 total)
`(SUC n = SUC m) \/ ~(SUC n = SUC m)`

 0 [`(SUC n = m) \/ ~(SUC n = m)`]
 1 [`!m. (n = m) \/ ~(n = m)`]
`(SUC n = 0) \/ ~(SUC n = 0)`

 0 [`!m. (n = m) \/ ~(n = m)`]

#e DISJ2_TAC;;
it : goalstack = 1 subgoal (2 total)
`~(SUC n = 0)`

 0 [`!m. (n = m) \/ ~(n = m)`]

#e ARITH_TAC;;
it : goalstack = 1 subgoal (1 total)
`(SUC n = SUC m) \/ ~(SUC n = SUC m)`

 0 [`(SUC n = m) \/ ~(SUC n = m)`]
 1 [`!m. (n = m) \/ ~(n = m)`]

#e (FIRST_ASSUM (fun thm -> DISJ_CASES_TAC (SPEC `m:num` thm)));;
it : goalstack = 2 subgoals (2 total)
`(SUC n = SUC m) \/ ~(SUC n = SUC m)`

 0 [`~(n = m)`]
 1 [`(SUC n = m) \/ ~(SUC n = m)`]
 2 [`!m. (n = m) \/ ~(n = m)`]
`(SUC n = SUC m) \/ ~(SUC n = SUC m)`

 0 [`n = m`]
 1 [`(SUC n = m) \/ ~(SUC n = m)`]
 2 [`!m. (n = m) \/ ~(n = m)`]

#e DISJ1_TAC;;
it : goalstack = 1 subgoal (2 total)
`SUC n = SUC m`

 0 [`n = m`]
 1 [`(SUC n = m) \/ ~(SUC n = m)`]
 2 [`!m. (n = m) \/ ~(n = m)`]

#e (ASM_REWRITE_TAC []);;
it : goalstack = 1 subgoal (1 total)
`(SUC n = SUC m) \/ ~(SUC n = SUC m)`

 0 [`~(n = m)`]
```

```
 1 [‘(SUC n = m) \/ ~(SUC n = m)‘]
 2 [‘!m. (n = m) \/ ~(n = m)‘]

#e DISJ2_TAC;;
it : goalstack = 1 subgoal (1 total)
‘~(SUC n = SUC m)‘

 0 [‘~(n = m)‘]
 1 [‘(SUC n = m) \/ ~(SUC n = m)‘]
 2 [‘!m. (n = m) \/ ~(n = m)‘]

#e (REWRITE_TAC [SUC_INJ]);;
it : goalstack = 1 subgoal (1 total)
‘~(n = m)‘

 0 [‘~(n = m)‘]
 1 [‘(SUC n = m) \/ ~(SUC n = m)‘]
 2 [‘!m. (n = m) \/ ~(n = m)‘]

#e (FIRST_ASSUM ACCEPT_TAC);;
it : goalstack = No subgoals
```

A.2.2 Preuve avec tacticals

```
#let eq_nat_dec = time prove
  (‘!(n:num) (m:num). (n=m) \/ ~(n=m)‘,
   INDUCT_TAC THENL
     [INDUCT_TAC THENL [DISJ1_TAC THEN ARITH_TAC;DISJ2_TAC THEN ARITH_TAC];
      INDUCT_TAC THENL
        [DISJ2_TAC THEN ARITH_TAC;
         (FIRST_ASSUM (fun thm -> DISJ_CASES_TAC (SPEC ‘m:num‘ thm))) THENL
           [DISJ1_TAC THEN (ASM_REWRITE_TAC []);
            DISJ2_TAC THEN (REWRITE_TAC [SUC_INJ]) THEN
              (FIRST_ASSUM ACCEPT_TAC)]]]);;
CPU time (user): 0.0
eq_nat_dec : thm = |- !n m. (n = m) \/ ~(n = m)
```

A.3 Coq

Ces preuves ont été executées sous Coq 7.0, version native (`coqtop.opt`), sous Linux Mandrake 7.2 sur un Intel Pentium III à 1 GHz.

A.3.1 Preuve expansée

```
Welcome to Coq 7.0 (April 2001)

Coq < Lemma eq_nat:(n,m:nat)n=m\/~n=m.
1 subgoal
```

```
  ============================
   (n,m:nat)n=m\/~n=m

eq_nat < Proof.

eq_nat < Induction n.
2 subgoals

  n : nat
  ============================
   (m:nat)O=m\/~O=m

subgoal 2 is:
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Intro.
2 subgoals

  n : nat
  m : nat
  ============================
   O=m\/~O=m

subgoal 2 is:
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Case m.
3 subgoals

  n : nat
  m : nat
  ============================
   O=O\/~O=O

subgoal 2 is:
 (n0:nat)O=(S n0)\/~O=(S n0)
subgoal 3 is:
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Left.
3 subgoals

  n : nat
  m : nat
  ============================
   O=O

subgoal 2 is:
 (n0:nat)O=(S n0)\/~O=(S n0)
subgoal 3 is:
```

```
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Auto.
2 subgoals

  n : nat
  m : nat
  ============================
   (n0:nat)O=(S n0)\/~O=(S n0)

subgoal 2 is:
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Right.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  ============================
   ~O=(S n0)

subgoal 2 is:
 (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Auto.
1 subgoal

  n : nat
  ============================
   (n0:nat)((m:nat)n0=m\/~n0=m)->(m:nat)(S n0)=m\/~(S n0)=m

eq_nat < Intros.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  ============================
   (S n0)=m\/~(S n0)=m

eq_nat < Case m.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  ============================
```

```
   (S n0)=O\/~(S n0)=O

subgoal 2 is:
 (n1:nat)(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Right.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  ============================
   ~(S n0)=O

subgoal 2 is:
 (n1:nat)(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Auto.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  ============================
   (n1:nat)(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Intro.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  n1 : nat
  ============================
   (S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Case (H n1).
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  n1 : nat
  ============================
   n0=n1->(S n0)=(S n1)\/~(S n0)=(S n1)
```

```
subgoal 2 is:
 ~n0=n1->(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Left.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  n1 : nat
  H0 : n0=n1
  ============================
   (S n0)=(S n1)

subgoal 2 is:
 ~n0=n1->(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Auto.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  n1 : nat
  ============================
   ~n0=n1->(S n0)=(S n1)\/~(S n0)=(S n1)

eq_nat < Right.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)n0=m\/~n0=m
  m : nat
  n1 : nat
  H0 : ~n0=n1
  ============================
   ~(S n0)=(S n1)

eq_nat < Auto.
Subtree proved!

eq_nat < Save.
Induction n.
Intro.
Case m.
Left.
Auto.
```

```
Right.
Auto.

Intros.
Case m.
Right.
Auto.

Intro.
Case (H n1).
Left.
Auto.

Right.
Auto.

eq_nat is defined
```

A.3.2 Preuve avec tacticals

```
Welcome to Coq 7.0 (April 2001)

Coq < Lemma eq_nat:(n,m:nat)n=m\/~n=m.
1 subgoal

  ============================
   (n,m:nat)n=m\/~n=m

eq_nat < Proof.

eq_nat < Time Induction n;
           [Intro;Case m;[Left;Auto|Right;Auto]|
            Intros;Case m;
              [Right;Auto|
               Intro;Case (H n1);[Left;Auto|Right;Auto]]].
Subtree proved!
Finished transaction in 0 secs (0.01u,0s)

eq_nat < Save.
Induction n;
[ Intro; Case m; [ Left; Auto | Right; Auto ]
| Intros; Case m;
  [ Right; Auto | Intro; Case (H n1); [ Left; Auto | Right; Auto ] ] ].

eq_nat is defined
```

A.4 ACL2

Ces preuves ont été executées sous ACL2 2.5, utilisant GNU-Common Lisp 2.3, sous Linux Mandrake 7.2 sur un Intel Pentium III à 1 GHz.

A.4.1 Preuve à toplevel

```
GCL (GNU Common Lisp)  Version(2.3) Wed Jun 13 16:26:08 CEST 2001
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter

 ACL2 Version 2.5 built June 13, 2001  08:50:32.
 Copyright (C) 2000  University of Texas at Austin
 ACL2 comes with ABSOLUTELY NO WARRANTY.  This is free software and you
 are welcome to redistribute it under certain conditions.  For details,
 see the GNU General Public License.

 Initialized with (INITIALIZE-ACL2 'INCLUDE-BOOK *ACL2-PASS-2-FILES* T).
 See the documentation topic note-2-5 for recent changes.

 NOTE!!  Proof trees are disabled in ACL2.  To enable them in emacs,
 look under the ACL2 source directory in interface/emacs/README.doc;
 and, to turn on proof trees, execute :START-PROOF-TREE in the ACL2
 command loop.   Look in the ACL2 documentation under PROOF-TREE.

ACL2 Version 2.5.  Level 1.  Cbd
"/home/pauillac/coq4/delahaye/ACL2-Scripts/".
Type :help for help.

ACL2 !>(defmacro naturalp (x)
  (list 'and (list 'integerp x) (list '<= 0 x)))

Summary
Form:  ( DEFMACRO NATURALP ...)
Rules: NIL
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 NATURALP
ACL2 !>(defun eqdec (n m)
  (if (and (naturalp n) (naturalp m))
    (if (= n 0) (= m 0)
      (if (= m 0) nil (eqdec (- n 1) (- m 1)))) nil))

For the admission of EQDEC we will use the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT N).  The non-trivial part of the measure
conjecture is

Goal
(IMPLIES (AND (AND (INTEGERP N)
```

```
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (NOT (= M 0)))
         (E0-ORD-< (ACL2-COUNT (+ -1 N))
                   (ACL2-COUNT N))).

By the simple :definition = we reduce the conjecture to

Goal'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0)))
         (E0-ORD-< (ACL2-COUNT (+ -1 N))
                   (ACL2-COUNT N))).

But we reduce the conjecture to T, by case analysis.

Q.E.D.

That completes the proof of the measure theorem for EQDEC.  Thus, we
admit this function under the principle of definition. We observe that
the type of EQDEC is described by the theorem
(OR (EQUAL (EQDEC N M) T) (EQUAL (EQDEC N M) NIL)).  We used primitive
type reasoning.

Summary
Form:  ( DEFUN EQDEC ...)
Rules: ((:DEFINITION =)
        (:DEFINITION NOT)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)
 EQDEC
ACL2 !>(defthm eq_nat_dec
  (implies (and (naturalp n) (naturalp m))
    (and (implies (eqdec n m) (= n m))
         (implies (= n m) (eqdec n m))))
  :rule-classes NIL :instructions (prove))

Entering the proof-checker....

->: PROVE
***** Now entering the theorem prover *****

By the simple :definition = we reduce the conjecture to
```

```
Goal'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

This simplifies, using primitive type reasoning and the :type-prescription
rule EQDEC, to the following two conjectures.

Subgoal 2
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQDEC N M))
         (EQUAL N M)).

Name the formula above *1.

Subgoal 1
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQUAL N M))
         (EQDEC M M)).

This simplifies, using trivial observations, to

Subgoal 1'
(IMPLIES (AND (INTEGERP M) (<= 0 M))
         (EQDEC M M)).

Normally we would attempt to prove this formula by induction.  However,
we prefer in this instance to focus on the original input conjecture
rather than this simplified special case.  We therefore abandon our
previous work on this conjecture and reassign the name *1 to the original
conjecture.  (See :DOC otf-flg.)

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  Subsumption reduces that number to one.

We will induct according to a scheme suggested by (EQDEC N M).  If
we let  (:P M N) denote *1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (AND (INTEGERP N)
                        (<= 0 N)
```

```
                        (INTEGERP M)
                        (<= 0 M)))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (NOT (= N 0))
                   (NOT (= M 0))
                   (:P (+ -1 M) (+ -1 N)))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (NOT (= N 0))
                   (= M 0))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (= N 0))
              (:P M N))).
This induction is justified by the same argument used to admit EQDEC,
namely, the measure (ACL2-COUNT N) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  Note, however, that the unmeasured variable M is
being instantiated.  When applied to the goal at hand the above induction
scheme produces the following three nontautological subgoals.

Subgoal *1/3
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (NOT (= M 0))
              (IMPLIES (AND (INTEGERP (+ -1 N))
                            (<= 0 (+ -1 N))
                            (INTEGERP (+ -1 M))
                            (<= 0 (+ -1 M)))
                       (AND (IMPLIES (EQDEC (+ -1 N) (+ -1 M))
                                     (= (+ -1 N) (+ -1 M)))
                            (IMPLIES (= (+ -1 N) (+ -1 M))
                                     (EQDEC (+ -1 N) (+ -1 M))))))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
```

```
                  (AND (IMPLIES (EQDEC N M) (= N M))
                       (IMPLIES (= N M) (EQDEC N M))))).

By the simple :definition = we reduce the conjecture to

Subgoal *1/3'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (OR (NOT (AND (INTEGERP (+ -1 N))
                            (<= 0 (+ -1 N))
                            (INTEGERP (+ -1 M))
                            (<= 0 (+ -1 M))))
                  (AND (OR (NOT (EQDEC (+ -1 N) (+ -1 M)))
                           (EQUAL (+ -1 N) (+ -1 M)))
                       (OR (NOT (EQUAL (+ -1 N) (+ -1 M)))
                           (EQDEC (+ -1 N) (+ -1 M))))))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

This simplifies, using the :definitions EQDEC and NOT, primitive type
reasoning and the :type-prescription rule EQDEC, to the following three
conjectures.

Subgoal *1/3.3
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (NOT (EQDEC (+ -1 N) (+ -1 M)))
              (NOT (EQUAL (+ -1 N) (+ -1 M)))
              (EQUAL N M))
         (EQDEC (+ -1 M) (+ -1 M))).

But simplification reduces this to T, using trivial observations.

Subgoal *1/3.2
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (EQDEC (+ -1 N) (+ -1 M))
              (EQUAL (+ -1 N) (+ -1 M)))
```

```
         (EQUAL N M)).

But simplification reduces this to T, using linear arithmetic.

Subgoal *1/3.1
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (EQDEC (+ -1 N) (+ -1 M))
              (EQUAL (+ -1 N) (+ -1 M)))
         (EQDEC (+ -1 M) (+ -1 M))).

But simplification reduces this to T, using trivial observations.

Subgoal *1/2
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (= M 0))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
                  (AND (IMPLIES (EQDEC N M) (= N M))
                       (IMPLIES (= N M) (EQDEC N M))))).

By the simple :definition = we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (EQUAL M 0))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

But simplification reduces this to T, using the :definition EQDEC,
the :executable-counterparts of <, EQUAL, INTEGERP and NOT and primitive
type reasoning.

Subgoal *1/1
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
```

```
                   (INTEGERP M)
                   (<= 0 M))
              (= N 0))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
                  (AND (IMPLIES (EQDEC N M) (= N M))
                       (IMPLIES (= N M) (EQDEC N M))))).

By the simple :definition = we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQUAL N 0))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

But simplification reduces this to T, using the :definition EQDEC,
the :executable-counterparts of <, EQDEC, EQUAL, INTEGERP and NOT and
primitive type reasoning.

That completes the proof of *1.

Q.E.D.

*!*!*!*!*!*!* All goals have been proved! *!*!*!*!*!*!*

Summary
Form:  ( DEFTHM EQ_NAT_DEC ...)
Rules: ((:DEFINITION =)
        (:DEFINITION EQDEC)
        (:DEFINITION IMPLIES)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART <)
        (:EXECUTABLE-COUNTERPART EQDEC)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART INTEGERP)
        (:EXECUTABLE-COUNTERPART NOT)
        (:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:TYPE-PRESCRIPTION EQDEC))
Warnings:  None
Time:  0.06 seconds (prove: 0.02, print: 0.03, other: 0.01)
 EQ_NAT_DEC
```

A.4.2 Preuve en mode batch

```
GCL (GNU Common Lisp)  Version(2.3) Wed Jun 13 16:26:08 CEST 2001
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter

 ACL2 Version 2.5 built June 13, 2001  08:50:32.
 Copyright (C) 2000  University of Texas at Austin
 ACL2 comes with ABSOLUTELY NO WARRANTY.  This is free software and you
 are welcome to redistribute it under certain conditions.  For details,
 see the GNU General Public License.

 Initialized with (INITIALIZE-ACL2 'INCLUDE-BOOK *ACL2-PASS-2-FILES* T).
 See the documentation topic note-2-5 for recent changes.

 NOTE!!  Proof trees are disabled in ACL2.  To enable them in emacs,
 look under the ACL2 source directory in interface/emacs/README.doc;
 and, to turn on proof trees, execute :START-PROOF-TREE in the ACL2
 command loop.   Look in the ACL2 documentation under PROOF-TREE.

ACL2 Version 2.5.  Level 1.  Cbd
"/home/pauillac/coq4/delahaye/ACL2-Scripts/".
Type :help for help.

ACL2 !>(certify-book "eq_nat_dec")

CERTIFICATION ATTEMPT FOR
"/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp"
ACL2 Version 2.5


* Step 1:  Read "/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp"
and compute its check sum.

* Step 2:  There were five forms in the file. The check sum is 67936999.
We now attempt to establish that each form, whether local or non-local,
is indeed an admissible embedded event form in the context of the previously
admitted ones.  Note that proof-tree output is inhibited during this
check; see :DOC proof-tree.


ACL2 !>>(DEFMACRO NATURALP (X)
                  (LIST 'AND
                        (LIST 'INTEGERP X)
                        (LIST '<= 0 X)))

Summary
Form:  ( DEFMACRO NATURALP ...)
Rules: NIL
Warnings:  None
```

```
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
NATURALP


ACL2 !>>(DEFUN EQDEC (N M)
               (IF (AND (NATURALP N) (NATURALP M))
                   (IF (= N 0)
                       (= M 0)
                       (IF (= M 0)
                           NIL (EQDEC (- N 1) (- M 1))))
                   NIL))

For the admission of EQDEC we will use the relation E0-ORD-< (which
is known to be well-founded on the domain recognized by E0-ORDINALP)
and the measure (ACL2-COUNT N).  The non-trivial part of the measure
conjecture is

Goal
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (NOT (= M 0)))
         (E0-ORD-< (ACL2-COUNT (+ -1 N))
                   (ACL2-COUNT N))).

By the simple :definition = we reduce the conjecture to

Goal'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0)))
         (E0-ORD-< (ACL2-COUNT (+ -1 N))
                   (ACL2-COUNT N))).

But we reduce the conjecture to T, by case analysis.

Q.E.D.

That completes the proof of the measure theorem for EQDEC.  Thus, we
admit this function under the principle of definition. We observe that
the type of EQDEC is described by the theorem
(OR (EQUAL (EQDEC N M) T) (EQUAL (EQDEC N M) NIL)).  We used primitive
type reasoning.

Summary
```

```
Form:  ( DEFUN EQDEC ...)
Rules: ((:DEFINITION =)
        (:DEFINITION NOT)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)
EQDEC


ACL2 !>>(DEFTHM EQ_NAT_DEC
                (IMPLIES (AND (NATURALP N) (NATURALP M))
                         (AND (IMPLIES (EQDEC N M) (= N M))
                              (IMPLIES (= N M) (EQDEC N M))))
                :RULE-CLASSES NIL :INSTRUCTIONS (PROVE))

Entering the proof-checker....

->: PROVE
***** Now entering the theorem prover *****

By the simple :definition = we reduce the conjecture to

Goal'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

This simplifies, using primitive type reasoning and the :type-prescription
rule EQDEC, to the following two conjectures.

Subgoal 2
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQDEC N M))
         (EQUAL N M)).

Name the formula above *1.

Subgoal 1
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQUAL N M))
         (EQDEC M M)).
```

```
This simplifies, using trivial observations, to

Subgoal 1'
(IMPLIES (AND (INTEGERP M) (<= 0 M))
         (EQDEC M M)).

Normally we would attempt to prove this formula by induction.  However,
we prefer in this instance to focus on the original input conjecture
rather than this simplified special case.  We therefore abandon our
previous work on this conjecture and reassign the name *1 to the original
conjecture.  (See :DOC otf-flg.)

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  Subsumption reduces that number to one.

We will induct according to a scheme suggested by (EQDEC N M).  If
we let  (:P M N) denote *1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M)))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (NOT (= N 0))
                   (NOT (= M 0))
                   (:P (+ -1 M) (+ -1 N)))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (NOT (= N 0))
                   (= M 0))
              (:P M N))
     (IMPLIES (AND (AND (INTEGERP N)
                        (<= 0 N)
                        (INTEGERP M)
                        (<= 0 M))
                   (= N 0))
              (:P M N))).
This induction is justified by the same argument used to admit EQDEC,
namely, the measure (ACL2-COUNT N) is decreasing according to the relation
E0-ORD-< (which is known to be well-founded on the domain recognized
by E0-ORDINALP).  Note, however, that the unmeasured variable M is
being instantiated.  When applied to the goal at hand the above induction
```

scheme produces the following three nontautological subgoals.

```
Subgoal *1/3
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (NOT (= M 0))
              (IMPLIES (AND (INTEGERP (+ -1 N))
                            (<= 0 (+ -1 N))
                            (INTEGERP (+ -1 M))
                            (<= 0 (+ -1 M)))
                       (AND (IMPLIES (EQDEC (+ -1 N) (+ -1 M))
                                     (= (+ -1 N) (+ -1 M)))
                            (IMPLIES (= (+ -1 N) (+ -1 M))
                                     (EQDEC (+ -1 N) (+ -1 M))))))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
                  (AND (IMPLIES (EQDEC N M) (= N M))
                       (IMPLIES (= N M) (EQDEC N M))))).
```

By the simple :definition = we reduce the conjecture to

```
Subgoal *1/3'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (OR (NOT (AND (INTEGERP (+ -1 N))
                            (<= 0 (+ -1 N))
                            (INTEGERP (+ -1 M))
                            (<= 0 (+ -1 M))))
                  (AND (OR (NOT (EQDEC (+ -1 N) (+ -1 M)))
                           (EQUAL (+ -1 N) (+ -1 M)))
                       (OR (NOT (EQUAL (+ -1 N) (+ -1 M)))
                           (EQDEC (+ -1 N) (+ -1 M))))))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).
```

This simplifies, using the :definitions EQDEC and NOT, primitive type reasoning and the :type-prescription rule EQDEC, to the following three conjectures.

```
Subgoal *1/3.3
(IMPLIES (AND (INTEGERP N)
```

```
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (NOT (EQDEC (+ -1 N) (+ -1 M)))
              (NOT (EQUAL (+ -1 N) (+ -1 M)))
              (EQUAL N M))
         (EQDEC (+ -1 M) (+ -1 M))).

But simplification reduces this to T, using trivial observations.

Subgoal *1/3.2
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (EQDEC (+ -1 N) (+ -1 M))
              (EQUAL (+ -1 N) (+ -1 M)))
         (EQUAL N M)).

But simplification reduces this to T, using linear arithmetic.

Subgoal *1/3.1
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (NOT (EQUAL M 0))
              (EQDEC (+ -1 N) (+ -1 M))
              (EQUAL (+ -1 N) (+ -1 M)))
         (EQDEC (+ -1 M) (+ -1 M))).

But simplification reduces this to T, using trivial observations.

Subgoal *1/2
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (NOT (= N 0))
              (= M 0))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
                  (AND (IMPLIES (EQDEC N M) (= N M))
```

```
                        (IMPLIES (= N M) (EQDEC N M))))).

By the simple :definition = we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (NOT (EQUAL N 0))
              (EQUAL M 0))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

But simplification reduces this to T, using the :definition EQDEC,
the :executable-counterparts of <, EQUAL, INTEGERP and NOT and primitive
type reasoning.

Subgoal *1/1
(IMPLIES (AND (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (= N 0))
         (IMPLIES (AND (INTEGERP N)
                       (<= 0 N)
                       (INTEGERP M)
                       (<= 0 M))
                  (AND (IMPLIES (EQDEC N M) (= N M))
                       (IMPLIES (= N M) (EQDEC N M))))).

By the simple :definition = we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M)
              (EQUAL N 0))
         (AND (OR (NOT (EQDEC N M)) (EQUAL N M))
              (OR (NOT (EQUAL N M)) (EQDEC N M)))).

But simplification reduces this to T, using the :definition EQDEC,
the :executable-counterparts of <, EQDEC, EQUAL, INTEGERP and NOT and
primitive type reasoning.

That completes the proof of *1.

Q.E.D.
```

```
*!*!*!*!*!*!* All goals have been proved! *!*!*!*!*!*!*

Summary
Form:  ( DEFTHM EQ_NAT_DEC ...)
Rules: ((:DEFINITION =)
        (:DEFINITION EQDEC)
        (:DEFINITION IMPLIES)
        (:DEFINITION NOT)
        (:EXECUTABLE-COUNTERPART <)
        (:EXECUTABLE-COUNTERPART EQDEC)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART INTEGERP)
        (:EXECUTABLE-COUNTERPART NOT)
        (:FAKE-RUNE-FOR-LINEAR NIL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:TYPE-PRESCRIPTION EQDEC))
Warnings:  None
Time:  0.06 seconds (prove: 0.04, print: 0.02, other: 0.00)
EQ_NAT_DEC


ACL2 !>>(VERIFY-GUARDS EQDEC)

The non-trivial part of the guard conjecture for EQDEC is

Goal
(AND (IMPLIES (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (ACL2-NUMBERP N))
     (IMPLIES (AND (INTEGERP N)
                   (<= 0 N)
                   (INTEGERP M)
                   (<= 0 M))
              (ACL2-NUMBERP M))).

By case analysis we reduce the conjecture to the following two conjectures.

Subgoal 2
(IMPLIES (AND (INTEGERP N)
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M))
         (ACL2-NUMBERP N)).

But we reduce the conjecture to T, by case analysis.

Subgoal 1
(IMPLIES (AND (INTEGERP N)
```

```
              (<= 0 N)
              (INTEGERP M)
              (<= 0 M))
         (ACL2-NUMBERP M)).

But we reduce the conjecture to T, by case analysis.

Q.E.D.

That completes the proof of the guard theorem for EQDEC.  EQDEC is
compliant with Common Lisp.

Summary
Form:  (VERIFY-GUARDS EQDEC)
Rules: ((:DEFINITION NOT))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
EQDEC

* Step 3:  That completes the admissibility check.  Each form read
was an embedded event form and was admissible. We now retract back
to the initial world and try to include the book.  This may expose
local incompatibilities.

Summary
Form:  ( INCLUDE-BOOK "eq_nat_dec" ...)
Rules: NIL
Warnings:  None
Time:  0.01 seconds (prove: 0.00, print: 0.00, other: 0.01)

* Step 4:  Write the certificate for
"/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp" in
"/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.cert".  The final
check sum alist is
(("/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp"
      "eq_nat_dec" "eq_nat_dec"
      ((:SKIPPED-PROOFSP) (:AXIOMSP))
      . 67936999)).


* Step 5:  Compile the functions defined in
"/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp".
Compiling /home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp.
End of Pass 1.

;; Note: Tail-recursive call of EQDEC was replaced by iteration.
End of Pass 2.
OPTIMIZE levels: Safety=0 (No runtime error checking), Space=0, Speed=3
Finished compiling /home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.o.
```

```
"/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.o"

Loading /home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.o
start address -T 9482a50
  Finished loading /home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.o

Summary
Form:  (CERTIFY-BOOK "eq_nat_dec" ...)
Rules: NIL
Warnings:  None
Time:  0.09 seconds (prove: 0.04, print: 0.02, other: 0.03)
 "/home/pauillac/coq4/delahaye/ACL2-Scripts/eq_nat_dec.lisp"
```

A.5 Alfa

La preuve de la figure A.1 a été construite sous Alfa (version du 01.07.2001), sous Linux Mandrake 7.2 sur un Intel Pentium III à 1 GHz.

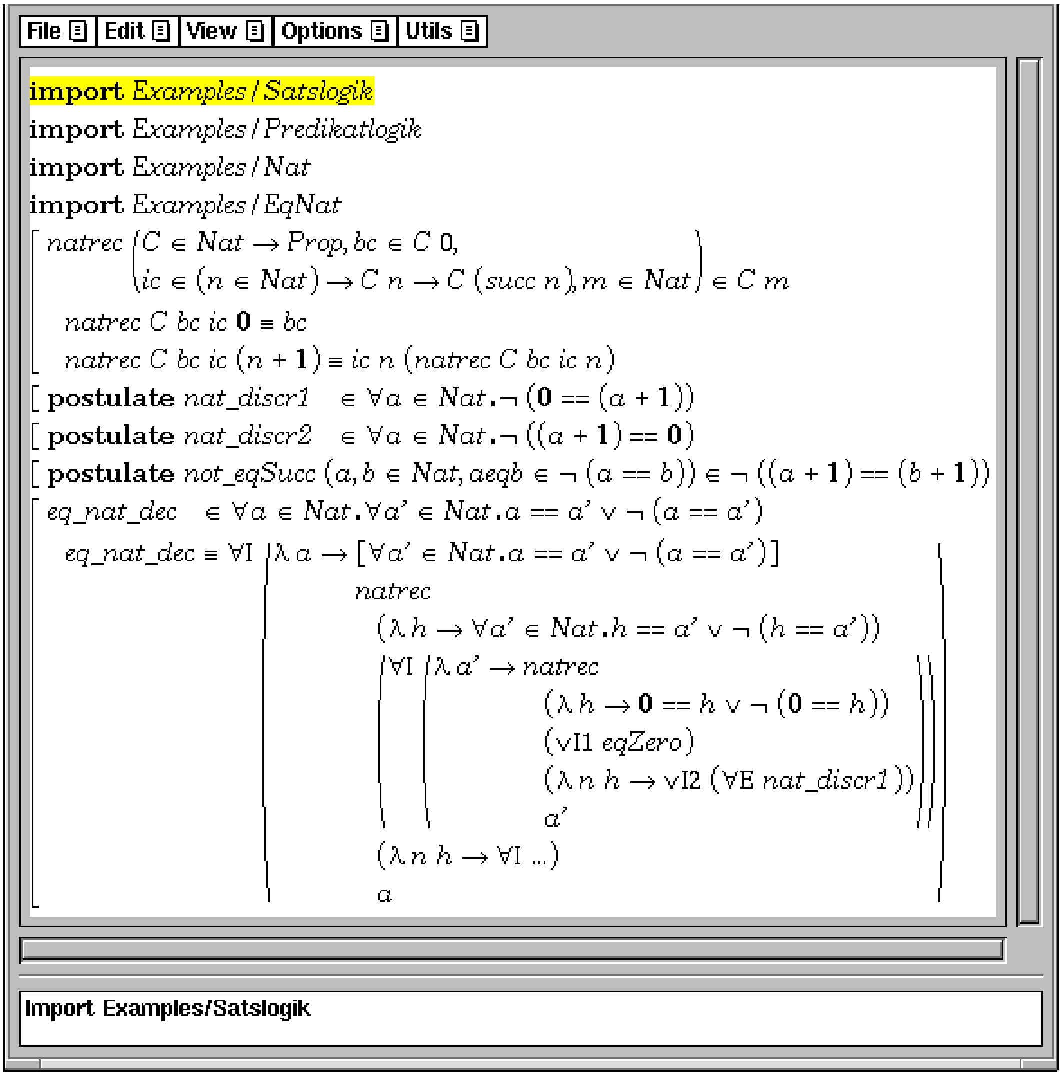


FIG. A.1 – Preuve de la décidabilité de l'égalité sur les entiers naturels en Alfa.

Annexe B

Règles d'erreurs de $\mathcal{L}_{pdt}$

B.1 Conventions

Dans les règles d'erreurs, nous utiliserons les mots-clés *et* et *ou*, avec leur sémantique usuelle. Le *et* aura une précédence strictement plus forte que celle du *ou*. La virgule utilisée entre les différents prédicats ou relations aura exactement la sémantique du *et*.

B.2 Règles d'erreurs

Les règles d'erreurs de $\mathcal{L}_{pdt}$ sont données par les figures B.1, B.2, B.3, B.4, B.5, B.6, B.7, B.8, B.9, B.10, B.11, B.12 et B.13. Les figures B.1, B.2, B.3, B.4 représentent les règles d'erreurs d'évaluation concernant les termes purs, les figures B.5, B.6, celles concernant les parties déclaratives, les figures B.7, B.8, B.9, celles concernant les parties procédurales (les tactiques étant traitées, en particulier, dans les figures B.8 et B.9), les figures B.10, B.11, B.12, celles concernant les phrases et la figure B.13, celles concernant les scripts.

$$\frac{x \notin S \qquad \mathsf{Erreur} = \mathsf{access}(x, \Delta[\Gamma])}{(x, \Delta[\Gamma \Vdash Type]) \triangleright \mathsf{Erreur}}\ (\mathsf{TSort\text{-}Err})$$

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \notin S \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S \\ \text{et } (T_2, \Delta[\Gamma_1, (x : T_3)]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S \\ \text{et } (T_2, \Delta[\Gamma_1, (x : T_3)]) \triangleright (T_4, \Delta[\Gamma_2, (x : T_5) \Vdash s_2], m_2, \sigma_2), s_2 \notin S\end{array}}{((x : T_1)T_2, \Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{UProd\text{-}Err})$$

$$\frac{\begin{array}{c}s_1 \notin S \text{ ou } (T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_2], m_1, \sigma_1), s_2 \notin S \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_2], m_1, \sigma_1), s_2 \in S \\ \text{et } (T_2, \Delta[\Gamma_1, (x : T_3) \Vdash s_1]) \triangleright \mathsf{Erreur}\end{array}}{((x : T_1)T_2, \Delta[\Gamma \Vdash s_1]) \triangleright \mathsf{Erreur}}\ (\mathsf{TProd\text{-}Err})$$

$$\frac{x \notin S \qquad \mathsf{Erreur} = \mathsf{access}(x, \Delta[\Gamma])}{(x, \Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{UVar\text{-}Err})$$

$$\frac{\begin{array}{c}T \neq Type \\ (T, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma_1), s \notin S \\ \text{ou } \mathsf{Erreur} = \mathsf{access}(x, \Delta[\Gamma]) \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma_1), s \in S \\ \text{et } T_2 = \mathsf{access}(x, \Delta[\Gamma]),\ \mathsf{Erreur} = \mathsf{unify}(T_1, T_2\sigma_1)\end{array}}{(x, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur}}\ (\mathsf{TVar\text{-}Err})$$

$$\frac{\begin{array}{c}(T, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma),\ s \notin S\end{array}}{(?, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur}}\ (\mathsf{Impl\text{-}Err})$$

$$\frac{\begin{array}{c}(T, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m, \sigma),\ s \notin S \\ \text{ou } \exists i.(?i \in \Gamma \text{ ou } ?i \in T)\end{array}}{(?n, \Delta[\Gamma \Vdash T]) \triangleright \mathsf{Erreur}}\ (\mathsf{Meta\text{-}Err})$$

FIG. B.1 – Erreurs dans le raffinement des termes purs (1/4).

$$\frac{\begin{array}{c} x \in \Delta[\Gamma] \text{ ou } (T, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1), s \notin S \\ \text{ou } (T, \Delta[\Gamma]) \triangleright (T_1, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1), s \in S \\ \text{et } (t, \Delta[\Gamma_1, (x : T_1)]) \triangleright \mathsf{Erreur} \end{array}}{([x : T]t, \Delta[\Gamma]) \triangleright \mathsf{Erreur}} \; (\lambda_{\mathsf{UChurch\text{-}Err}})$$

$$\frac{\begin{array}{c} x \in \Delta[\Gamma] \text{ ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright ((x : T_4)T_5, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), \; s_1 \notin S \\ \text{ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright ((x : T_4)T_5, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), \; s_1 \in S \\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright \mathsf{Erreur} \\ \text{ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright ((x : T_4)T_5, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), \; s_1 \in S \\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_6, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2), \; s_2 \notin S \\ \text{ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright ((x : T_4)T_5, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), \; s_1 \in S \\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_6, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2), \; s_2 \in S \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_2\sigma_2, T_6) \\ \text{ou } ((x : T_2)T_3, \Delta[\Gamma]) \triangleright ((x : T_4)T_5, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), \; s_1 \in S \\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_6, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2), \; s_2 \in S \\ \text{et } \sigma_3 = \mathsf{unify}(T_2\sigma_2, T_6), \; (t, \Delta[\Gamma_2, (x : T_6) \Vdash T_5\sigma_2]\sigma_3) \triangleright \mathsf{Erreur} \end{array}}{([x : T_1]t, \Delta[\Gamma \Vdash (x : T_2)T_3]) \triangleright \mathsf{Erreur}} \; (\lambda_{\mathsf{TChurch\text{-}Err}})$$

$$\frac{x \in \Delta[\Gamma] \text{ ou } \mathsf{new\_impl}(n), \; (t, \Delta[\Gamma, (x : ?_{in})]) \triangleright \mathsf{Erreur}}{([x : ?]t, \Delta[\Gamma]) \triangleright \mathsf{Erreur}} \; (\lambda_{\mathsf{UCurry\text{-}Err}})$$

$$\frac{\begin{array}{c} x \in \Delta[\Gamma] \text{ ou } ((x : T_1)T_2, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } ((x : T_1)T_2, \Delta[\Gamma]) \triangleright ((x : T_3)T_4, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1), \; s \notin S \\ \text{ou } ((x : T_1)T_2, \Delta[\Gamma]) \triangleright ((x : T_3)T_4, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1), \; s \in S \\ \text{et } (t, \Delta[\Gamma_1, (x : T_3) \Vdash T_4]) \triangleright \mathsf{Erreur} \end{array}}{([x : ?]t, \Delta[\Gamma \Vdash (x : T_1)T_2]) \triangleright \mathsf{Erreur}} \; (\lambda_{\mathsf{TCurry\text{-}Err}})$$

FIG. B.2 – Erreurs dans le raffinement des termes purs (2/4).

$$\frac{\begin{array}{c} t_2 =? \text{ ou } \exists i.t_2 =?i \\ \text{ou } (t_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (t_1, \Delta[\Gamma]) \triangleright (t_3, \Delta[\Gamma_1 \Vdash (x:T_1)T_2], m_1, \sigma_1) \\ \text{et } (t_2, \Delta[\Gamma_1]) \triangleright \mathsf{Erreur} \\ \text{ou } (t_1, \Delta[\Gamma]) \triangleright (t_3, \Delta[\Gamma_1 \Vdash (x:T_1)T_2], m_1, \sigma_1) \\ \text{et } (t_2, \Delta[\Gamma_1]) \triangleright (t_4, \Delta[\Gamma_2 \Vdash T_3], m_2, \sigma_2) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_1\sigma_2, T_3) \end{array}}{((t_1\ t_2), \Delta[\Gamma]) \triangleright \mathsf{Erreur}} \ (\mathsf{UApp\text{-}Err})$$

$$\frac{\begin{array}{c} t_2 =? \text{ ou } \exists i.t_2 =?i \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \notin S \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright (t_3, \Delta[\Gamma_2 \Vdash (x:T_3)T_4], m_2, \sigma_2) \\ \text{et } (t_2, \Delta[\Gamma_2]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright (t_3, \Delta[\Gamma_2 \Vdash (x:T_3)T_4], m_2, \sigma_2) \\ \text{et } (t_2, \Delta[\Gamma_2]) \triangleright (t_4, \Delta[\Gamma_3 \Vdash T_5], m_3, \sigma_3) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_3\sigma_3, T_5) \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright (t_3, \Delta[\Gamma_2 \Vdash (x:T_3)T_4], m_2, \sigma_2) \\ \text{et } (t_2, \Delta[\Gamma_2]) \triangleright (t_4, \Delta[\Gamma_3 \Vdash T_5], m_3, \sigma_3) \\ \text{et } \sigma_4 = \mathsf{unify}(T_3\sigma_3, T_5) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_2\sigma_2\sigma_3\sigma_4, T_4\sigma_3[x \backslash t_4]\sigma_4) \end{array}}{((t_1\ t_2), \Delta[\Gamma \Vdash T_1]) \triangleright \mathsf{Erreur}} \ (\mathsf{TApp\text{-}Err})$$

$$\frac{\begin{array}{c} t_2 \neq \ ? \text{ et } \forall i.t_2 \neq \ ?i \\ \text{ou } (t_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (t_1, \Delta[\Gamma]) \triangleright (t_3, \Delta[\Gamma_1 \Vdash (x:T_1)T_2], m_1, \sigma_1) \\ \text{et } (t_2, \Delta[\Gamma_1 \Vdash T_1]) \triangleright \mathsf{Erreur} \end{array}}{((t_1\ t_2), \Delta[\Gamma]) \triangleright \mathsf{Erreur}} \ (\mathsf{UAppsynt\text{-}Err})$$

$$\frac{\begin{array}{c} t_2 \neq \ ? \text{ et } \forall i.t_2 \neq \ ?i \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \notin S \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright (t_3, \Delta[\Gamma_2 \Vdash (x:T_3)T_4], m_2, \sigma_2) \\ \text{et } (t_2, \Delta[\Gamma_2 \Vdash T_3]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ s \in S \\ \text{et } (t_1, \Delta[\Gamma_1]) \triangleright (t_3, \Delta[\Gamma_2 \Vdash (x:T_3)T_4], m_2, \sigma_2) \\ \text{et } (t_2, \Delta[\Gamma_2 \Vdash T_3]) \triangleright (t_4, \Delta[\Gamma_3 \Vdash T_5], m_3, \sigma_3) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_2\sigma_2\sigma_3, T_4\sigma_3[x \backslash t_4]) \end{array}}{((t_1\ t_2), \Delta[\Gamma \Vdash T_1]) \triangleright \mathsf{Erreur}} \ (\mathsf{TAppSynt\text{-}Err})$$

FIG. B.3 – Erreurs dans le raffinement des termes purs (3/4).

$$\frac{\begin{array}{c}(t,\Delta[\Gamma]) \triangleright \mathsf{Erreur}\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (\mathsf{Erreur} = \mathsf{induct\_info}(\Delta,T_1) \text{ ou } (P\sigma_1,\Delta[\Gamma_1]) \triangleright \mathsf{Erreur})\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ \text{et } (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l) \to s)\sigma_2],m_2,\sigma_2)\\ \text{et } (\mathsf{Erreur} = \mathsf{elim}(I,s) \text{ ou } P \neq (x_1:o1)\ldots(x_l:ol)(c:oi)P_b)\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ \text{et } (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l) \to s)\sigma_2],m_2,\sigma_2)\\ \text{et } s = \mathsf{elim}(I,s),\ P = (x_1:o1)\ldots(x_l:ol)(c:oi)P_b\\ \text{et } \exists i.(ap_i = \mathsf{abstract}(I,c_i\sigma_{fi},t_i\sigma_{fi})\\ \text{et } it_i = \mathsf{branch\_type}(\Delta,I,c_i,\{p_1,\ldots,p_k\}\sigma_{fi},P\sigma_{fi})\\ \text{et } (ap_i,\Delta[\Gamma_{i1} \Vdash it_i]) \triangleright \mathsf{Erreur}),\ i=1,\ldots,n\end{array}}{(<P> Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end,\Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{UCases\text{-}Err})$$

$$\text{où } \sigma_{fi} = \sigma_1\sigma_2\sigma_{11}\ldots\sigma_{(i-1)1}$$

$$\frac{\begin{array}{c}(t,\Delta[\Gamma]) \triangleright \mathsf{Erreur}\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (\mathsf{Erreur} = \mathsf{induct\_info}(\Delta,T_1) \text{ ou } (P\sigma_1,\Delta[\Gamma_1]) \triangleright \mathsf{Erreur})\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ \text{et } (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l) \to s)\sigma_2],m_2,\sigma_2)\\ \text{et } (\mathsf{Erreur} = \mathsf{elim}(I,s) \text{ ou } P \neq (x_1:o1)\ldots(x_l:ol)(c:oi)P_b)\\ \text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ \text{et } (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l) \to s)\sigma_2],m_2,\sigma_2)\\ \text{et } s = \mathsf{elim}(I,s),\ P = (x_1:o1)\ldots(x_l:ol)(c:oi)P_b\\ \text{et } \exists i.(ap_i = \mathsf{abstract}(I,c_i\sigma_{fi},t_i\sigma_{fi})\\ \text{et } it_i = \mathsf{branch\_type}(\Delta,I,c_i,\{p_1,\ldots,p_k\}\sigma_{fi},P\sigma_{fi})\\ \text{et } (ap_i,\Delta[\Gamma_{i1} \Vdash it_i]) \triangleright \mathsf{Erreur}),\ i=1,\ldots,n\text{ou } (t,\Delta[\Gamma]) \triangleright (t_1,\Delta[\Gamma_1 \Vdash T_1],m_1,\sigma_1)\\ \text{et } (I,\{p_1,\ldots,p_k\},\{a_1,\ldots,a_l\},\{ta_1,\ldots,ta_l\}) = \mathsf{induct\_info}(\Delta,T_1)\\ \text{et } (P\sigma_1,\Delta[\Gamma_1]) \triangleright\\ (P_1,\Delta[\Gamma_2 \Vdash ((x_1:ta_1)\ldots(x_l:ta_l)(I\ p_1\ldots p_k\ x_1\ldots x_l) \to s)\sigma_2],m_2,\sigma_2)\\ \text{et } s = \mathsf{elim}(I,s),\ P = (x_1:o1)\ldots(x_l:ol)(c:oi)P_b\\ \text{et } \forall i.(ap_i = \mathsf{abstract}(I,c_i\sigma_{fi},t_i\sigma_{fi})\\ \text{et } it_i = \mathsf{branch\_type}(\Delta,I,c_i,\{p_1,\ldots,p_k\}\sigma_{fi},P\sigma_{fi})\\ \text{et } (ap_i,\Delta[\Gamma_{i1} \Vdash it_i]) \triangleright (ap_{i1},\Delta[\Gamma_{(i+1)1} \Vdash it_{i1}],m_{i1},\sigma_{i1})),\ i=1,\ldots,n\\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T\sigma_{f(n+1)},(P_b[x_1\backslash a_1;\ldots;x_l\backslash a_l;c\backslash t]\sigma_{f(n+1)})\end{array}}{(<P> Cases\ t\ of\ |c_1 => t_1\ |\ \ldots\ |c_n => t_n\ end,\Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{TCases\text{-}Err})$$

$$\text{où } \sigma_{fi} = \sigma_1\sigma_2\sigma_{11}\ldots\sigma_{(i-1)1}$$

FIG. B.4 – Erreurs dans le raffinement des termes purs (4/4).

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur}\\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ \ s \notin S\\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ \ s \in S\\ \text{et } (t_1, \Delta[\Gamma_1 \Vdash T_2]) \triangleright \mathsf{Erreur}\\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ \ s \in S\\ \text{et } (t_1, \Delta[\Gamma_1 \Vdash T_2]) \triangleright (t_3, \Vdash [\Gamma_2 \Vdash T_3], m_2, \sigma_2)\\ \text{et } (t_2, \Delta[\Gamma_2, (x : T_3)]) \triangleright \mathsf{Erreur}\end{array}}{(Let\ x : T_1 := t_1\ In\ t_2, \Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{ULetIn\text{-}Err})$$

$$\frac{\begin{array}{c}(T_2, \Delta[\Gamma]) \triangleright \mathsf{Erreur}\\ \text{ou } (T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \notin S\\ \text{ou } (T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S\\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright \mathsf{Erreur}\\ \text{ou } (T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S\\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_4, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2),\ \ s_2 \notin S\\ \text{ou } (T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S\\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_4, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2),\ \ s_2 \in S\\ \text{et } (t_1, \Delta[\Gamma_2 \Vdash T_4]) \triangleright \mathsf{Erreur}\\ \text{ou } (T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S\\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_4, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2),\ \ s_2 \in S\\ \text{et } (t_1, \Delta[\Gamma_2 \Vdash T_4]) \triangleright (t_3, \Vdash [\Gamma_3 \Vdash T_5], m_3, \sigma_3)\\ \text{et } (t_2, \Delta[\Gamma_3, (x : T_5)]) \triangleright \mathsf{Erreur}\\ \text{ou } T_2, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s_1], m_1, \sigma_1), s_1 \in S\\ \text{et } (T_1, \Delta[\Gamma_1]) \triangleright (T_4, \Delta[\Gamma_2 \Vdash s_2], m_2, \sigma_2),\ \ s_2 \in S\\ \text{et } (t_1, \Delta[\Gamma_2 \Vdash T_4]) \triangleright (t_3, \Vdash [\Gamma_3 \Vdash T_5], m_3, \sigma_3)\\ \text{et } (t_2, \Delta[\Gamma_3, (x : T_5)]) \triangleright (t_4, \Delta[\Gamma_4, (x : T_6) \Vdash T_7], m_4, \sigma_4)\\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_3\sigma_2\sigma_3\sigma_4, T_7[x \backslash t_3\sigma_4])\end{array}}{(Let\ x : T_1 := t_1\ In\ t_2, \Delta[\Gamma \Vdash T_2]) \triangleright \mathsf{Erreur}}\ (\mathsf{TLetIn\text{-}Err})$$

FIG. B.5 – Erreurs dans l'évaluation des parties déclaratives (1/2).

$$\dfrac{\begin{array}{c}\exists i.((T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \notin S \\ \text{ou } (T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \in S \\ \text{et } (t_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]) \triangleright \mathsf{Erreur}),\ i = 1 \ldots n \\ \text{ou } \forall i.((T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \in S \\ \text{et } (t_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]) \triangleright (t_{i1}, \Vdash [\Gamma_{i2} \Vdash T_{i2}], m_{i2}, \sigma_{i2})),\ \ i = 1 \ldots n \\ \text{et } (t_{n+1}, \Delta[\Gamma_{n2}, (x_1 : T_{12}\sigma_{c2}), \ldots, (x_n : T_{n2}\sigma_{c(n+1)})]]) \triangleright \mathsf{Erreur}\end{array}}{(Let\ x_1 : T_1 := t_1\ And\ \ldots And\ x_n : T_n := t_n\ In\ t_{n+1}, \Delta[\Gamma]) \triangleright \mathsf{Erreur}}\ (\mathsf{ULetsIn\text{-}Err})$$

$$\text{où } \Gamma_{02} = \Gamma \text{ et } \sigma_{ci} = \sigma_{i1} \ldots \sigma_{n1}\sigma_{n2}$$

$$\dfrac{\begin{array}{c}(T_{n+1}, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_{n+1}, \Delta[\Gamma]) \triangleright (T_{n+2}, \Delta[\Gamma_1 \Vdash s_{n+1}], m_1, \sigma_1),\ \ s_{n+1} \notin S \\ \text{ou } (T_{n+1}, \Delta[\Gamma]) \triangleright (T_{n+2}, \Delta[\Gamma_1 \Vdash s_{n+1}], m_1, \sigma_1),\ \ s_{n+1} \in S \\ \text{et } \exists i.((T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \notin S \\ \text{ou } (T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \in S \\ \text{et } (t_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]) \triangleright \mathsf{Erreur}),\ i = 1 \ldots n \\ \text{ou } \forall i.((T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \in S \\ \text{et } (t_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]) \triangleright (t_{i1}, \Vdash [\Gamma_{i2} \Vdash T_{i2}], m_{i2}, \sigma_{i2})),\ \ i = 1 \ldots n \\ \text{et } (t_{n+1}, \Delta[\Gamma_{n2}, (x_1 : T_{12}\sigma_{c2}), \ldots, (x_n : T_{n2}\sigma_{c(n+1)})]]) \triangleright \mathsf{Erreur} \\ \text{ou } \forall i.((T_i, \Delta[\Gamma_{(i-1)2}]) \triangleright (T_{i1}, \Delta[\Gamma_{i1} \Vdash s_i], m_{i1}, \sigma_{i1}),\ \ s_i \in S \\ \text{et } (t_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]) \triangleright (t_{i1}, \Vdash [\Gamma_{i2} \Vdash T_{i2}], m_{i2}, \sigma_{i2})),\ \ i = 1 \ldots n \\ \text{et } (t_{n+1}, \Delta[\Gamma_{n2}, (x_1 : T_{12}\sigma_{c2}), \ldots, (x_n : T_{n2}\sigma_{c(n+1)})]]) \triangleright \\ (t_{(n+1)1}, \Delta[\Gamma_{n+1}, (x_1 : T_{13}), \ldots, (x_n : T_{n3}) \Vdash T_{n+3}], m_{n+1}, \sigma_{n+1}) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_{n+2}\sigma_{t1}, T_{n+3}[x_1 \backslash t_{11}\sigma_{t2}; \ldots; x_n \backslash t_{n1}\sigma_{t(n+1)}]) \end{array}}{\begin{array}{c}(Let\ x_1 : T_1 := t_1\ And\ \ldots\ And\ x_n : T_n := t_n\ In\ t_{n+1}, \\ \Delta[\Gamma \Vdash T_{n+1}]) \triangleright \mathsf{Erreur}\end{array}}\ (\mathsf{TLetsIn\text{-}Err})$$

$$\text{où } \Gamma_{02} = \Gamma_1,\ \sigma_{ci} = \sigma_{i1} \ldots \sigma_{n1}\sigma_{n2} \text{ et } \sigma_{ti} = \sigma_{ci}\sigma_{n+1} \text{ et}$$

FIG. B.6 – Erreurs dans l'évaluation des parties déclaratives (2/2).

$$\frac{}{(<\texttt{by}\ tac>, \Delta[\Gamma]) \triangleright \textsf{Erreur}}\ (\textsf{By-Err1})$$

$$\frac{\begin{array}{c}(T_1, \Delta[\Gamma]) \triangleright \textsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ \ s \notin S \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_2, \Delta[\Gamma_1 \Vdash s], m_1, \sigma_1),\ \ s \in S \\ \text{et } (tac, \Delta[\Gamma_1 \Vdash T_2]) \triangleright \textsf{Erreur}\end{array}}{(<\texttt{by}\ tac>, \Delta[\Gamma \Vdash T_1]) \triangleright \textsf{Erreur}}\ (\textsf{By-Err2})$$

FIG. B.7 – Erreurs dans l'évaluation des parties procédurales.

$$\frac{T \neq (x:T_1)T_2}{(\texttt{Intro}, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{AIntro\text{-}Err})$$

$$\frac{T \neq (x:T_1)T_2 \text{ ou } x \in \Delta[\Gamma]}{(\texttt{Intro}\ x, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{NIntro\text{-}Err})$$

$$\frac{\begin{array}{c} T = (x_1:T_1;x_2:T_2;\ldots;x_m:T_m)T_{m+1} \qquad T_{m+1} \neq (x_{m+1}:T_{m+2})T_{m+3} \\ m < n \text{ ou } (m \geq n,\ \exists i.x_i \in \Delta[\Gamma],\ i = 1\ldots n) \end{array}}{(\texttt{Intros}\ x_1\ x_2\ \ldots\ x_n, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{NIntros\text{-}Err})$$

$$\frac{\begin{array}{c} id \notin \Gamma \\ \text{ou } \Gamma = \Gamma_1(id:T)\Gamma_2,\ \exists(x_i:T_i) \in \Gamma_1.id \in T_i \text{ ou } id \in T \end{array}}{(\texttt{Clear}\ id, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{ClearHyp\text{-}Err})$$

$$\frac{\exists i.(\texttt{Clear}\ id_i, \Delta[\Gamma_{i-1} \Vdash T])\triangleright\mathsf{Erreur},\ i = 1,\ldots,n}{(\texttt{Clear}\ id_1\ \ldots\ id_m, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{ClearHyps\text{-}Err})$$

où $\Gamma_0 = \Gamma$

$$\frac{\forall(x_i,T_i) \in \Gamma.\mathsf{Erreur} = \mathsf{unify}(T,T_i)}{(\texttt{Assumption}, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Assumption\text{-}Err})$$

$$\frac{\begin{array}{c} (T_1, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (T_1, \Delta[\Gamma]) \triangleright (T_3, \Delta[\Gamma_1 \Vdash s], \emptyset, \sigma_1),\ s \notin S \end{array}}{(\texttt{Cut}\ T_1, \Delta[\Gamma \Vdash T_2])\triangleright\mathsf{Erreur}}\ (\mathsf{Cut\text{-}Err})$$

$$\frac{\begin{array}{c} T = (x:T_1)T_2 \text{ ou } (t, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \\ \text{ou } (t, \Delta[\Gamma]) \triangleright (t_{n+1}, \Delta[\Gamma_1 \Vdash (x_1:T_1;x_2:T_2;\ldots;x_n:T_n)T_{n+1}], \emptyset, \sigma_1) \\ \text{et } T, T_{n+1} \neq (x_{n+1}:T_{n+2})T_{n+3} \\ \text{et } \mathsf{Erreur} = \mathsf{apply\_unify}(T\sigma_1, (x_1:T_1;x_2:T_2;\ldots;x_n:T_n)T_{n+1}, \Delta[\Gamma_1]) \end{array}}{(\texttt{Apply}\ t, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Apply\text{-}Err})$$

$$\frac{(t, \Delta[\Gamma]) \triangleright \mathsf{Erreur}}{(\texttt{Pattern}\ t, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{PatternAll\text{-}Err})$$

$$\frac{(t, \Delta[\Gamma]) \triangleright \mathsf{Erreur} \text{ ou } \exists i.T_{|n_i} \neq t,\ i = 1\ldots m}{(\texttt{Pattern}\ n_1\ n_2\ \ldots\ n_m\ t, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{PatternOcc\text{-}Err})$$

FIG. B.8 – Erreurs dans l'évaluation de quelques tactiques.

$$\frac{\begin{array}{c}(tac_1, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\\ \text{ou } (tac_1, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1),\ \Gamma \Vdash T = \Gamma_1 \Vdash T_1\\ \text{et } (tac_2, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\end{array}}{(tac_1\ \mathtt{Orelse}\ tac_2, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Orelse\text{-}Err})$$

$$\frac{\begin{array}{c}(tac, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\\ \text{ou } (tac, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1), \Gamma \Vdash T = \Gamma_1 \Vdash T_1\end{array}}{(\mathtt{Progress}\ tac, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Progress\text{-}Err})$$

$$\frac{\begin{array}{c}(tac_1, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\\ \text{ou } (tac_1, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1],\\ \{(n_1, \Delta[\Gamma_{11} \Vdash T_{11}]), (n_2, \Delta[\Gamma_{21} \Vdash T_{21}]), \ldots, (n_p, \Delta[\Gamma_{p1} \Vdash T_{p1}])\}, \sigma_1)\\ \text{et } \exists i.(tac_2, \Delta[\Gamma_{i1} \Vdash T_{i1}]\sigma_{si})\triangleright\mathsf{Erreur},\ i = 1 \ldots p\end{array}}{(tac_1;tac_2, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Then\text{-}Err})$$

$$\frac{\begin{array}{c}(tac_0, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\\ \text{ou } (tac_0, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1],\\ \{(n_1, \Delta[\Gamma_{11} \Vdash T_{11}]), (n_2, \Delta[\Gamma_{21} \Vdash T_{21}]), \ldots, (n_q, \Delta[\Gamma_{p1} \Vdash T_{p1}])\}, \sigma_1),\ p \neq q\\ \text{ou } (tac_0, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1],\\ \{(n_1, \Delta[\Gamma_{11} \Vdash T_{11}]), (n_2, \Delta[\Gamma_{21} \Vdash T_{21}]), \ldots, (n_p, \Delta[\Gamma_{p1} \Vdash T_{p1}])\}, \sigma_1)\\ \text{et } \exists i.((tac_i, \Delta[\Gamma_{i1} \Vdash T_{i1}]\sigma_{si})\triangleright\mathsf{Erreur},\ i = 1 \ldots p\end{array}}{(tac_0; [tac_1 | tac_2 | \ldots | tac_p], \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{ThenS\text{-}Err})$$

où $\sigma_{si} = \sigma_{11}\sigma_{21} \ldots \sigma_{(i-1)1}$

$$\frac{n > 0 \qquad (tac; \mathtt{Do}\ (n-1)\ tac, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}{(\mathtt{Do}\ n\ tac, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Do\text{-}Err})$$

$$\frac{\forall i.(tac_i, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur},\ i = 1 \ldots n}{(\mathtt{First}\ [tac_1 | tac_2 | \ldots | tac_n], \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{First\text{-}Err})$$

$$\frac{\begin{array}{c}\forall i.((tac_i, \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}\\ \text{ou } (tac_i, \Delta[\Gamma \Vdash T])\triangleright(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, \sigma_1),\ m_1 \neq \emptyset),\ i = 1 \ldots n\end{array}}{(\mathtt{Solve}\ [tac_1 | tac_2 | \ldots | tac_n], \Delta[\Gamma \Vdash T])\triangleright\mathsf{Erreur}}\ (\mathsf{Solve\text{-}Err})$$

FIG. B.9 – Erreur dans l'évaluation des tacticals.

$$\frac{(c, t, \Delta[\Gamma \Vdash T], m, p, d) \ggg \mathsf{Erreur}}{(c., t, \Delta[\Gamma \Vdash T], m, p, d) \twoheadrightarrow \mathsf{Erreur}} \ (\mathsf{Sen\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \text{ ou } m = \emptyset \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (T, \Delta[\Gamma_n]) \triangleright \mathsf{Erreur} \end{array}}{(\mathtt{Let}\ x : T, t, \Delta[\Gamma \Vdash T_1], m, p, d) \ggg \mathsf{Erreur}} \ (\mathsf{LetCur\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (T, \Delta[\Gamma_n]) \triangleright \mathsf{Erreur} \end{array}}{(\mathtt{Let}\ [?n]\ x : T, t, \Delta[\Vdash T_1], m, p, d) \ggg \mathsf{Erreur}} \ (\mathsf{LetGiv\text{-}Err})$$

FIG. B.10 – Évaluation des phrases (1/3).

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \text{ ou } m = \emptyset \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n]) \triangleright \mathsf{Erreur} \end{array}}{(\texttt{Let } x := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \blacktriangleright \mathsf{Erreur}} \ (\mathsf{LetOne_1\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n]) \triangleright \mathsf{Erreur} \end{array}}{(\texttt{Let } [?n]\ x := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \blacktriangleright \mathsf{Erreur}} \ (\mathsf{LetOne_2\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \text{ ou } m = \emptyset \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n \Vdash T]) \triangleright \mathsf{Erreur} \\ \text{ou } n = min\{i \mid (?i, b) \in m\},\ (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n \Vdash T]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_n\sigma, T_2) \end{array}}{(\texttt{Let } x : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \blacktriangleright \mathsf{Erreur}} \ (\mathsf{LetOne_3\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \in \Gamma_n \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n \Vdash T]) \triangleright \mathsf{Erreur} \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ x \notin \Gamma_n \\ \text{et } (t, \Delta[\Gamma_n \Vdash T]) \triangleright (t_2, \Delta[\Gamma_{n1} \Vdash T_2], \emptyset, \sigma) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_n\sigma, T_2) \end{array}}{(\texttt{Let } [?n]\ x : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \blacktriangleright \mathsf{Erreur}} \ (\mathsf{LetOne_4\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } \exists i.(\texttt{Let } c_i, t, \Delta[\Gamma_{i-1} \Vdash T_{i-1}], m_{i-1}, p_{i-1}, d) \blacktriangleright \mathsf{Erreur},\ i = 1 \ldots n \end{array}}{(\texttt{Let } c_1 \texttt{ And } \ldots \texttt{ And } c_n, t, \Delta[\Gamma \Vdash T], m, p, d) \blacktriangleright \mathsf{Erreur}} \ (\mathsf{LetAnd\text{-}Err})$$

où $t_0 = t,\ \Gamma_0 = \Gamma,\ T_0 = T,\ m_0 = m$ et $p_0 = p$

FIG. B.11 – Évaluation des phrases (2/3).

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright \mathsf{Erreur} \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}], m_n, \sigma) \\ \text{et } \mathsf{Erreur} = \mathsf{unicity}((m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup m_n) \end{array}}{(?n := t, t_1, \Delta[\Gamma \Vdash T], m, p, d) \rhd\!\!\rhd \mathsf{Erreur}} \;(\mathsf{Inst\text{-}Err})$$

$$\frac{\begin{array}{c} d = \mathsf{Lemma} \text{ ou } d = \mathsf{Let}_i \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright \mathsf{Erreur} \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}], m_n, \sigma) \\ \text{et } \mathsf{Erreur} = \mathsf{unify}(T_{n1}, T\sigma) \\ \text{ou } (?n, \Delta[\Gamma_n \Vdash T_n]) \in m,\ (t, \Delta[\Gamma_n \Vdash T_n]) \triangleright (t_n, \Delta[\Gamma_{n1} \Vdash T_{n1}], m_n, \sigma) \\ \text{et } \sigma_1 = \mathsf{unify}(T_{n1}, T\sigma) \\ \text{et } \mathsf{Erreur} = \mathsf{unicity}(((m - \{(?n, \Delta[\Gamma_n \Vdash T_n])\})\sigma \cup m_n)\sigma_1) \end{array}}{(?n : T := t, t_1, \Delta[\Gamma \Vdash T_1], m, p, d) \rhd\!\!\rhd \mathsf{Erreur}} \;(\mathsf{InstCast\text{-}Err})$$

$$\frac{t \neq ?1 \text{ ou } \mathsf{Erreur} = \mathsf{begin}(d)}{(\mathtt{Proof}, t, \Delta[\Gamma \Vdash T], m, p, d) \rhd\!\!\rhd \mathsf{Erreur}} \;(\mathsf{Proof\text{-}Err})$$

$$\frac{\begin{array}{c} ?i \in t \text{ ou } d \neq \mathsf{Let}_{n,b} \text{ ou } p = \lfloor\rfloor \\ \text{ou } p = \lfloor(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1)\rfloor \triangleleft p_2,\ p_1 \neq p_2 \\ \text{ou } p = \lfloor(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1)\rfloor \triangleleft p_1,\ (?n, \Delta[\Gamma_n \Vdash T_n]) \notin m_1 \end{array}}{(\mathtt{Qed}, t, \Delta[\Gamma \Vdash T], m, p, d) \rhd\!\!\rhd \mathsf{Erreur}} \;(\mathsf{Qed\text{-}Err})$$

$$\frac{\Gamma \neq \emptyset \text{ ou } ?k \in t \text{ ou } ?_{ik} \notin t \text{ ou } ?_{ik} \in T \text{ ou } m \neq \emptyset \text{ ou } p \neq \lfloor\rfloor \text{ ou } d \neq \mathsf{Lemma}_b}{(\mathtt{Save}, t, \Delta[\Gamma \Vdash T], m, p, d) \rhd\!\!\rhd \mathsf{Erreur}} \;(\mathsf{Save\text{-}Err})$$

FIG. B.12 – Évaluation des phrases (3/3).

$$
\frac{\begin{array}{c}
(T, \Delta[]) \triangleright \mathsf{Erreur} \\
\text{ou } (T, \Delta[]) \triangleright (T_0, [\Gamma_0 \Vdash s], m_0, \sigma_0),\ s \notin S \\
\text{ou } (T, \Delta[]) \triangleright (T_0, [\Gamma_0 \Vdash s], m_0, \sigma_0),\ s \in S,\ m_0 \neq \emptyset \\
\text{ou } (T, \Delta[]) \triangleright (T_0, [\Gamma_0 \Vdash s], m_0, \sigma_0),\ s \in S,\ m_0 = \emptyset \\
\text{et } \exists i.((p_i, t_{i-1}, \Delta[\Gamma_{i-1} \Vdash T_{i-1}], m_{i-1}, p_{i-1}, d_{i-1}) \twoheadrightarrow \\
\mathsf{Erreur},\ i = 1 \ldots n \\
\text{ou } (T, \Delta[]) \triangleright (T_0, [\Gamma_0 \Vdash s], m_0, \sigma_0),\ s \in S,\ m_0 = \emptyset \\
\text{et } (p_1, t_0, \Delta[\Gamma_0 \Vdash T_0], m_0, p_0, d_0) \twoheadrightarrow \\
(t_1, \Delta[\Gamma_1 \Vdash T_1], m_1, p_1, d_1) \\
\vdots \\
\text{et } (p_n, t_{n-1}, \Delta[\Gamma_{n-1} \Vdash T_{n-1}], m_{n-1}, p_{n-1}, d_{n-1}) \twoheadrightarrow \\
(t_n, \Delta[\Gamma_n \Vdash T_n], m_n, p_n, d_n) \\
\text{et } (\Gamma_n \neq \emptyset \text{ ou } ?k \in t_n \text{ ou } ?_{ik} \in t_n \text{ ou } ?_{ik} \in T_n \\
\text{ou } m_n \neq \emptyset \text{ ou } p_n \neq \lfloor\rfloor \text{ ou } d_n \neq \mathsf{Lemma})
\end{array}}{(p_1.\ \ldots\ p_n., \Delta[\Vdash T]) \downarrow \mathsf{Erreur}} \ (\mathsf{Script\text{-}Err})
$$

$$
\text{où } \Gamma_0 = \emptyset,\ p_0 = \lfloor\rfloor \text{ et } d_0 = \mathsf{Lemma}
$$

FIG. B.13 – Erreurs dans l'évaluation des scripts.

Index

Symboles

A

B

V

Bibliographie

[1] Maria-Virginia Aponte and Roberto Di Cosmo. Type isomorphisms for module signatures. In *Programming Languages Implementation and Logic Programming (PLILP)*, volume 1140 of *Lecture Notes in Computer Science*, pages 334–346. Springer-Verlag, 1996.

[2] Maria-Virginia Aponte, Roberto Di Cosmo, and Catherine Dubois. Signature subtyping modulo type isomorphisms, 1998.
http ://www.pps.jussieu.fr/~dicosmo/ADCD97.ps.gz.

[3] L. Augustsson. A compiler for lazy ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, TX, USA, August 1984. ACM Press.

[4] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2) :127–141, 1989.

[5] Lennart Augustsson. The Interactive Lazy ML System. *Journal of Functional Programming*, 3(1) :77–92, January 1993.

[6] Bruno Barras et al. *The Coq Proof Assistant Reference Manual Version 6.3.1.* INRIA-Rocquencourt, May 2000.
http ://coq.inria.fr/doc-eng.html.

[7] Janet Bertot and Yves Bertot. CtCoq : A System Presentation, 1996.

[8] Y. Bertot and L. Théry. A Generic Approach to Building User Interfaces for Theorem Provers. *J. Symbolic Computation*, 25(2) :161–194, February 1998.

[9] Yves Bertot. The CtCoq System : Design and Architecture. *Formal Aspects of Computing*, 11(3) :225–243, 1999.

[10] Nicolas Bourbaki. *Éléments de mathématique. Théorie des ensembles.* Hermann, 1966.

[11] Samuel Boutin. *Réflexions sur les quotients.* PhD thesis, Université Paris 7, Avril 1997.

[12] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, 1979.

[13] Robert S. Boyer and J. Strother Moore. A computational Logic Handbook. *PERSPEC : Perspectives in Computing*, 23, 1988.

[14] Magnus Carlsson and Thomas Hallgren. *Fudgets - Purely Functional Processes with applications to Graphical User Interfaces.* PhD thesis, Department of Computing Science, Chalmers University of Technology, March 1998.

[15] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American J. Mathematics*, 58(1) :345–363, 1936.

[16] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, volume 33, pages 134–183. Springer-Verlag LNCS, 1976.

[17] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[18] Thierry Coquand, Catarina Coquand, Thomas Hallgren, and Aarne Ranta. The Alfa Home Page, 2001.
http ://www.md.chalmers.se/~hallgren/Alfa/.

[19] Thierry Coquand and Gérard Huet. Constructions : A Higher Order Proof System for Mechanizing Mathematics. In *EUROCAL'85*, volume 203 of *LNCS*, Linz, 1985. Springer-Verlag.

[20] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. April 1995.

[21] N. G. De Bruijn. The Mathematical Language AUTOMATH, its Usage and some of its Extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[22] David Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In *Proceedings of TYPES'99, Lökeberg*, pages 131–147. Springer-Verlag LNCS, June 1999.
ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/TYPES99-SIsos.ps.gz.

[23] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/LPAR2000-ltac.ps.gz.

[24] David Delahaye and Micaela Mayero. `Field` : une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier*, pages 33–47. INRIA, Janvier 2001.
ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/JFLA2000-Field.ps.gz.

[25] Roberto Di Cosmo. *Isomorphisms of Types.* PhD thesis, Dipartimento di Informatica, Universitá di Pisa, Corso Italia - 56100 Pisa - Italy, January 1993.

[26] Roberto Di Cosmo. *Isomorphisms of Types : from λ-calculus to information retrieval and language design.* Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[27] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 366–374, San Diego, California, USA, June 1995.

[28] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. In *The Journal of Symbolic Logic*, volume 57(3), September 1992.

[29] Melvin C. Fitting. *Intuitionnistic Logic Model Theory and Forcing.* North-Holland Publishing Co. Amsterdam, 1969.

[30] John K. Foderaro, K. L. Sklower, and K. Layer. *The FRANZ LISP Manual.* University of California, Berkeley, California, 1979.

[31] D. Gabbay. The Undecidability of Intuitionnistic Theories of Algebraically Closed Fields and Real Closed Fields. In *Journal of Symbolic Logic*, volume 38, pages 86–92, 1973.

[32] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38 :173–198, 1931. [Transl. in From Frege to Gödel, van Heijenoort, Harvard Univ. Press, 1971].

[33] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. Equational Reasoning via Partial Reflection. In *Proceedings of TPHOL*. Springer-Verlag, August 2000.

[34] M. J. C. Gordon et al. A Metalanguage for Interactive Proof in LCF. In *5th POPL*, ACM, 1978.

[35] M. J. C. Gordon and T. F. Melham. *Introduction to* HOL *: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[36] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. Edinburgh LCF : a Mechanised Logic of Computation. In *Lectures Notes in Computer Science*, volume 78. Springer-Verlag, 1979.

[37] Jean Goubault-Larrecq. Théorie de la preuve et démonstration automatique. Cours de DEA, 2001.

[38] John Harrison. Metatheory and Reflection in Theorem Proving : a Survey and Critique. Technical Report CRC-053, SRI Cambridge, UK, 1995. http ://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

[39] John Harrison. HOL Light : A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166, pages 265–269. Springer LNCS, 1996.

[40] John Harrison. Proof style. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs : International Workshop TYPES'96*, volume 1512 of *LNCS*, pages 154–172, Aussois, France, 1996. Springer-Verlag.

[41] John Harrison. A Mizar mode for HOL. In *J. von Wright, J. Grundy, and J. Harrison, editors, Theorem Proving in Higher Order Logics : TPHOLs '96*, volume 1125 of *LNCS*, pages 203–220, 1996.

[42] John Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.

[43] C. Horn. The nuprl proof development system. Technical Report 214, Dept. of Artificial Intelligence, Edinburgh, 1988.

[44] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 1994.

[45] Simon Peyton Jones et al. *Haskell 98*, February 1999. http ://www.haskell.org/definition/.

[46] Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Technical Report 19, Institute for Computing Science, University of Texas at Austin, May 1988.

[47] Matt Kaufmann and J. Strother Moore. ACL2 : An Industrial Strength Version of Nqthm. In *Compass'96 : Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, USA, 1996. National Institute of Standards and Technology.

[48] Matt Kaufmann and J. Strother Moore. The ACL2 Home Page, 2001. http ://www.cs.utexas.edu/users/moore/acl2/.

[49] G. Kreisel and J.-L. Krivine. *Éléments de logique mathématique : théorie des modèles*. Dunod, 1964.

[50] Edmund Landau. *Foundations of analysis : the arithmetic of whole, rational, irrational, and complex numbers. A supplement to textbooks on the differential and integral*

calculus. Chelsea Publishing Company, third edition, 1966.
[Translated from German 'Grundlagen der Analysis' by F. Steinhardt].

[51] Pascal Lequang, Yves Bertot, Laurence Rideau, and Loïc Pottier. The Pcoq Home Page, 2001.
http ://www-sop.inria.fr/lemme/pcoq/.

[52] Xavier Leroy. *The Caml Special Light system, release 1.07 - Documentation and user's manual*. INRIA, September 1995.
[Documentation distributed with the Caml Special Light system].

[53] Xavier Leroy. *The Caml Light system, documentation and user's guide Release 0.74*. INRIA-Rocquencourt, December 1997.
http ://caml.inria.fr/man-caml/index.html.

[54] Xavier Leroy et al. *The Objective Caml system release 3.00*. INRIA-Rocquencourt, April 2000.
http ://caml.inria.fr/ocaml/htmlman/.

[55] Lena Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, 1994.

[56] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[57] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 94.

[58] Micaela Mayero. The Three Gap Theorem (Steinhauss Conjecture). In *Proceedings of TYPES'99, Lökeberg*. Springer-Verlag LNCS, 1999.
ftp ://ftp.inria.fr/INRIA/Projects/coq/Micaela.Mayero/PS/three-gap.ps.tar.gz.

[59] Micaela Mayero. *Formalisation et automatisation de preuves en analyse et en analyse numérique*. PhD thesis, Université Paris 6, Décembre 2001.

[60] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of* Standard ML. The MIT Press, February 1990. ISBN 0-262-63132-6.

[61] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of* Standard ML - *Revised*. The MIT Press, May 1997. ISBN 0-262-63181-4.

[62] Anthony P. Morse. *A Theory of Sets*. Academic Press, New York, 1965.

[63] César Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Mémoire du DEA d'informatique fondamentale, Université Paris 7, Septembre 1994.

[64] César Muñoz. *Un calcul de substitutions pour la représentation de preuves partielles en théorie de types*. Thèse de doctorat, Université Paris 7, 1997.
[Version en anglais disponible comme rapport de recherche INRIA RR-3309].

[65] Michal Muzalewski. *An Outline of PC* Mizar. Series Editor Roman Matuszewski. Fondation Philippe le Hodey, Brussels, 1993.

[66] Bengt Nordström. The ALF Proof Editor. In *Informal Proceedings of the Nijmegen workshop on Types for Proofs and Programs*, 1993.

[67] Sam Owre, Natarajan Shankar, and John Rushby. PVS : A prototype verification system. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.

[68] Larry Paulson. Interactive theorem proving with Cambridge LCF : a user's manual. Technical Report 80, Computer Laboratory, University of Cambridge, Cambridge, England, 1985.

[69] Larry Paulson. *Logic and Computation — Interactive proof with Cambridge LCF.* Cambridge University Press, 1987.

[70] Larry Paulson and Tobias Nipkow. The Isabelle Home Page, 2001. http ://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html.

[71] Lawrence C. Paulson. Isabelle : The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[72] Randy Pollack. The Theory of Lego. Manuscrit, 1988.

[73] D. Putnam and R. Davis. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :201–215, 1960.

[74] L. Quam, W. Diffie, and L. Manual. Stanford LISP 1.6 Manual. Technical Report 28.7, Stanford Artificial Intelligence Laboratory, Stanford, California, 1972.

[75] A. Ranta and P. Mäenpää. The type theory and type checker of GF. In *Colloquium on Principles, Logics, and Implementations of High-Level Programming Languages, Workshop on Logical Frameworks and Meta-languages*, Paris, France, September 1999.

[76] Daniel de Rauglaudre. *Camlp4 - Reference Manual version 3.02.* INRIA-Rocquencourt, July 2001.
http ://caml.inria.fr/camlp4/manual/.

[77] Daniel de Rauglaudre. *Camlp4 - Tutorial version 3.02.* INRIA-Rocquencourt, July 2001.
http ://caml.inria.fr/camlp4/tutorial/.

[78] Sergei Romanenko, Claudio Russo, and Peter Sestoft. The Moscow ML Home Page, 2001.
http ://www.dina.kvl.dk/~sestoft/mosml.html.

[79] Piotr Rudnicki. An Overview of the Mizar Project. In *Proceedings of the Workshop on Types for Proofs and Programs*, Bastad, Sweden, 1992.

[80] Sergei Soloviev. The category of finite sets and cartesian closed categories. In *Journal of Soviet Mathematics*, volume 22(3), pages 154–172, 1983.

[81] Guy L. Steele Jr. Common Lisp *: The Language.* Digital Press, 1984.

[82] Don Syme. *Declarative Theorem Proving for Operational Semantics.* PhD thesis, University of Cambridge, 1998.

[83] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948 ; prepared for publication by J. C. C. McKinsey.

[84] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.0.* INRIA-Rocquencourt, April 2001.
http ://coq.inria.fr/doc-eng.html.

[85] Bell Laboratories (Lucent Technologies), Princeton University, Yale University (The FLINT Project), and AT&T Research. The Standart ML of New Jersey Home Page, 2001.
http ://cm.bell-labs.com/cm/cs/what/smlnj/index.html.

[86] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 120–129, 1992.

[87] Andrezej Trybulec. Tarski Grothendieck Set Theory. In *Formalized Mathematics*, volume 1, pages 9–11, 1990.

[88] Andrzej Trybulec. The Mizar-QC/6000 logic information language. In *ALLC Bulletin (Association for Literary and Linguistic Computing)*, volume 6, pages 136–140, 1978.

[89] P. Weis, M. V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML Reference Manual. Technical Report 121, INRIA, 1990.

[90] Benjamin Werner. *Une théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[91] Freek Wiedijk. Checker. Manuscrit
http ://www.cs.kun.nl/~freek/mizar/by.ps.gz.

[92] Freek Wiedijk. Mizar : An impression. Manuscrit
http ://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz.

[93] Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent, Canterbury, October 1998.

[94] Ernst Zermelo. Untersuchungen ueber die Grundlagen der Mengenlehre. In *Math. Annalen*, volume 65, pages 261–281, 1908.

Résumé. Nous nous proposons d'étudier deux catégories distinctes de langages intervenant dans les outils d'aide à la preuve : les langages de preuves, permettant d'exprimer les déductions formelles des démonstrations, et les langages de tactiques, qui, dans le cadre de systèmes à la LCF, permettent d'étendre l'automatisation. Concernant les langages de preuves, une étude comparée des styles procéduraux, déclaratifs, et à base de termes (isomorphisme de Curry-Howard) nous permet de comprendre l'adéquation de ces styles avec certaines situations spécifiques de preuves. Dans l'optique de bénéficier des avantages de ces trois styles dans toute sorte de preuves, nous proposons, dans le cadre du système Coq, un nouveau langage, appelé $\mathcal{L}_{pdt}$, et destiné à réaliser la fusion de ces trois styles. À propos des langages de tactiques, nous mettons en évidence qu'un métalangage Turing-complet peut se révéler inapproprié, notamment pour traiter de petites automatisations ponctuelles. Pour pallier ce problème, nous présentons, toujours dans le cadre de Coq, un nouveau langage, que nous appelons $\mathcal{L}_{tac}$ et qui élabore un réel lien entre le langage objet et le métalangage complet du système. Comme un bonus, on remarque que des automatisations non triviales peuvent être codées avec $\mathcal{L}_{tac}$. C'est, en particulier, le cas de la tactique `Field`, qui permet de décider d'égalités sur les corps commutatifs. Enfin, nous décrivons deux outils dédiés à $\mathcal{L}_{tac}$, à savoir un système de quotations permettant d'interfacer $\mathcal{L}_{tac}$ avec le métalangage complet (Objective Caml), et un debugger visant à mieux contrôler les automatisations codées en $\mathcal{L}_{tac}$.

Mots-clés: Outils d'aide à la preuve, Langage de preuves, Style de preuves, Langage de tactiques, Métalangage, Automatisation, Système Coq

Abstract. We study two different kinds of languages involving in proof assistants : proof languages, which allow us to express the proof formal deductions steps, and tactic languages, which, in the context of LCF-like systems, allow us to extend automation. Regarding proof languages, a comparison between procedural, declarative and term based (Curry-Howard isomorphism) styles allows us to understand how these styles can be suitable in some specific proof situations. In order to benefit the advantages of these three styles in every kind of proof, we propose, in the context of the Coq system, a new language, called $\mathcal{L}_{pdt}$, and intended to make the fusion of these three styles. Regarding tactic languages, we show that a Turing-complete metalanguage may be inappropriate in some cases, e.g. when dealing with small and local automations. To solve this problem, we present, always in the context of Coq, a new language, we call $\mathcal{L}_{tac}$ and which builds a real link between the object language and the full programmable metalanguage of the system. As a bonus, we notice that non trivial automations can be coded using $\mathcal{L}_{tac}$. In particular, it is the case of the tactic `Field`, which allows us to decide equalities on commutative fields. Finally, we describe two tools dedicated to $\mathcal{L}_{tac}$: a quotation system, which allows us to interface $\mathcal{L}_{tac}$ with the complete metalanguage (Objective Caml), and a debugger, to better control the automations written with $\mathcal{L}_{tac}$.

Keywords: Proof assistants, Proof language, Proof style, Tactic language, Metalanguage, Automation, Coq system