Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML

Sylvain Baro

UNIVERSITÉ PARIS 7 - DENIS DIDEROT – UFR D'INFORMATIQUE
ANNÉE 2003

Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML

THÈSE

pour l'obtention du diplôme de
docteur de l'université Paris 7, spécialité informatique

présentée et soutenue publiquement par

Sylvain Baro

le 10 juillet 2003

DIRECTEUR DE THÈSE

Pascal Manoury

RAPPORTEURS

Mme. Catarina Coquand
M. René David
M. Christophe Raffalli (co-rapporteur)

JURY

Mme. Véronique Donzeau-Gouge (président)
Mme. Catarina Coquand
M. Guy Cousineau
M. Hugo Herbelin
M. Pascal Manoury
M. Christophe Raffalli

2

Remerciements

Cette partie est probablement la plus difficile à rédiger dans un manuscrit de thèse, et cela pour deux raisons : tout d'abord, c'est la plus personnelle ; mais surtout c'est la seule partie qui va être consciencieusement lue par la famille, les amis ainsi que les collègues de bureau.

Tout d'abord, je souhaite remercier Pascal Manoury, mon directeur de thèse, pour sa patience et pour m'avoir soutenu durant ces années. Je dois également avouer que si ce document est lisible, c'est en grande partie grâce à ses talents de relecteur.

Je tiens également à remercier Emmanuel Chailloux, dont le soutien m'a été précieux, particulièrement lors des difficiles débuts de rédaction. Ses conseils avisés et ses efforts de relecture m'ont permis "d'amorcer la pompe".

Je remercie particulièrement Catarina Coquand et Christophe Raffalli pour avoir bien voulu rapporter mon manuscrit et pour le travail important qu'ils ont fourni pour cela. Je remercie également les membres de mon jury. Parmi les membres du jury, je tiens à remercier Guy Cousineau pour le soutien qu'il m'a apporté en tant que directeur du DEA SPP.

Je suis également redevable à mes collègues de bureau (et néanmoins amis) qui ont participés à la relecture du présent document : Anne-Gwenn Bosser, François Maurel et Raphaël Montelatici. Je m'engage donc ici à relire des chapitres de leurs thèses (au moins les parties pas trop techniques).

Rien de tout cela n'aurait évidemment pu être fait sans le support amical, logistique et administratif d'Odile Ainardi, Noëlle Delgado et Dominique Raharijesy.

Les membres du groupe de travail "assistant de preuves", m'ont été d'une grande aide, particulièrement Yves Legrandgérard qui a écrit l'interface graphique et rédigé le pré-RFC du protocole de communication, mais également Michel Parigot, Paul Rozière et Marianne Simonot.

Ces années n'auraient pas été aussi plaisantes sans la présence des thésards (et ancien thésards) PPS : Vincent Balat, Emmanuel Beffara, Feng Yangyue, Samuel Lacas (grand ancien), Michel Hirschowitz, Benjamin Leperchey, Jean-Vincent Loddo et Emmanuel Polonovski.

Je ne peux ni passer sous silence l'ambiance sympathique de l'équipe PPS, ni malheureusement en citer tous les membres, an-

ciens membres et visiteurs. Mais vous avez tous une part de responsabilité dans le fait que je puisse enfin soutenir ma thèse.

Pour finir, je tiens à remercier mes parents, pour leur soutien. Et surtout, je suis redevable de beaucoup à Juliette Ricou, pour sa patience souvent mise à l'épreuve (y compris sur la durée) et son soutien quotidien, ce qui n'a probablement pas toujours été facile.

Résumé

Pouvoir vérifier la conformité d'un programme avec sa spécification représente un enjeu important. Pouvoir le faire à l'aide d'une machine offre en plus la possibilité de *vérifier cette vérification*. On utilise pour cela un *assistant de preuve* permettant la description du problème, la construction des preuves et leur vérification.

Plusieurs approches existent pour parvenir à cette fin : engendrer le programme à partir de sa spécification ; utiliser un programme annoté avec sa spécification et des éléments de preuves pour construire la preuve complète de correction ; ou encore partir de la spécification et du programme, puis montrer la conformité du second vis-à-vis du premier. C'est cette dernière approche que nous avons choisi.

Dans le système que nous avons implémenté, l'utilisateur décrit la spécification du programme dans un formalisme logique *ad hoc*, puis donne le programme dans le sous-ensemble fonctionnel de ML (comprenant filtrage, définitions récursives et fonctions partielles), puis construit interactivement les preuves de correction nécessaires pour prouver la validité du programme.

Le travail que nous avons fourni se découpe en trois volets :

- la formalisation d'une logique dédiée à la preuve de programmes dans le fragment fonctionnel de ML ;
- la spécification détaillée du moteur de preuve, de son interface, et du protocole qu'ils utilisent pour communiquer ;
- l'implémentation du moteur de preuve en *Objective Caml*, qui utilise une architecture particulièrement originale, mélangeant programmation objet et programmation fonctionnelle.

Tous ces éléments sont présents dans ce document, y compris une description détaillée de l'implémentation, et des raisons qui nous ont poussés à effectuer certains choix, plutôt que d'autres. Le lecteur y trouvera une description de l'utilisation de notre système, ainsi que des exemples de mise en œuvre de celui-ci.

6

Abstract

One should not underestimate the import of being able to verify the conformance between a program and its specification. Furthermore, being able to verify it using a dedicated program allows one to check the correctness of this verification. For this purpose, one uses *proof assistants*, which are programs that allows to describe the problem, build proofs, then check them.

There are more than one way to achieve this goal : one could either generate the program from its specification ; one could use an annotated program which carries the specification and hints on how to prove the conformance, then check it afterwards ; or one could start from the specification and the program, then prove the conformance of the latter with respect to the first. For our system, we chose the last approach.

We implemented a system in which the user describes the specification of a program in a dedicated logical framework, then writes the program in the ML programming language, restricted to the functional subset (with pattern matching, inductive definitions and partial functions), then, after all, interactively builds the proof of conformance of the program with respect to its specification.

There are three different aspects in our work :

- formalization of a logical framework dedicated to the verification of programs written in the functional part of ML ;
- precize specification of the proof assistant, its user interface, and the protocol used by both in order to communicate ;
- the implementation of the proof assistant in *Objective Caml*, using an original architecture which mixes object oriented programming and functional programming.

All these elements may be found in this document, including a precize description of the implementation, the choices we made and the reasons of these choices. The reader will also find here a description on how to use our system, and some examples of problems handled with it.

Table des matières

Introduction

The Art of Computer Programming *is, however, still a work in progress.*
Donald E. Knuth

La première apparition écrite connue du mot “bug” dans son acception informatique date du 9 septembre 1947 et correspondait à la découverte d'une mite ayant produit une défaillance dans les circuits d'un ordinateur[1] [57]. Au fur et à mesure de la progression de l'informatique dans la société, que cela soit à travers les ordinateurs ou les systèmes embarqués, ce mot a fait son chemin à tel point qu'il y a probablement plus de personnes pouvant en donner une définition que de personnes pouvant définir ce qu'est un système d'exploitation.

Les erreurs dans les programmes (nous ne parlerons pas ici des erreurs dues au défaillances du matériel), peuvent être une gêne, mais peuvent également avoir des conséquences catastrophiques, que cela soit dans un ordinateur (destruction de fichiers, panne de central téléphonique), ou dans un système embarqué (véhicule de transport en commun). Cependant, même une erreur dans un logiciel simple – par exemple le système de contrôle d'une machine à laver – peut avoir des répercussions importantes pour le fabricant (échanger les circuits de plusieurs milliers de machines à laver déjà vendues). Il convient de distinguer les erreurs de conceptions des erreurs de mise en œuvre : soit le programme est incorrect parce qu'il ne correspond pas à son cahier des charges, soit parce que ce cahier des charges est incomplet ou incohérent, ou encore parce que ce cahier des charges ne correspond pas à la volonté du commanditaire du programme.

[1] Cependant, le mot semble avoir été employé bien avant pour des appareils électriques, ou pour décrire les parasites dans une ligne télégraphique.

* * *

Un grand nombre de moyens sont appliqués aujourd'hui pour tenter d'éliminer ces erreurs. Cela va de l'utilisation de langages de programmation sophistiqués, détectant plus d'erreurs à la compilation des programmes (ML, ADA, Java. . .), à la rédaction de standards méthodologiques pour le développement, requérant le respect de procédures précises pour les phases de spécifications, tests et intégration (par exemple la norme de qualité du ministère de la défense des États-Unis, souvent citée comme référence dans l'industrie [26]), ou encore l'utilisation de normes de "style" pour les programmeurs d'une entreprise, de manière à interdire certaines constructions complexes, sources d'erreur, ou risquant d'être mal interprétées par une personne souhaitant modifier ou corriger le programme.

Les programmes sont en général testés, et chaque partie du programme doit également être vérifiée avant d'être intégrée à l'ensemble des travaux. Le problème est que les logiciels ne peuvent pas être testés exhaustivement et ceci est également vrai pour chaque partie simple de ceux-ci. Prenons deux exemples : d'abord une fonction concaténant deux listes d'entier. Il existe une infinité de listes d'entiers possibles (même si dans le cas d'un programme, ce nombre est fini, bien que très grand). Il n'est bien sûr pas possible de tester ce programme avec toutes les listes. On doit donc se limiter au test d'un certain nombre de cas "normaux" et de cas "limites" (concaténation de deux listes vides, une liste vide et une liste non vide, etc.). On n'obtient donc pas une *garantie* de la correction de la fonction, mais plutôt une *forte présomption*. Un autre exemple est celui d'un programme simple interagissant avec l'utilisateur. Dans ce cas, le problème est d'essayer toutes les suites de commandes pouvant être entrées par celui-ci. On se retrouve de nouveau avec un nombre effrayant de possibilités. Il est possible de faire l'analogie avec l'analyse qui serait nécessaire pour examiner tous les coups possibles et leurs conséquences dans une partie d'échec.

Ces méthodes (tests, méthodologie précise, etc.) ne permettent pas de garantir qu'il n'y pas d'erreur dans le programme, mais peuvent servir à en détecter ou à en éviter un certain nombre. Cela est satisfaisant lorsqu'il s'agit de programmer des logiciels de moindre importance mais insuffisant pour des logiciels ou des

fragments de logiciels *critiques*. Nous entendons par là que la défaillance de ceux-ci aurait des conséquences graves. Cette notion de “critique” a une grande importance. Il est intéressant de découper un programme de manière à isoler les zones critiques de celui-ci, cela permet de les vérifier avec plus d'attention et des outils plus sûrs, bien que plus contraignants et plus complexes à mettre en œuvre.

Par exemple dans le cas d'un traitement de texte, la partie critique est celle qui s'occupe de sauvegarder le document, y compris en cas d'arrêt imprévisible du logiciel. Ainsi, même si cet arrêt représente une nuisance pour l'utilisateur, son travail ne sera jamais perdu. Pour un métro automatique, une des sections critiques du système de contrôle est le dispositif chargé de freiner en cas d'obstacle sur la voie.

* * *

Il n'est pas possible de vérifier qu'un programme est correct dans l'absolu, il est donc nécessaire d'avoir une *spécification* du problème, c'est-à-dire un document décrivant précisément ce qui doit être fait par le programme, les entrées qu'il est susceptible de recevoir et les réponses qu'il doit fournir.

L'objectif que nous nous fixons est d'arriver à vérifier qu'un programme donné satisfait sa *spécification formelle*. La spécification formelle est une réécriture du cahier des charges dans un langage mathématique formel. À travers la *sémantique* du langage de programmation utilisé – qui est une formalisation mathématique des constructions et fonctions primitives du langage – on peut représenter le programme comme un objet mathématique, dont on peut alors prouver des propriétés. Celles-ci sont issues de la spécification du programme et pourront être par exemple, que le programme ne se bloque jamais, qu'il renvoie toujours une valeur vérifiant certains critères, etc.

Si les preuves servant à vérifier ces programmes sont effectuées à la main, il est encore possible que des erreurs s'y soient glissées. Pour éviter cela, on utilise des *systèmes d'aide à la preuve* ou *assistants de preuve*. Ces systèmes sont des logiciels permettant à l'utilisateur d'énoncer des définitions, axiomes, théorèmes, puis de construire les preuves nécessaires pour vérifier ses affirmations. Le système est chargé de garantir la correction des étapes

de preuves et de vérifier que tout ce qui est énoncé est justifié, mais aussi d'assister l'utilisateur dans la construction des preuves. Ces assistants de preuves (que nous noteront AP) existent depuis la fin des années 60 (Automath, de De Bruijn, voir [14]). Nous verrons dans le chapitre 1 les contraintes que nous imposent l'usage de tels systèmes.

* * *

Avant d'aller plus loin, il est nécessaire de préciser que dans le cycle de vie du logiciel, nous ne nous intéressons qu'aux parties spécification, implémentation et test. Du point de vue de la maintenance du logiciel, on peut seulement dire qu'elle devrait être facilitée si le programme correspond à sa spécification, mais le fait de modifier directement une partie de celui-ci conduit nécessairement à refaire un travail de vérification formelle dessus si l'on souhaite encore en garantir la correction. Cependant, si l'on respecte une architecture modulaire, seules les preuves touchant directement à la partie modifiée seront à refaire.

La réutilisation de composants logiciels est également facilitée si ceux-ci ont des spécifications complètes, qu'ils les respectent et que cela est prouvé. Si une portion de programme doit être réutilisée de nombreuse fois, une erreur dans celle-ci pourra être très préjudiciable. D'autre part, le fait d'avoir une spécification formelle diminue considérablement le risque d'avoir des interactions "parasites" entre deux composants, les conditions d'utilisation de chacun étant nécessairement explicites.

Présentation des travaux

Les travaux décrits dans ce document présentent deux aspects :

1. La formalisation d'un système logique orienté vers la spécification et la preuve de correction de programmes fonctionnels écrits en ML.
2. La conception de l'architecture générale d'un système interactif d'aide à la preuve formelle qui a servi de support à l'implantation de notre système dédié à la preuve de programmes.

Ces deux domaines sont indépendants et représentent chacun une contribution aux domaines

1. de la preuve formelle de programme,
2. de la preuve assistée par ordinateur.

Notre système de preuve de programmes est formalisé dans un calcul des prédicats du second ordre, dont les objets sont les programmes et les données qu'ils manipulent. C'est son originalité majeure : le langage de termes est le langage des expressions ML, ce qui comprend les types algébriques, les fonctions définies par filtrage, la récursivité et le polymorphisme.

La définition des types récursifs ML est interprétée à la manière de la théorie des types comme la définition récursive d'un prédicat en terme de règles d'introduction (les constructeurs des valeurs du type) et d'élimination (la règle de récurrence structurelles sur les valeurs du type). Notre langage logique autorise le polymorphisme paramétrique des types.

La terminaison étant une propriété fondamentale des fonctions, nous avons inclus dans le système logique un *prédicat de terminaison* que nous présentons comme un *prédicat de typage renforcé*. Cela nous permet de dissocier la propriété syntaxique de typage au sens ML (que nous vérifions automatiquement à l'aide de l'algorithme d'inférence de type usuel) et la propriété logique de terminaison ou totalité des fonctions, qui peut ne pas être requise, et pour laquelle il n'existe pas d'algorithme complet.

Enfin, nous avons réservé un sort particulier au prédicat d'égalité. La relation d'équivalence correspondant à l'évaluation des programmes ML est intégrée à notre système logique et tient ici lieu de théorie équationnelle.

Le second aspect de notre travail dépasse le cadre de la preuve de programmes ML pour aborder le problème de la conception et de l'implémentation des systèmes d'aide à la preuve en général, pour lesquels nous présentons une approche que nous pensons originale. Notre ligne de conduite pour ce travail a été le souci de pouvoir offrir à l'utilisateur une interface d'édition "pleine page". Ceci a pour effet de nous amener à concevoir l'architecture de notre système comme un dialogue entre un "moteur" chargé de la partie logique du processus d'édition (en particulier, la validité des étapes de preuves) et une interface présentant à l'utilisateur l'état de son développement et prenant en charge les outils d'édition proprement dits (ajout, suppression, etc.). Le mode de dialogue suit le modèle client-serveur pour lequel un protocole dédié a été défini. Cet as-

pect de notre travail a été mené en collaboration avec le groupe de travail MathOS du laboratoire PPS.

Nous avons également mis l'accent sur les possibilités d'extensions de notre système au niveau des tactiques de preuve, et cela nous a amené à concevoir pour le moteur de preuve une architecture particulièrement originale, mêlant programmation objet et programmation fonctionnelle en *Objective Caml*.

Plan du document

Tous aspects les présentés plus haut sont présents dans ce document. Dans le chapitre 1, nous introduirons les systèmes d'aide à la preuve et les concepts utilisés (le lecteur familier avec ce domaine pourra passer directement au chapitre suivant). Nous présenterons également quelques-uns des systèmes, choisis en fonction de leur intérêt dans le cadre qui nous intéresse. Dans le chapitre 2, nous exposerons les motivations qui nous ont conduit à effectuer le présent travail. Nous situerons ce travail par rapport à l'existant. La formalisation du système de preuve, décrivant les interactions entre "le monde des preuves" et "le monde des programmes" apparaît dans le chapitre 3, où nous décrirons la logique du système, c'est-à-dire ses fondements mathématiques et les objets manipulés dans celui-ci. Les aspects spécification et implémentation évoqués dans la section précédente sont décrits dans les chapitres 4 et 5. Dans le chapitre 4, nous aborderons la spécification de l'architecture interne de notre moteur de preuve, ainsi que de son mécanisme d'interface utilisateur. Nous décrirons ensuite l'implémentation effectuée de ce système dans le chapitre 5. Enfin, nous terminerons en décrivant l'état actuel de notre système et en donnant deux exemples de problèmes traités avec celui-ci.

Chapitre 1

Les systèmes d'aide à la preuve

Il existe actuellement un grand nombre d'*Assistants de Preuve* (AP), certains étant pourvus d'une importante communauté d'utilisateurs (académiques et industriels), d'autres expérimentaux et restant à l'intérieur d'un cercle restreint d'utilisateurs. Mais dans tous les cas, l'utilisation en est réservée de fait aux spécialistes, c'est-à-dire à des personnes possédant un bagage mathématique suffisant — qui plus est, dans le domaine des mathématiques formelles et de la logique — et, lorsqu'il s'agit de faire de la preuve de programmes, d'un bagage informatique (algorithmique et parfois sémantique des langages de programmation).

Nous commençons par expliquer ce que sont et ne sont pas les AP, et nous effectuons une brève présentation des mathématiques formelles qui nous seront utiles. Nous passons en revue les différentes utilisations possibles des systèmes de preuve. Nous esquissons ensuite une description des principales familles de systèmes existants.

1.1 Qu'est-ce qu'un assistant de preuve ?

1.1.1 Notions préliminaires

Les assistants de preuve reposant sur les mathématiques formelles, il est utile de présenter succinctement quelques notions nécessaires.

Les travaux de formalisation des mathématiques ont deux aspects, d'un côté formaliser le raisonnement et le langage mathématique, d'un autre formaliser les briques de base sur lesquelles faire reposer les mathématiques (fondements des mathématiques). Ces deux niveaux apparaissent dès l'antiquité, avec notamment du côté de la formalisation du langage, les travaux d'Aristote, et du point de vue des objets manipulés, les travaux d'Euclide.

1.1.1.1 Variables et fonctions

Les notions de variables et de fonctions prennent de l'importance au dix-huitième siècle, avec Leibniz et Newton. Une variable est susceptible de dénoter une inconnue, que l'on cherche à calculer, une valeur générique ("soit x un entier, montrons que x est pair ou x est impair"), ou encore un paramètre d'une fonction. Les fonctions peuvent être considérées selon deux points de vue : le point de vue intentionnel, c'est-à-dire le *procédé* "comment calcule-t-on une valeur à partir des arguments" et le point de vue extensionnel (qui ne se préoccupe pas du calcul, mais uniquement du graphe de la fonction).

1.1.1.2 Prédicats et types

On distinguera deux notions *a priori* assez proches, les notions de prédicats et de types. Un prédicat dénote une propriété sur un objet ("être pair"). Les types dénotent la classe d'objets vérifiant une propriété. A chaque valeur du langage, on associera un type : 3 est de type nat (le type des entiers naturels), et l'addition sur les entiers naturels est de type $nat * nat \rightarrow nat$ (prend deux entiers et renvoie un entier comme résultat). Les types permettent de restreindre les formules "légales" : on ne peut donc pas additionner un entier et

une fonction. La théorie des types de Russell-Whitehead qui essaie de pallier au paradoxe de Russell a pour but d'empêcher d'écrire dans un langage mathématique formel certaines expressions amenant une incohérence, comme par exemple "x est élément de x"[1].

1.1.1.3 Termes et formules

On peut diviser le langage mathématiques en quatre parties (la quatrième, les preuves sont traitées dans la section suivante) :

- les termes, qui dénotent les objets manipulés (entiers, fonctions, variables d'*individus*) ;
- les formules qui permettent d'exprimer les propriétés sur ces termes, à l'aide de connecteurs (*et*, *ou*, *implique*. . .) et de quantificateurs (*pour tout*, *il existe*) ;
- le langage vernaculaire, permettant d'exprimer que telle formule est un théorème, de poser des définitions, des axiomes et de déclarer une variable.

1.1.1.4 Preuves

La preuve d'une proposition est une justification de celle-ci, sous la forme d'un raisonnement dont chaque étape est irréfutable, permettant de convaincre de la véracité de la proposition.

Une preuve *formelle* est une preuve exprimée dans un langage formel (c'est-à-dire dont la syntaxe est complètement définie, et ne permet aucune ambiguïté) et où l'irréfutabilité des étapes de cette preuve repose sur le fait que chaque étape de cette preuve peut être vérifiée syntaxiquement. On utilise un ensemble de *règles de déduction* établissant quelles sont les étapes admises dans la preuve. Cela passe donc par une formalisation du raisonnement mathématique[2].

Lorsqu'un énoncé mathématique est prouvé formellement, il est donc possible de remonter en suivant la ou les preuves jusqu'aux "bases" de la théorie dans laquelle on travaille.

[1] Cette expression est à la base du paradoxe de Russell, qui utilise "l'ensemble des ensembles qui ne se contiennent pas eux-mêmes".

[2] On notera que le terme *formel* mathématique et le terme *formel* juridique n'ont pas du tout le même sens.

Un théorème est un énoncé pour lequel il existe une preuve.

1.1.1.5 Lambda-calcul

Le *lambda-calcul*, inventé par Church, est avant tout une formalisation de l'écriture des fonctions. On écrira par exemple $\lambda x.\lambda y.x$ la fonction qui prend deux arguments (x et y) et qui renvoie comme résultat le premier (la valeur de x). Un terme du lambda-calcul est soit une variable, soit l'application de deux termes (notée par la juxtaposition) $f\ x$ signifiant "f appliqué à x", et l'abstraction d'une variable $\lambda x.t$ signifiant "la fonction qui à x associe t", où x peut apparaître dans t. On ajoute à cela des parenthèses pour lever les ambiguïtés.

Cette notation ne serait pas très utile sans la règle de calcul qui lui est associée : la *bêta-réduction*. Cette règle de calcul est purement syntaxique. Elle exprime qu'un terme de la forme $(\lambda x.t)\ u$ (la fonction qui à x associe t, appliquée à u), se calcule en t, dans lequel on aura substitué u à chaque occurrence de x. Par exemple, $(\lambda x.\lambda y.x)\ t_1\ t_2$ donne d'abord $(\lambda y.t_1)\ t_2$, puis t_1 (si y n'apparaît pas dans t_1).

À travers des codages, il est possible d'exprimer des données et les opérations sur ces données. Par exemple, on codera le 0 par $\lambda x.\lambda f.x$, le 1 par $\lambda x.\lambda f.(f\ x)$, le 2 par $\lambda x.\lambda f.(f(f\ x))$, etc.

Le lambda-calcul étant en fait "trop expressif" (on peut appliquer n'importe quelle fonction, voire quelque chose qui n'est pas une fonction, à n'importe quoi), il est possible de le *typer*, c'est-à-dire de déterminer le type d'un terme de façon à savoir d'une part s'il est correct, d'autre part dans quel contexte l'utiliser. Cela permet de restreindre le langage à une certaine classe de fonctions (par exemple les fonctions totales). On écrira par exemple $\lambda(f : \iota \to \iota)\lambda(x : \iota).(f\ x)$ (où ι est un type arbitraire) est une fonction prenant en argument f (une valeur fonctionnelle) et x et renvoyant le résultat de l'application $f\ x$, lui même de type ι. On ne pourra par plus écrire $x\ x$ (x appliqué à lui-même), car on ne saura pas le typer. Deux grandes familles de méthodes existent pour typer le lambda-calcul : à la Church, en contraignant les variables par un type (comme ci-dessus), ou à la Curry, en cherchant à déterminer le type sans que le terme soit annoté.

Il est courant d'ajouter des extensions au lambda-calcul, que cela soit des types (entiers, booléens, listes), ou des combinateurs (récurseurs, analyse par cas, point fixe) avec leurs règles de calcul.

Par rapport à la vision habituelle des fonctions en mathématiques, le lambda-calcul offre l'avantage de mettre en évidence le "procédé" de calcul. La fonction est donc définie par son *algorithme*. C'est pour cette raison que ce langage est adapté à la représentation de programmes informatiques. Plusieurs langages informatiques dérivent directement du lambda-calcul : tout d'abord LISP et Scheme (lambda-calcul non typé), mais aussi les langages de la famille ML (lambda-calcul typé) qui nous intéressent particulièrement et sur lesquels nous reviendrons à la section 2.1.

1.1.1.6 Isomorphisme de Curry-Howard

L'isomorphisme de Curry-Howard met en évidence la similarité entre les preuves et les lambda-termes, notamment il est possible de "calculer" avec des preuves comme avec des lambda-termes. Par exemple une preuve de $A \to B$ est considérée comme une fonction qui à toute preuve de A associe une preuve de B. De plus, on peut noter les preuves de la même façon que les fonctions en utilisant les termes du lambda-calcul.

1.1.2 Généralités sur les systèmes de preuve

Les assistants de preuve[3] sont des logiciels issus du mariage de ces formalisations des mathématiques et de l'informatique. L'ordinateur est en effet un outil très efficace lorsqu'il s'agit de manipuler des symboles et de calculer. C'est insuffisant pour faire des mathématiques, car pour cela, il est primordial de savoir *raisonner*, c'est-à-dire de choisir un chemin particulier pour atteindre son but, lorsqu'il n'est pas possible de les parcourir tous un par un jusqu'à trouver le bon. Mais autant il est nécessaire d'*imaginer* un nouveau théorème ou la preuve d'un théorème, autant la vérification scrupuleuse d'une suite d'énoncés mathématiques accompagnés

[3]Plus connus sous le nom de *theorem provers* ("prouveurs" de théorèmes, en général les systèmes aidant l'utilisateur à faire les preuves) ou *proof checkers* ("vérificateurs" de preuve) prenant en entrée les preuves et en vérifiant la correction

de leurs justifications est une tâche ne requérant aucune imagination, mais seulement de la rigueur (et du temps). C'est donc une tâche qui peut être laissée aisément à un logiciel adéquat, pour peu que l'on représente la preuve comme un objet formel (muni d'une syntaxe et de règles de constructions précises). La vérification par une machine a l'avantage d'être plus rapide et plus sûre.

Un assistant de preuve est un environnement de travail permettant à un utilisateur de formaliser un "développement mathématique" en lui fournissant des outils pour l'aider (automatisation partielle, base de connaissances) mais surtout, pour garantir que tout ce qui a été fait est correct. Nous revenons en 2.5 sur la correction.

1.1.3 Interaction

Les premiers systèmes de preuves n'étaient pas interactifs et étaient utilisés pour vérifier des fichiers contenant le texte formel des propositions et des preuves. Maintenant, la plupart des systèmes sont interactifs et peuvent être vus comme l'adversaire dans un jeu où l'utilisateur essaierait de le convaincre de la justesse de son argumentation. Cela prend la forme d'un dialogue de commandes de l'utilisateur et de réponses du système :
Utilisateur : Je pense que la proposition Φ est vraie, en supposant $y = 2\ T$.
AP : Mais il reste à me prouver Φ' et Φ''
Utilisateur : Φ' est trivial, mais pour Φ'', j'utilise une récurrence sur x. etc.

Le système aide donc l'utilisateur en vérifiant que tous les cas ont bien été traités, mais également en lui construisant les formules qu'il reste à prouver et en lui donnant la liste des hypothèses pertinentes en un point de la preuve. On notera cependant que l'adversaire a une stratégie déterministe. Ce jeu se situe donc plutôt dans la catégorie des jeux à un joueur où l'on joue contre les règles (réussites).

La preuve est constituée par l'enchaînement (suite ou arbre) des choix de l'utilisateur. Celle-ci est terminée lorsque le système n'a plus de questions.

Tous les systèmes interactifs ne se présentent pas de cette fa-

çon, notamment ACL2, dont le mode d'interaction est différent (nous en verrons plus section 1.3.6).

1.1.4 Ingrédients

On peut distinguer deux parties dans le système, la partie que nous appelons *moteur de preuve* qui prend en charge la construction et la vérification des preuves et de l'environnement (c'est-à-dire l'ensemble de définitions, déclarations, axiomes et théorèmes), la seconde partie étant l'*interface utilisateur*, qui est toute aussi importante et qui gère en fait le dialogue entre le moteur de preuve et l'utilisateur.

Il est nécessaire d'avoir un langage pour communiquer avec la machine, celle-ci étant encore difficilement capable d'interpréter la langue naturelle et son cortège de sous-entendus et d'imprécisions. Le langage doit permettre d'exprimer des faits et des suppositions (on utilise pour cela des formules), mais il doit aussi pouvoir exprimer complètement des preuves en en détaillant chacune des étapes. Dans la plupart des systèmes, ces étapes de preuve sont les règles logiques de base (*modus ponens*, axiome, etc.) mais aussi des étapes plus complexes généralement appelées *tactiques*. Aux formules et étapes de preuve, il faut encore ajouter le vocabulaire permettant d'énoncer des axiomes ou des théorèmes, de définir des notions, ou de déclarer des constantes, que l'on appelle langage *vernaculaire* [22].

Parallèlement au langage, il faut également choisir un formalisme mathématique, c'est-à-dire une description des objets les plus élémentaires et de ce qui permet de les manipuler. C'est en quelque sorte le fondement des mathématiques existant dans le système de preuve, et nous ne pouvons rien décrire qui ne sera pas exprimable dans ce formalisme. Par exemple, on peut choisir la théorie axiomatique des ensembles de Zermelo-Frænkel [35], et en partant de cette théorie, tout doit être exprimé en terme d'ensembles, avec cependant la possibilité d'abstraire un peu plus à chaque nouvelle notion définie.

A partir de là, il est nécessaire de construire toutes les mathématiques dont on aura besoin pour décrire au système ce que l'on souhaite lui faire vérifier. Il sera nécessaire de construire l'arithmé-

tique, des notions sur les ordres, sur les fonctions, sur les suites... La liste peut être très longue, mais comme en programmation, on utilisera des "bibliothèques" déjà écrites. Dans un nouveau système, tous les théorèmes usuels que l'on ne précise jamais dans les mathématiques "à la main" devront être écrits préalablement.

Lorsque tout ceci est disponible, il reste encore à traduire les travaux que l'on veut vérifier dans le formalisme et le langage du système. Le formalisme choisi pour l'AP donnera en général lieu à des preuves plus grosses, contenant un grand nombre de formules simples à prouver[4]. Les preuves manipulées dans les AP sont de beaucoup plus "bas niveau" que les preuves mathématiques usuelles, d'une part parce que le langage formel est plus verbeux que le langage usuel, d'autre part parce que toutes les preuves doivent être explicitées complètement.

Un programme informatique étant un objet formel, il est naturel de vouloir utiliser des assistants de preuve pour manipuler des programmes, comme nous allons le voir dans ce qui suit.

1.2 Pour l'informatique

Au cours de la conception d'une application, il est possible d'utiliser un AP dans la phase de spécification, d'implémentation ou de validation de l'application (vérification de la correction de l'application par rapport à sa spécification).

1.2.1 Spécification

La première démarche lors de la spécification d'une application consiste à compiler les *desiderata* du "client" dans un cahier des charges, c'est-à-dire une spécification informelle, en langue naturelle qui doit pouvoir être validée par le destinataire de l'application. Ce document devrait ensuite être réécrit en termes mathé-

[4]De Bruijn remarque empiriquement que lors de la transcription d'un texte mathématique (article, manuel, etc.) la taille du texte formel devant être fourni au système de preuve est proportionnelle à la taille du texte initial, si le système connaît déjà les bases nécessaires. En revanche le facteur de taille varie énormément en fonction du type de texte étudié (voir pour cela les articles de De Bruijn et Wiedijk [14, 66]).

matiques, mais toujours dans un langage informel, de manière à éclater la spécification en un certain nombre de points (voir les travaux de Nestor Lopez sur la spécification de programme et les méthodes formelles [42]).

A partir de cette deuxième étape de spécification, si l'on souhaite utiliser des méthodes formelles de conception, il est possible de réécrire chacun des points du cahier des charges, mais cette fois en langage mathématique formel (théorie des ensembles, des types, logique d'ordre supérieure) [2, 60].

Ce passage du langage mathématique informel au langage mathématique formel est nécessaire pour trois raisons : d'abord la difficulté de traiter des textes en langage naturel ; ensuite, le fait que contrairement à ce que l'on pourrait penser, la langue des mathématiciens est riche en non-dits, en preuves laissées implicites, en abus de langage et de notations ne posant pas ou peu de problèmes pour un lecteur "humain", mais impossible à traiter par un programme (par exemple, les points de suspensions). De plus, comme indiqué plus haut, la formalisation des mathématiques se fait dans une théorie particulière, donc tous les objets manipulés doivent être explicitement codés en terme d'objets primitifs de cette théorie : il n'y a pas de "notions bien connues de tous" qui ne nécessitent pas d'explications préalables.

Un autre problème important se pose. La spécification peut-être fausse ou incomplète. Si l'on souhaite spécifier une fonction f de tri sur les liste d'entiers, on doit préciser ou vérifier les six points suivants :

1. Définir les listes.
2. Définir la propriété sur les listes "être triée".
3. Définir la propriété "être une permutation d'une liste" (le contenu de la liste est préservé par le tri).
4. Montrer que pour toute liste l, $f(l)$ est une liste.
5. Montrer que pour toute liste l, $f(l)$ est triée.
6. Montrer que pour toute liste l, $f(l)$ est une permutation de l.

Si on omet de préciser le troisième point, la spécification accepte la fonction f qui à toute liste associe la liste vide (qui est une liste triée). Dans certains cas, on pourra se rendre compte de cette erreur lorsque l'on voudra utiliser des propriétés de la liste triée pour montrer une autre propriété d'une fonction utilisant le tri. Mais ce

n'est pas toujours le cas. Il est donc nécessaire de s'assurer (il n'y a pas de garantie formelle) que la spécification est complète. Par rapport à quoi peut-on vérifier la spécification, si ce n'est la volonté du destinataire du programme ?

En dépit de ces difficultés, l'usage des systèmes d'aide à la preuve ne peut qu'améliorer la fiabilité des programmes. De plus, le passage d'une spécification informelle à une spécification formelle pourra éventuellement amener à se rendre compte d'un certain nombre de d'oublis, d'ambiguïtés, voire d'erreurs dans la spécification informelle.

1.2.2 Implémentation

Après la phase de spécification vient la partie programmation de l'application. Il est possible de spécifier formellement une application, sans pour autant utiliser de méthodes formelles pour la programmation et la vérification. En revanche, si l'on souhaite "programmer en prouvant", ou vérifier le code produit, il est nécessaire de l'avoir spécifié formellement.

Dans l'approche "programmer en prouvant" se distinguent principalement deux méthodes : l'extraction et raffinement. Elles reposent toutes deux sur une formalisation mathématique des programmes, donc du langage de programmation utilisée : il est nécessaire d'avoir un langage mathématique permettant de manipuler les programmes écrits dans le langage de programmation utilisé. Il faut donner une *sémantique* (un sens mathématique) aux différentes instructions, fonctions et opérateurs de contrôle utilisés dans ce langage, ainsi qu'au données manipulées.

1.2.2.1 Extraction

L'*extraction* est adaptée à la conception de programmes fonctionnels. Elle fait appel à l'isomorphisme de Curry-Howard. On a vu au début de ce chapitre que cet isomorphisme établissait la nature "exécutable" des preuves. L'extraction consiste à transformer la preuve qu'une fonction existe (et est bien typée) en un programme calculant cette fonction. Pour que cela fonctionne, il est nécessaire que les preuves soient faites dans une logique "constructi-

ve", c'est-à-dire une logique dans laquelle toute preuve d'existence d'un objet donne une méthode pour construire celui-ci. On peut citer comme exemple de formalisation permettant l'extraction le calcul des constructions de Thierry Coquand [18] utilisé dans Coq, ou encore AF2 de Jean-Louis Krivine [34].

Le mécanisme d'extraction de programme dans sa version la plus simple consiste à exécuter la preuve en lui fournissant l'entrée souhaitée. Cependant, il y a dans la preuve plus d'informations que dans le programme que l'on souhaite obtenir (voir Berardi [6] ou Curmin [19]). Il est donc utile de perfectionner le système de manière à ne conserver que la partie de la preuve ayant un *contenu algorithmique*, c'est-à-dire effectuant un calcul nécessaire pour obtenir la valeur finale, par opposition aux calculs qui serviront à montrer que cette valeur vérifie la propriété. Plusieurs méthodes existent pour cela. Dans le système AF2, certaines opérations effectuées dans la preuve n'apparaissent pas dans le terme (le code) extrait, notamment les manipulations sur les quantificateurs, et l'utilisation de l'égalité. Dans le calcul des constructions, on utilise deux types différents pour les formules avec ou sans contenu algorithmique (plus de détails sur Coq sont donnés section 1.3.1).

L'extraction pose également le problème de la maîtrise de ce contenu algorithmique : on peut se demander quel est l'algorithme de tri obtenu si l'on prouve l'existence d'une fonction de tri sur les listes par ce procédé. Il dépend en fait de la preuve elle-même. Ce qui oblige l'utilisateur du système à écrire une preuve correspondant à un tri efficace, au lieu de prouver la proposition le plus simplement possible. On montre qu'il est possible de trier une liste en donnant une méthode pour le faire (dans une logique constructive, il n'y a pas d'autre solution pour prouver l'existence de quelque chose), et cette méthode correspond à l'algorithme choisi. Dans AF2, les définitions équationnelles peuvent être vues comme une spécification de l'algorithme, plus précise que la spécification globale (les propriétés attendues du programme). Sur les preuves et l'extraction de différents tris dans le calcul des constructions, on peut se référer à P. Manoury [43].

1.2.2.2 Raffinement

Le *raffinement* est issue des travaux d'axiomatisation des langages de programmation de Morgan [45] et Dijkstra [25]. Contrairement à la méthode précédente, celle-ci est beaucoup plus adaptée au traitement des programmes impératifs.

Comme pour l'extraction, on part de la spécification formelle, mais cette fois, on cherche à "raffiner" cette spécification en passant par des états intermédiaires (des spécifications intégrants des parties de programme ou *vice-versa*), jusqu'à atteindre un programme vérifiant cette spécification.

On ne passe donc pas directement de la spécification au programme définitif, mais plutôt de pseudo-programme en pseudo-programme, les premiers ressemblants beaucoup à la spécification, chaque étape introduisant de plus en plus de déterminisme, jusqu'à arriver à des programmes annotés par quelques propriétés, puis au programme final. Chacune de ces étapes engendrera des *obligations de preuves*, permettant de garantir que l'étape $n+1$ satisfait bien ce qui est requis à l'étape n. Les représentations des données et les algorithmes seront de plus en plus précis.

Alors que l'extraction de programme consiste en partie à faire le tri entre ce qui dans une preuve relève du contenu algorithmique et le reste que l'on "oublie", dans le raffinement, le contenu algorithmique est introduit petit-à-petit par l'utilisateur du système, tandis qu'il fait disparaître la spécification.

Le principal système faisant appel au raffinement est la Méthode B de J.-R. Abrial, qui combine cette technique avec une méthodologie de génie logiciel dans une formalisation en théorie des ensembles [2].

1.2.3 Vérification

La méthode de la *vérification de programme* consiste à prendre un programme déjà écrit, et à en vérifier des propriétés, notamment sa spécification formelle. Contrairement aux deux méthodes précédentes, celle-ci peut donc s'exercer *a posteriori* sur un programme existant. Cette méthode n'est pas forcément plus simple que les autres, mais elle a l'avantage de dissocier la partie programmation

de la partie preuve.

La question du contenu algorithmique est ici très simple : le programme nous est donné, il reste à vérifier qu'il effectue ce qu'on en attend. Pour vérifier un programme, on procède en montrant que chacune des parties du programme accomplit son devoir, et qu'en les combinant entre elle, le tout vérifiera les propriétés attendues.

Une des plus anciennes approches de la vérification de programme est la logique de Hoare-Floyd consistant à annoter un programme impératif de clauses exprimant les pré-conditions (ce qui est vérifié avant d'entrer dans une portion de code), les post-conditions (ce qui est vérifié après celle-ci) ainsi que les invariants des boucles [30]. Les règles de cette logique décrivent l'action de chacune des constructions du langage. A partir de ce jeu de règles, on peut montrer que si la clause d'entrée est vérifiée, alors la clause de sortie le sera forcément.

Une autre approche (celle de PVS, ACL2, mais aussi celle que nous avons choisi d'adopter), consiste à utiliser une logique munie d'un langage de spécification et un langage de terme qui est le langage de programmation sur lequel nous voulons travailler. Les formules du langage logique expriment alors les propriétés des programmes. Cette approche est adaptée à la preuve de programme écrits dans un langage fonctionnel. En effet, dans le cadre de la programmation fonctionnelle, on ne manipule pas un *état*. Les propriétés des fonctions ne dépendent donc pas d'une notion de temps ou de déroulement du programme.

Nous revenons sur les raisons qui nous ont fait choisir la vérification de programme pour notre système dans la section 2.3.

1.3 Tour d'horizon de quelques systèmes

Depuis le système Automath de De Bruijn, un grand nombre de systèmes d'aide à la preuve ont été développés par des équipes différentes et avec des motivations différentes. Certains sont plus orientés pour l'informatique ou pour les mathématiques, d'autres encore se veulent destinés à ces deux domaines.

Nous évoquons dans ce qui suit quelques systèmes, choisis de manière à avoir une vue d'ensemble. Les systèmes auxquels nous

nous intéressons sont ceux qui se prêtent à la preuve de programmes fonctionnels (ce qui exclue le raffinement et la logique de Hoare). Nous avons sélectionné quelques systèmes typiques dans leur approche du problème. Nous passerons en revue Coq, Agda et Alfa, Isabelle/HOL, PhoX, PVS et ACL2. Évidemment cette liste est très loin d'être exhaustive, compte tenu du nombre de systèmes existants.

Dans tous les exemples qui suivent, une caractéristique est commune : les preuves sont vues comme des arbres dont la racine est la proposition que l'on souhaite prouver et dont les nœuds sont étiquetés par la règle (ou tactique) utilisée pour décomposer le but en sous-buts qu'il reste à prouver.

Les critères nous intéressant dans ce tour d'horizon sont :
- la logique (langage de terme et de spécification) et plus particulièrement sa simplicité ;
- la facilité de créer et manipuler des types récursifs (très utiles dans le cadre de la programmation fonctionnelle) ;
- la présence de termes de preuve vérifiés par le système ou exportables vers un autre système (pour la sûreté du système) ;
- le degré d'automatisation.

1.3.1 Coq

Le système Coq est développé depuis 1986 à l'INRIA. Il est dû à l'origine à Thierry Coquand et Gérard Huet [62], et a profité de nombreuses contributions. Ce système est basé sur le calcul des constructions inductives [65], une extension du lambda-calcul typé, comprenant les types récursifs, le polymorphisme de type, ainsi que le produit et la somme "dépendants", c'est-à-dire dont le deuxième argument peut dépendre du premier. Un exemple de type dépendant (d'un entier) est le type "être une liste de longueur n". La richesse du système de type de Coq est nécessaire : dans le calcul des constructions on assimile types et formules (*types as formulae* [20, 31]) et la preuve d'un énoncé est un terme dont le type est l'énoncé démontré.

Dans Coq, tout est codé dans ce calcul : aussi bien les termes, que les formules, les preuves ou les types. . . La logique utilisée est d'ordre supérieur, et elle est intuitionniste (n'utilise pas le tiers-

exclu). Coq est un système qui permet l'extraction de programme à partir d'une preuve. Dans ce système, la preuve d'une proposition de la forme "pour tout x de type τ, il existe y de type τ' tel que la proposition $\Phi(x, y)$ est vraie" est une fonction prenant en argument un x de type τ et retournant un témoin y de type τ' et une preuve Π que y vérifie bien cette propriété. Dans le cadre de l'extraction de programme (cf. section 1.2.2.1), on n'est intéressé que par la valeur de retour y, et non par la preuve : dans Coq, on distinguera donc les propositions ayant un contenu algorithmique (dont la preuve permet de construire y), celles-ci seront de type Set, des propositions sans contenu algorithmique (dont la preuve permet de construire Π), qui porteront le type $Prop$. L'extraction oublie la preuve de tout ce qui est de type $Prop$ pour ne conserver qu'une fonction (la preuve de ce qui est de type Set) qui à x va associer y.

Le codage dans le calcul des constructions inductive a pour avantage d'être uniforme, mais pour inconvénient de n'être pas naturel pour l'utilisateur. En effet, le calcul des constructions est un système riche, mais complexe, et l'utilisateur ne peut jamais oublier qu'il travaille dans ce système.

En ce qui concerne les définitions, déclarations, etc. on utilise un langage étendu permettant de construire les objets de façon plus simple et avec un langage plus riche que le calcul des constructions (bien que tout soit traduit vers celui-ci). Pour les preuves, l'objectif est de construire le terme de preuve, mais l'utilisateur est aidé par des tactiques plus ou moins automatisées de "granularité" supérieure au calcul des constructions. Comme la preuve est traduite en un terme du calcul, elle peut être validée (une fois terminée) par une partie de faible taille du système, et cette validation est une simple vérification de type (le type d'une preuve étant l'énoncé qu'elle prouve). La sûreté du système repose donc sur cette vérification du type du terme de preuve.

L'interface de Coq est une boucle d'interaction de type "ligne de commande", cependant d'autres interfaces ont été conçues pour communiquer avec le système. On notera particulièrement Proof General [3] qui est une interface générique réalisée comme extension de l'éditeur de texte XEmacs [67], qui peut être adaptée à d'autres systèmes de preuve en ligne de commande et qui se rapproche d'une édition pleine page. D'autres interfaces ont aussi été implémentées pour Coq, notamment PCoq ou son ancêtre Ct-

Coq, de Yves Bertot et Laurent Théry [63], qui ont une approche "prouver en cliquant", c'est-à-dire mettant en avant l'ergonomie et l'utilisation de la souris pour l'utilisation de systèmes d'aide à la preuve.

1.3.1.1 Exemple

Nous avons choisi quelques exemples en Coq pour illustrer ce qui a été dit, et pour montrer à quoi ressemble la manipulation d'un système d'aide à la preuve. Ces exemples sont des extraits d'un exercice de Jean Goubault consistant à prouver la correction d'une fonction de tri par insertion.

Les entiers sont définis comme un type récursif (`O` est un entier et `S` prend un entier et construit son successeur), ainsi que les booléens, comme des types inductifs. Les deux constructions suivantes définissent les constantes `O`, `S`, `true` et `false`, ainsi que les noms de type `nat` et `bool`. Elles expriment que les entiers sont définis comme le plus petit ensemble contenant `O` et clos par `S` et que `bool` est le plus petit ensemble contenant `true` et `false`. La commande `Inductive` a également pour effet de créer automatiquement la règle d'induction, le filtrage par cas et la définition récursive du type de donnée.

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

```
Inductive bool : Set := true : bool | false : bool.
```

Il ne faut pas confondre le type `bool` et ses valeurs `true` et `false` avec les valeurs de vérités qui sont situées au niveau logique (le type des propositions est `Prop`).

On peut définir des fonctions en utilisant un point fixe et une construction "par cas". On définit ici la fonction `insere` prenant deux arguments `x` (un entier) et `l` (une liste) et retournant une liste. La fonction est récursive et définie par cas sur `l`. Dans le cas où `l` est la liste vide (`Nil`), on renvoie une liste constituée du seul élément `x`.

Lorsque `l` est de la forme `(Cons m l')` (au moins un élément), il y a deux possibilités, si `x` est inférieur à `m`, on renvoie `(Cons x l)`,

sinon, on se trouve dans le cas récursif, on retourne
`(Cons m (insere x l'))`.

```
Fixpoint insere [x:nat; l:liste] : liste :=
  Cases l of
      Nil => (Cons x Nil)
    | (Cons m l') =>
      if (inferieur x m)
      then (Cons x l)
      else (Cons m (insere x l'))
  end.
```

Cette fonction est totale et Coq admet sa définition car la construction de la fonction met en évidence que son second argument décroit à chaque appel récursif. Les listes étant bien fondées en Coq, la récursion finit donc par s'arrêter.

Il faut définir le prédicat "être ordonné" et pour cela on commence par donner le prédicat exprimant qu'un entier minore une liste. Ce prédicat prend deux argument (`x` et `l`). Si `l` est la liste vide (`nil`), alors le prédicat est vrai, sinon, la liste est de la forme `(cons m l')` (un élément `m` suivi du reste de la liste `l'`) et dans ce cas, `x` doit être inférieur ou égal à `m`, et `x` doit minorer le reste de la liste.

```
Fixpoint minore [x:nat; l:liste] : Prop :=
  Cases l of
      nil => True
    | (cons m l') => (inf_eq x m)=true /\ (minore x l')
  end.

Fixpoint ordonnee [l : liste] : Prop :=
  Cases l of
      nil => True
    | (cons n l') => (minore n l') /\ (ordonnee l')
  end.
```

Après avoir prouvé quelques lemmes "techniques", on peut exprimer le premier lemme fondamental : l'insertion préserve l'ordre. Ce lemme s'écrirait $\forall l : liste.ordonne(l) \rightarrow \forall x : nat.ordonne(insere(x, l))$

```
Lemma insere_preserve_l_ordre :
   (l:liste) (ordonnee l) ->
```

```
      (x:nat) (ordonnee (insere x l)).
Proof.
insere_preserve_l_ordre < Induction l.
```

La preuve se fait par induction sur `l`. Le système nous présente les deux cas à prouver, et demande de prouver le cas de base (la formule à montrer est sous la double barre) en nous répondant :

```
2 subgoals

  l : liste
  ============================
   (ordonnee nil)->(x:nat)(ordonnee (insere x nil))

subgoal 2 is:
 (n:nat; l0:liste)
  ((ordonnee l0)->(x:nat)(ordonnee (insere x l0)))
  ->(ordonnee (cons n l0))
  ->(x:nat)(ordonnee (insere x (cons n l0)))

insere_preserve_l_ordre <  Simpl.
2 subgoals

  l : liste
  ============================
   True->nat->True/\True
```

Les évaluations de `ordonnee` et `insere` transforment le but à prouver en :

```
  l : liste
  ============================
   True->nat->True/\True
```

Ce qui se prouve aisément.

Le deuxième cas de la récurrence est alors affiché :

```
  l : liste
  ============================
   (n:nat; l0:liste)
    ((ordonnee l0)->(x:nat)(ordonnee (insere x l0)))
    ->(ordonnee (cons n l0))
    ->(x:nat)(ordonnee (insere x (cons n l0)))
```

```
insere_preserve_l_ordre < Intros.
```

La commande `Intros` (introduction du quantificateur universel et de l'implication) permet d'obtenir le séquent suivant, où l'on a la liste des hypothèses en haut, et la formule à prouver en bas.

```
  l : liste
  n : nat
  l0 : liste
  H : (ordonnee l0)->(x:nat)(ordonnee (insere x l0))
  H0 : (ordonnee (cons n l0))
  x : nat
  ============================
   (ordonnee (insere x (cons n l0)))

insere_preserve_l_ordre < Simpl.
```

Nous arrêterons là la description de cet exemple...

1.3.1.2 Interface

En ce qui concerne l'interface, on utilisera par exemple Proof General (cf. figure 1.1), qui offre des possibilités d'édition pleine page, en utilisant les fonctions d'historique (*undo*) de Coq pour les retours en arrière, et en rejouant le *script* (les commandes entrées par l'utilisateur) pour repartir en avant. Dans la fenêtre du haut, l'utilisateur entre le script constitué des commandes vernaculaires et des commandes de preuve. Dans la partie basse, le système affiche les buts à prouver lorsque l'on est en mode preuve.

La zone grisée de la fenêtre de script indique ce qui a déjà été validé par Coq. Nous nous apprêtons à appliquer le lemme `le_O_n` sur le but courant qui est affiché dans la fenêtre du bas `(le (O) (S n0))` avec les hypothèses disponibles.

Coq extrait des objets preuves exprimés dans le calcul des constructions. Si on l'utilise pour la programmation, les programmes seront exprimés dans le calcul des construction, puis, à travers l'extraction, seront convertis en ML. Coq ne peut exprimer que des fonctions totales.

Au niveau de l'automatisation, on distinguera des tactiques puissantes mais spécialisées (fragment décidable de l'arithmétique, al-

gèbre) et des tactiques de plus bas niveau (proches du formalisme).

1.3.1.3 `Program`

La tactique `Program` de Coq, développée à partir des travaux de Catherine Parent [47, 48], est dédiée à la preuve de programmes fonctionnels totaux.

L'approche utilisée est de considérer l'opération inverse de l'extraction de programme : pour prouver qu'il existe une fonction vérifiant une spécification, on donne le programme correspondant au contenu algorithmique de la preuve, celle-ci ayant en général la même structure que le programme. Il manque alors des informations pour pouvoir fabriquer celle-ci : par exemple, on connaît les récurrences à appliquer, mais on ne sait pas quelle hypothèse d'induction utiliser. Il est donc nécessaire de guider la construction de la preuve par des annotations dans le programme (invariants, etc.) correspondant au contenu "logique" de la preuve, par opposition à son contenu algorithmique.

Lors de l'utilisation de cette tactique, on donne un programme annoté, réalisant le théorème à prouver et la tactique essaie de construire la preuve à partir du programme et des indications, tout en engendrant des *obligations de preuves* que l'utilisateur doit prouver dans Coq, de façon habituelle.

Cette technique de preuve, bien qu'utilisée sur des programmes fonctionnels, se rapproche de ce que l'on pourrait écrire en utilisant une logique à la Hoare.

1.3.2 Agda et Alfa

Agda est un système de preuve basé sur la théorie des types de Martin-Löf, développé par Catarina Coquand dans le langage Haskell. C'est un vérificateur de preuve, prenant une preuve (un lambda-terme typé) et une proposition (son type), et vérifiant la correction de la preuve et son adéquation avec la proposition. Il y a donc la même approche *types as formulae* que dans Coq. Agda peut être vu soit comme un système de vérification de preuve, soit comme un langage de programmation. Le lambda-calcul est étendu avec les définitions récursives, des types algébriques, une

FIG. 1.1 – Coq et Proof General

construction "par cas", ainsi que des types enregistrements. Les preuves manipulées sont de bas niveau et il n'y a pas d'automatisation [17].

On peut l'utiliser conjointement avec Alfa (également écrit en Haskell), un éditeur interactif et structuré de preuve où l'utilisateur "remplis les trous" dans la preuve ou le programme, tout en étant guidé par le type (on ne peut insérer qu'une expression du type attendu). La preuve se fait en deux temps : d'abord elle est construite par l'utilisateur, et ensuite le système se charge de la vérifier, notamment au niveau de la terminaison (on n'utilise pas le théorème dans sa propre preuve). L'approche d'Alfa est différente de celle des autres systèmes cités ici : on manipule explicitement la structure du terme de preuve [29].

En Agda, la terminaison des programmes n'est pas garantie par construction, mais par un critère externe (cf. [1]) : il est possible d'écrire une fonction qui ne termine pas, mais cette heuristique est capable de le détecter. On n'exprime pas les programmes avec un récurseur (un opérateur prenant en charge la récursion), mais avec une définition récursive et une analyse par cas (proche des langages comme ML).

Comme dans la plupart des systèmes *types as formulae*, on a le choix entre deux approches de la preuve de programme. La première approche consiste à écrire une spécification puis la fonction l'implémentant et enfin montrer que celle-ci est effectivement valide par rapport à la spécification. La deuxième solution, qui semble plus naturelle dans Agda est d'écrire un programme prenant en entrée les données et retournant le résultat et la preuve que ce résultat est conforme à la spécification. Dans le cas d'une fonction de tri sur les listes d'entiers, par exemple, la fonction de tri prendra une liste et retournera la liste triée et la preuve que celle-ci l'est, ou encore une valeur du type "liste triée" garantissant l'ordre par construction des valeurs du type.

1.3.3 Isabelle/HOL

Isabelle est un AP générique (un *logical framework*) développé par Lawrence Paulson depuis 1986, avec la contribution de nombreuses personnes, dont Tobias Nipkow [52]. Ce système est un des

FIG. 1.2 – Alfa

descendants de LCF de Robin Milner (de même que HOL, ou Nuprl), et comme celui-ci, repose fortement sur le langage d'implémentation (Standard ML), qui peut être utilisé comme "macro-langage" pour combiner des tactiques et automatiser des procédures.

Une des particularités d'Isabelle est qu'il implémente en fait une "méta-logique", c'est-à-dire une logique permettant de définir d'autres logiques. Parmi les logiques développées pour ce système, on trouve une logique d'ordre supérieure (HOL), une logique du premier ordre, la théorie des ensembles de Zermelo-Frænkel, et bien d'autres. Cette méta-logique est également une logique d'ordre supérieure basée sur l'implication, la quantification universelle et l'égalité. Cela est suffisant pour définir des règles pour les logiques cibles que l'on souhaite utiliser.

Par exemple, on définira la conjonction ($\wedge$) par :
- $\forall P, Q. Trueprop(P \wedge Q) \Rightarrow (Trueprop(P))$ signifiant que si $P \wedge Q$ est vrai, alors P est vrai ;
- $\forall P, Q. Trueprop(P \wedge Q) \Rightarrow (Trueprop(Q))$ la règle duale ;
- $\forall P, Q. Trueprop(P) \Rightarrow (Trueprop(Q) \Rightarrow (Trueprop(P \wedge Q)))$, qui exprime que si P est vrai et Q est vrai, alors $P \wedge Q$.

On note l'utilisation de $Trueprop$ qui permet d'interpréter une valeur de vérité de la logique que l'on définit vers les valeurs de vérité de la méta-logique.

Isabelle/HOL, c'est-à-dire Isabelle avec une logique d'ordre supérieure, permet de manipuler des termes dans un langage s'apparentant à un lambda-calcul typé, avec types algébriques et analyse par cas. Le langage de termes et de formules est le même, ainsi, les valeurs booléennes du langage de programmation sont aussi les valeurs de vérité du langage, et l'égalité dans les termes est la même chose que le "si et seulement si".

Isabelle ne permet pas de construire et d'exporter des objets preuves, mais utilise les types de données abstraits pour ne permettre de construire les preuves qu'avec un certain nombre de fonctions prédéfinies. Cela permet de ne faire reposer la confiance que sur ce jeu de fonctions, car toute fonction construisant une preuve ou partie de preuve est obligée de passer par ces fonctions *primitives*.

1.3.4 PhoX

PhoX, développé par Christophe Raffalli [55] est basé sur une logique d'ordre supérieure, inspirée par l'Arithmétique Fonctionnelle d'ordre 2 (AF2) de Jean-Louis Krivine (lambda-calcul, égalité et définitions de types inductif voir [34]). PhoX étend donc AF2 à l'ordre supérieur, et ajoute des constructions pour simplifier les définitions inductives.

Les preuves se font sur le même principe que pour la plupart des autres systèmes : on travaille sur des séquents avec des règles basées sur la déduction naturelle et le calcul des séquents.

Une particularité de PhoX est que, de manière similaire à la déduction naturelle, les règles peuvent être classées en règles d'introduction (celles qui, si l'on regarde la preuve des hypothèses vers la conclusion, permettent d'introduire un connecteur ou une constante), règles d'élimination (l'inverse, qui comprend les règles combinant une hypothèse avec le but courant), et règles de réécritures. PhoX généralise cela en permettant de créer de nouvelles règles *intro* et *elim* ou *rewrite* à partir des théorèmes montrés par l'utilisateur. Cela offre l'avantage pour celui-ci de pouvoir taper la commande `intro`, par exemple, et le système déterminera laquelle utiliser parmi sa base de commandes d'introduction répertoriées. Cela permet d'avoir une tactique automatique capable d'utiliser la récurrence (c'est une règle d'élimination), ainsi que les lemmes de l'utilisateur en fonction de la catégorie dans laquelle il les a rangé.

PhoX construit des termes de preuve, mais ne les exporte pas.

Le système PhoX a de plus l'avantage que sa logique est plus "simple" à manipuler que, par exemple le calcul des constructions. Il peut également être utilisé avec l'interface Proof General (voir 1.3.1).

1.3.5 PVS

PVS (Prototype Verification System) est un AP très utilisé (y compris dans l'industrie) développé à l'origine par N. Shankar et S. Owre [46]. Ce système est basé sur une logique d'ordre supérieur simplement typée avec du sous-typage (c'est-à-dire que l'on peut définir le type des entiers pairs, par exemple, comme un sous-type

des entiers naturels) ressemblant au schéma de compréhension de la théorie des ensembles. Le typage n'est donc pas décidable. En fait PVS engendre des obligations de preuves laissées à la charge de l'utilisateur pour chacune des fonctions récursives et lorsque le sous-typage est utilisé. Dans la plupart des cas, ces obligations de preuves peuvent être résolues automatiquement.

Les données manipulées par PVS sont organisées en théories pouvant être paramétrées. Dans le cas de la théorie sur les listes, par exemple, le paramètre sera le type des éléments de la liste.

Dans PVS, le langage de termes est assez riche. C'est un lambda-calcul typé, avec sous-typage, analyse par cas et des extensions comme la conditionnelle, les enregistrements, les fonctions sont redéfinissable par point, etc. Le langage de spécification est une logique d'ordre supérieur avec types dépendants.

PVS n'autorise que des fonctions totales, mais le sous-typage permet de restreindre les types par un prédicat. Il suffit donc d'exprimer exactement l'ensemble sur lequel est défini une fonction. Par exemple, on définira la division euclidienne comme totale sur l'ensemble des entiers non nuls.

Lorsque l'on définit une fonction récursive, il est nécessaire de préciser une mesure sur ses arguments et de vérifier que celle-ci décroît à chaque appel récursif pour en justifier la terminaison.

PVS utilise la notion d'obligation de preuve. Il essaie systématiquement de vérifier le typage de ce qui lui est proposé, et en cas d'échec, signale à l'utilisateur qu'il est nécessaire de prouver la conjecture sur laquelle il a échoué. Les procédures de décision et de simplification automatiques sont assez efficaces.

Aucun objet preuve n'est engendré, tout repose donc sur la correction du code du système (les sources ne sont pas disponibles).

PVS est un système assez complet comprenant une interface dédiée sous XEmacs (voir figure 1.3).

1.3.6 ACL2

ACL2 est un héritier de Nqthm de Boyer et Moore [12]. Il est développé par Matt Kaufmann et J Strother Moore [33].

FIG. 1.3 – PVS

Ce système est écrit en LISP, les termes et formules manipulés sont exprimées en LISP, et il permet de prouver des programmes LISP. Cela implique que les termes manipulés ne sont pas typés, et que tout est construit à partir des listes et des paires.

De façon similaire aux mathématiques usuelles, toutes les variables libres sont quantifiées universellement. En revanche, l'usage explicite de quantificateurs est assez difficile dans ce système[5] (cela passe par l'utilisation de lemmes engendrés par le système).

Le premier trait marquant de ce système est son approche permettant de travailler directement sur des programmes LISP. Comme en PVS, les fonctions sont définies normalement, mais il est nécessaire de donner un argument en justifiant la terminaison, c'est-à-dire indiquer ce qui décroît au cours des appels récursifs. De cette *mesure* (par exemple un ordre lexicographique sur la taille des paramètres de la fonction), le système déterminera automatiquement quel principe d'induction est adapté aux preuves mettant en jeu cette fonction. Ceci fait, on peut commencer à définir et à montrer les propriétés relatives à celle-ci.

Le second trait particulier d'ACL2 est son approche des preuves que l'on pourrait qualifier de "complètement automatisée et particulièrement bavarde". En effet, pour prouver une proposition dans ce système, il suffit de l'énoncer, et si tout se passe bien de patienter en lisant la description de la preuve en cours jusqu'à ce que le système termine par *Qed*. Lorsque le système échoue, ou que l'utilisateur suspecte que la preuve s'enferre dans une mauvaise direction et presse `Ctrl+C`, il est nécessaire de lire le texte de la preuve incomplète produit par le moteur de preuve pour trouver ce qui a pu mal se passer. Soit on y remédie en ajoutant avant la proposition un nouveau lemme (et en le prouvant) qui sera pris en compte au prochain essai, soit on fournit des suggestions au système, du type : "au nœud 1.3.112 de la preuve, il conviendrait de ne pas faire d'induction." Quelque soit la solution choisie, il est nécessaire de relancer la recherche de preuve en espérant que cette fois sera la bonne. Une bonne connaissance de l'heuristique de recherche de preuve est donc indispensable pour l'utilisation efficace d'ACL2. Il faut comprendre ce qui risque de poser des problèmes, et être capable de détecter le plus tôt possible dans le texte qui défile un échec éventuel.

[5] Ceci est dû à l'héritage de Nqthm, où "nq" signifie "not quantified".

1.3.6.1 Exemple

Dans l'exemple qui suit, on montre que la fonction de tri par insertion (définie en Common LISP) satisfait les deux propriétés suivantes : pour toute liste d'entier `l`, `(tri l)` est une liste triée ; pour toute liste `l`, `(tri l)` est une permutation de `l`.

On commence par définir la fonction d'insertion en place dans une liste triée, ainsi que le prédicat être une liste triée (un prédicat est simplement une fonction LISP) :

```
;; Declaration de la fonction d'insertion
(defun insere (x l)
  ; gardes
  (declare (type (satisfies integer-listp) l)
           (type INTEGER x))
  ; corps de la fonction
  (cond ((endp l)       (list x))    ; liste vide
        ((<= x (car l)) (cons x l))  ; x<=1er elt de l
        (t              (cons (car l); autres cas
                              (insere x (cdr l))))))

;; Predicat "est triee"
(defun triep (l)
  ; gardes
  (declare (type (satisfies integer-listp) l))
  ; corps de la fonction
  (cond ((endp l)       t)
        ((endp (cdr l)) t)
        (t (let ((x1 (car l))
                 (x2 (cadr l))
                 (l2 (cdr l)))
             (and (<= x1 x2) (triep l2))))))
```

On notera qu'il est nécessaire d'ajouter des gardes, restreignant l'utilisation des fonctions. Avant de pouvoir exécuter la fonction `(insere x l)`, le système vérifiera que `x` est un entier et que `l` est une liste finie d'entiers. Cela permet de garantir que les gardes des fonctions appelées dans le corps (par exemple `<=`) seront elle-même vérifiées.

Une fois la définition de `insere` entrée dans le système, celui-

ci tente de montrer que moyennant le respect des gardes de cette fonction, toutes les gardes des fonctions appelées sont correctes.

La seconde vérification effectuée par le système lors de la définition d'une fonction est la terminaison de celle-ci. En ce qui concerne `insere`, le système détermine aisément que le critère de terminaison est la décroissance de la taille de `l` à chaque appel récursif.

```
(defthm trie-insere
  (implies (and (triep l) (integerp x))
           (triep (insere x l))))
```

Ce premier lemme (l'insertion dans une liste ordonnée préserve l'ordre) est montré aisément (et automatiquement) par récurrence sur `l`, puis simplification et raisonnement arithmétique. Contrairement aux autres systèmes vus précédemment, on remarque que les variables `x` et `l` sont libres. ACL2 considère que dans les propositions, toutes les variables libres sont quantifiées universellement.

```
(defun trie (l)
    (declare (type (satisfies integer-listp) l))
    (cond ((endp l) l)
          (t        (let ((x  (car l))
                          (l2 (cdr l)))
                           (insere x (trie l2))))))
```

La fonction `trie` consiste simplement à insérer tous les éléments d'une liste dans une liste vide. On montre ensuite que la fonction de tri renvoie une liste d'entier ordonnée.

```
(defthm trie-integer-list
  (implies (integer-listp l)
           (integer-listp (trie l))))

(defthm trie-trie
  (implies (integer-listp l)
           (triep (trie l))))
```

Il reste à vérifier que la fonction de tri retourne une permutation de la liste. On définit une fonction retournant le nombre d'occurrence d'un élément dans une liste.

```
(defun nocc (x l)
```

```
(declare (type (satisfies integer-listp) l)
         (type INTEGER x))
(cond ((endp l)           0)
      ((equal x (car l))  (1+ (nocc x (cdr l))))
      (t                  (nocc x (cdr l)))))
```

Et on montre enfin le théorème exprimant que pour tout entier, son nombre d'occurrences dans une liste `l` est égal au nombre d'occurrence dans `(trie l)`.

```
(defthm trie-perm
  (implies
   (and
    (integer-listp l)
    (integerp x))
   (equal (nocc x (trie l)) (nocc x l))))
```

L'égalité a un statut particulier : dans la plupart des systèmes de preuve, elle est orientée de gauche à droite pour être utilisée comme une règle de réécriture. Il convient donc, dans un théorème se concluant par une égalité, de mettre l'expression la "plus complexe" à gauche de celle-ci, et la plus "simple" à droite.

La preuve se fait automatiquement, sans difficultés. ACL2 est un des systèmes (comme PhoX) pour lesquels l'automatisation est capable d'effectuer les récurrences.

Comme pour PVS, il est nécessaire de faire confiance au programmeurs des tactiques d'ACL2. Il n'y a pas de génération d'un objet preuve simple à vérifier.

Un des inconvénients d'ACL2 est que celui-ci étant basé sur le langage LISP, toutes les structures de données doivent être exprimées sous forme de paire pointée ou de liste.

Les systèmes existant étant très nombreux, il devient pertinent de se demander lequel d'entre eux choisir et sur quels critères effectuer ce choix.

1.4 Synthèse

On ne peut pas répondre à la question "quel système choisir ?", sans se poser auparavant un certain nombre de questions : "pour

quelle tâche ?", "qui doit utiliser le système ?", "quelle est la tolérance aux erreurs ?"...

Nous allons d'abord répondre à la question la plus simple : pour qui ? Un système qui doit pouvoir être utilisé par un ingénieur ne sera pas conçu de la même manière qu'un système destiné à des mathématiciens ayant une forte connaissance de la logique. Ce qui signifie que du point de vue du concepteur du système, il faut se demander à qui est destiné l'AP. Si celui-ci doit être orienté vers les ingénieurs, par exemple, il est souhaitable d'éviter les constructions mathématiques complexes, ou du moins de les masquer dans des constructions connues ou intuitives.

La question de la tolérance aux erreurs est un peu plus délicate. En effet, les AP eux-mêmes ne sont pas épargnés par les erreurs se glissant dans les programmes. Il peut donc être possible, dans un système, de prouver des assertions fausses, ou de ne pas pouvoir prouver des assertions vraies. Il est difficile de vérifier l'intégralité d'un système d'aide à la preuve, et cela aussi bien pour des raisons théoriques (pour vérifier un système S_1, il faut un système S_2 plus puissant au sens logique du terme que S_1, mais alors, qui vérifie S_2...), que pour des raisons pratiques (un AP est un logiciel de taille imposante, et tout vérifier imposerait de certifier les fonctions d'affichage, d'analyse syntaxique, etc.)

En revanche, il est possible de faire construire au système des "objets preuves" simples à manipuler et de les valider avec une partie restreinte du système. Ces objets preuves sont des preuves formelles de très bas niveau, utilisant en général un nombre restreint de constructions. Ce sont des objets dont la correction est vérifiable *syntaxiquement*. Un objet preuve doit être vu comme le *certificat* accompagnant un théorème.

Il sera donc possible de prouver (s'il y a des erreurs dans le système) un théorème faux, ou de prouver un théorème juste avec une preuve fausse, mais le système s'en rendra compte une fois la preuve terminée et refusera la partie fausse et ce qui en découle : on différencie la phase de construction de la preuve de la phase de vérification de la preuve qui ne peut se faire qu'une fois celle-ci terminée. Évidemment, la partie du système prenant en charge la vérification doit être au dessus de tout soupçon, elle doit donc être suffisamment courte et simple pour être "trivialement correcte", ou pouvoir être validée dans un autre AP. De plus il n'est pas inutile de

demander à ce que l'AP ait un moyen d'exporter ces objets preuves simples, de manière à ce qu'eux même puissent être vérifiés dans un AP différent, ou que l'utilisateur du système mettant en doute sa correction puisse écrire aisément un programme validant ces objets preuves. Tous les systèmes n'ont malheureusement pas recours à une telle méthode pour vérifier le travail effectué (c'est notamment le cas de PVS et ACL2). Dans ce cas, la confiance que l'on peut accorder aux travaux effectués dans le système est la confiance que l'on peut accorder au code de celui-ci.

La question la plus importante reste celle de l'utilisation que l'on compte faire du système. Les systèmes adaptés à une utilisation dans le cadre de la preuve de programmes ne sont, en général pas les mêmes qui sont adaptés aux mathématiques. Prenons par exemple le formalisme du système, c'est-à-dire ses fondements mathématiques : si l'on souhaite manipuler des objets représentant des programmes ou des fonctions (au sens programmation), on pourra choisir un formalisme basé sur la théorie des types (pour plus de détails, voir [44]). En effet, dans la théorie des types, les fonctions sont des objets primitifs (dits "de première classe"), ce qui signifie que l'on peut les manipuler simplement, directement. Il n'est pas nécessaire d'utiliser un codage complexe. La fonction est représentée par le "programme" qui la calcule. Dans les mathématiques conventionnelles, on manipule principalement des ensembles, des fonctions définies par leur graphe, des équations, et ce sont ces objets que l'on souhaite manipuler comme objets de première classe (on souhaite au moins avoir des facilités offertes pour les manipuler).

C'est pourquoi il est important de choisir un système dont le formalisme est le plus proche possible des objets que l'on veut manipuler. Il n'est pas souhaitable d'avoir à utiliser un codage complexe pour pouvoir résoudre un problème dans un système de preuve. Le langage proposé pour le codage des termes, et la facilité de créer et manipuler des types de données sont donc très importants dans le choix d'un système, particulièrement dans le cadre de la preuve de programmes.

Pour reprendre les présentations des systèmes de la section précédente, ACL2 est le seul système permettant de vérifier un "vrai" langage de programmation, mais il ne permet pas la vérification d'un terme de preuve. PVS est dans le même cas, bien qu'il ne

travaille que sur un "langage de prototypes". De l'autre côté, Coq, Isabelle/HOL et PhoX donnent plus de garanties mais les formalismes choisis pour ces systèmes s'éloignent plus d'un langage de programmation. Même si Coq permet l'extraction d'un programme ML, l'écriture de celui-ci ne s'apparente plus que de très loin à l'écriture d'un programme. Agda se situe plutôt entre les deux, dans la mesure où l'on manipule des lambda-termes, mais avec des facilités d'écriture se rapprochant de ML.

1.5 Conclusion

On a vu qu'il existe déjà un bon nombre de systèmes d'aide à la preuve, pour lesquels des choix différents ont été faits, et couvrant une partie des besoins. Nous pouvons conclure ce chapitre en affirmant que chaque AP doit être conçu en ayant en tête les tâches auxquelles il est destiné, et, ce qui est tout aussi important, qui sont les utilisateurs qui doivent l'employer.

Dans le chapitre qui suit, nous allons montrer où nos travaux s'inscrivent dans ce paysage et pourquoi nous avons pensé qu'il était nécessaire de créer un nouveau système d'aide à la preuve.

Nous exposerons les choix que nous avons fait au niveau du système et la raison pour laquelle nous avons choisi ML comme langage cible. Nous justifierons également l'originalité et l'intérêt de ces travaux.

Chapitre 2

Motivations de ce travail

Nous présentons dans ce chapitre les choix que nous avons fait lors de la spécification et de l'implémentation de notre système. Nous justifions ici les raisons de ces choix et nous comparons ceux-ci avec les approches qui ont été suivies par les développeurs d'autres systèmes.

Le but de ce travail est la spécification et la réalisation d'un système d'aide à la preuve spécialisé dans la spécification et la vérification de programmes écrits dans un langage de type ML. Pour des raisons de simplicité et de clarté du langage, nous avons choisi une restriction de ce langage que nous appelons miML (pour mini-ML).

Pour spécifier les programmes, c'est-à-dire décrire leurs propriétés, nous utilisons un langage logique basé sur la théorie des types récursive au second ordre. Une utilisation typique sera une suite de déclarations de constantes, de définitions de types et fonctions ML, de définitions mathématiques, d'axiomes et de théorèmes accompagnés de leurs preuves.

Nous désirons également que les prédicats du langage mathématique portent directement sur les programmes miML ; c'est-à-dire que les programmes (mais aussi les données) soient les objets mathématiques manipulés dans notre langage.

Le système doit être interactif et pourvu d'une interface dynamique et structurée. Celle-ci doit permettre de revenir en arrière – par exemple pour ajouter un lemme nécessaire à l'accomplissement d'une preuve – ou de laisser une preuve inachevée pour la terminer ultérieurement, tout en pouvant d'ores et déjà utiliser le

résultat du lemme dans la suite.

Nous aimerions avoir un système se rapprochant d'ACL2, c'est-à-dire qui doit permettre de vérifier *directement* des programmes (ou plutôt des parties de programmes), mais en évitant deux traits d'ACL2 que nous considérons comme des inconvénients : le typage dynamique, qui oblige à vérifier explicitement le types des termes, que cela soit dans les programmes ou dans les théorèmes, mais aussi le manque d'interactivité, bien que cela soit compensé dans ACL2 par une très bonne automatisation.

Dans la suite, nous appelons ce système PAF!, pour "Prouve Avec Facilité !".

2.1 ML comme langage cible

Les langages de la familles ML (*Objective Caml*, *SML*, Moscow ML...), sont des langages fonctionnels, statiquement typés avec des types polymorphes paramétriques. Ils permettent la manipulation de types algébriques, c'est-à-dire de types définis comme une union de valeurs distinctes (les constructeurs) pouvant avoir des arguments dont le type est fixé (mais peut utiliser des paramètres de type). Ces types algébriques peuvent être récursifs.

Nous avons choisi un mini langage ML comme langage cible, pour de nombreuses raisons. Il est pourvu d'une sémantique claire (bien que ne faisant pas l'objet d'une norme), le typage statique permet la détection de nombreuses erreurs dès la compilation et le mécanisme d'inférence de type (de Damas-Milner [21]) évite d'alourdir la syntaxe en précisant le type des variables et fonctions. Ce langage facilite grandement la définition des types récursifs et leur manipulation à l'aide de fonctions définies par filtrage. Le polymorphisme et les fonctionnelles (ou fonctions d'ordre supérieur) permettent un bon niveau d'abstraction.

En plus de ces avantages, il convient d'ajouter que ML est largement répandu dans le monde universitaire, et qu'il est particulièrement adapté à l'enseignement de l'informatique (cf. 2.2).

Un dernier point est que des implémentations très efficaces des langages de la famille ML existent [39].

On ne décrira pas plus ces langages, mais on peut se référer à [41] ou [16].

miML notre mini langage ML est un langage fonctionnel simplifié, statiquement et fortement typé, autorisant, comme ML, le polymorphisme paramétrique et les fonctionnelles. Il autorise la définition de types algébriques (types sommes de ML, potentiellement récursifs), les types n-uplets, les booléens et les entiers naturels. Au niveau des constructions syntaxiques, on donne la conditionnelle `if then else`, le filtrage, et un combinateur de point fixe. La syntaxe donnée pour le langage est assez proche de celle de *Caml Light*. La définition complète du langage est donnée en 3.2, la syntaxe concrète est décrite section 6.4.1.

Le système doit offrir la possibilité de spécifier et vérifier des programmes écrits en pur style fonctionnel : pas d'exceptions, pas de références, pas d'affectations et pas d'entrées-sorties.

2.2 Pourquoi et pour qui ?

Comme on l'a vu dans le chapitre précédent, vérifier du code, même simple, est une tâche beaucoup plus complexe que l'écrire, mais également plus complexe qu'effectuer une vérification informelle, dans la mesure où il est nécessaire de se plier au formalisme du langage mathématique de l'AP. Pour cette raison, un système de ce type n'est pas destiné (en général) à vérifier un programme complet (ce qui peut comprendre un analyseur syntaxique, une interface graphique, etc.), mais plutôt une partie plus fragile, ou plus critique de celui-ci.

Ce système doit pouvoir être utilisé par des personnes n'ayant pas de formation particulière en logique. Il est destiné à des informaticiens souhaitant développer des programmes ou parties de programmes certifiés, dans la mesure où ceux-ci sont exprimables dans miML.

Nous souhaitons également qu'il puisse être utilisé par des étudiants en informatique de deuxième cycle. Une des motivations pour l'écriture de ce logiciel étant d'avoir un outil adapté à l'enseignement des méthodes formelles de spécification et développement.

Ces choix sur les destinations de PAF! ont évidemment influencés à la fois son architecture et la logique sous-jacente au système.

2.3 Vérification de programme

Comme on l'a vu au chapitre précédent, trois approches peuvent être choisies lorsque l'on souhaite faire du développement de programme "certifié" : la vérification de programme, l'extraction ou le raffinement.

Pour le système PAF!, nous avons choisi comme approche la vérification de programme. Celle-ci nous semblait plus intéressante que le raffinement dans la mesure où nous avions choisi de travailler sur un langage de programmation fonctionnel. Nous n'avons pas choisi d'utiliser l'extraction de programme, entre autre parce que le fait de se limiter à une logique intuitionniste peut compliquer certaines preuves très simples sinon. De plus la vérification a comme caractéristique de permettre de travailler *a posteriori* sur un programme.

2.4 Une logique simple

Par rapport à l'objectif que l'on s'est fixé quand aux utilisateurs de PAF!, il est nécessaire de choisir une logique qui soit aisément compréhensible. Certaines logiques sont dites "simples" car elles ne définissent que le minimum de constructions nécessaires : c'est le cas par exemple de certaines logiques d'ordre supérieure, n'introduisant comme connecteur que l'implication et comme quantification que le "pour tout" ; de même, le calcul des constructions peut être perçu comme "simple", car il utilise le même langage pour définir preuves, formules et termes. Ces deux approches simplifient le développement de l'AP, mais compliquent les choses pour l'utilisateur. Ce n'est pas cette notion là de simplicité qui nous intéresse, mais plutôt celle de simplicité pour l'utilisateur.

Nous avons choisi de restreindre l'utilisateur à une logique du second ordre, à laquelle on ajoute des constructions syntaxiques "dédiées" permettant d'avoir l'expressivité nécessaire. Par exemple,

on voudra définir les listes par un type algébrique ML. Cette construction ne pose pas de difficulté, et nous permet de construire tout ce qui est nécessaire à l'utilisation de ce type de données dans un système de preuve (règle de récurrence, typage des constructeurs. . .).

La logique utilisée est typée avec un système de type proche de celui de ML, c'est-à-dire types sommes (dont les types inductifs, comme ci-dessus), types paramétrés par un type (pas de types dépendants de valeurs). De façon similaire à ML, les variables de types sont quantifiées implicitement juste avant le type. Par exemple lorsque l'on définit une constante comme $L : \alpha\ list$ (L est une liste d'éléments de type α), cela signifie en fait $L : (\forall\alpha.\alpha\ list)$, où α ne peut être instanciée que par un type. On ajoute cependant une construction pour quantifier explicitement sur les variables de type. On veut pouvoir écrire

$$\forall\alpha.\forall l : \alpha\ list.\exists l' : \alpha\ list.length(l) = length(l')$$

Dans ce cas, on explicite la quantification sur α, celle-ci ne sera donc pas quantifiée comme prescrit au paragraphe précédent, ce qui garantit que l et l' sont des listes d'éléments de même type.

Les termes du langage sont les termes de miML (ce qui signifie que les fonctions sont considérées comme étant du premier ordre). Cette approche est similaire à celle d'AF2 [34] où l'on remplacerait les définitions équationnelles par la définition de la fonction en ML et la réduction des termes ML.

Passons maintenant au qualités requises relevant de l'architecture du système.

2.5 Un système extensible, mais sûr

À de très rares exceptions près, un assistant de preuve n'est pas développé par une personne seule : ces logiciels sont de taille imposante, ils sont donc en général développés par une équipe, et étendus par ou avec les utilisateurs. C'est un point qui est indispensable à prendre en compte. Il faut ancrer profondément dans le système de quoi faciliter les extensions ultérieures. En l'occurrence, on souhaite qu'il soit aisé d'écrire de nouvelles tactiques et procédures de décisions permettant de faciliter ou d'automatiser les preuves.

Pour cela, il y a deux approches possibles : soit les tactiques supplémentaires sont écrites dans un langage de script plus ou moins riche qui se situe au niveau "utilisateur" du système (voir par exemple pour Coq, David Delahaye [23]), seules les tactiques primitives et celles nécessitant un pouvoir d'expression trop fort pour ce langage de script sont donc écrites dans le même langage de programmation que l'AP. On peut aller d'un simple langage avec des itérations et la conditionnelle à un langage de programmation complet. Soit les tactiques sont écrites directement dans le même langage que le système de preuve (par exemple LCF [28] ou Isabelle [52]). Dans ce cas, il est indispensable d'isoler ces tactiques du reste du programme et d'indiquer explicitement toutes les fonctions et types de données que le programmeur de la tactique a le droit d'utiliser (une *API*). Sans quoi, il sera quasiment impossible à une tierce personne d'étendre le système.

Nous avons choisi ici la deuxième approche, dans la mesure ou elle est plus riche et moins complexe à mettre en œuvre. Elle ne "nécessite pas d'apprendre un énième langage de programmation" pour écrire les tactiques (à condition toutefois de connaître *Objective Caml*).

L'inconvénient est qu'il sera plus difficile pour un utilisateur d'écrire de nouvelles tactiques.

Nous donnons donc un "patron" pour l'écriture de nouvelles tactiques, ainsi qu'une API isolant les opérations légales à ce niveau.

L'extensibilité du système est, comme on l'a vu, indispensable, mais sa sûreté l'est au moins autant. Il ne doit pas être possible de prouver un théorème faux! La fiabilité d'un AP ne doit donc pas reposer sur des tactiques écrites par un utilisateur (et encore moins par le programmeur originel). Nous avons donc choisi d'obliger le développeur de tactiques à écrire deux éléments pour chacune d'entre-elles. Le premier de ces éléments permet au système de générer les sous-buts produits par l'application de la règle en prenant en compte le but courant, les arguments fournis, et l'état courant du système (théorèmes connus, etc.), mais également de vérifier que la règle s'applique bien. La seconde partie d'une tactique est en quelque sorte sa justification : elle doit être capable avec le même but à prouver, d'engendrer les mêmes sous-buts en n'utilisant que des tactiques de plus bas niveau (éventuellement en utilisant des informations collectées lors de l'exécution de la

première étape). Cette approche des tactiques en deux parties est proche ce celle utilisée dans LCF où une tactique renvoie un séquent résultat et une preuve [28].

On peut prendre comme exemple la tactique `Oui` qui répondrait systématiquement que le séquent sur lequel on l'applique est prouvé, la partie création des sous-buts est triviale, mais il est impossible d'écrire une justification pour cette tactique.

Chaque tactique étant pourvue de cette capacité, à l'exception des règles primitives, cela permet donc de construire à partir de toute preuve une autre preuve composée uniquement de ces règles de base, que nous appellerons *preuve atomique*, car on ne peut la décomposer plus (dans le système). Il est alors simple de vérifier la cohérence de cette preuve *a posteriori*. Notre système doit donc vérifier le "critère de De Bruijn" (c'est-à-dire la possibilité de vérifier le travail effectué avec une partie simple et sûre du système)[1]. Nous verrons dans la discussion sur l'ergonomie du système un autre avantage à cette manière de procéder.

2.6 Un système ergonomique

Il n'est pas dans notre propos d'avoir une approche "cliquer pour prouver", ceci pouvant être une extension future du système. Cependant l'interface à un rôle important à jouer dans un système de preuve [63].

L'approche que nous avons choisi est un modèle client-serveur où le moteur de preuve joue le rôle du serveur et l'interface celui du client. Les deux doivent communiquer à l'aide d'un protocole documenté de façon à ce qu'il soit possible de faire fonctionner le système de preuve avec d'autres interfaces, ou que celui-ci puisse répondre à des requêtes envoyées par un programme différent (en quelque sorte, fournir un service de preuve).

L'interface doit permettre une édition *pleine page*, ce qui signifie que l'utilisateur peut se déplacer librement dans le document édité pour ajouter où supprimer des éléments du texte (déclarations, dé-

[1] D'autres méthodes existent pour garantir la correction des preuves. Par exemple dans LCF et ses descendants, elle repose sur le typage du langage d'implémentation (ML) [28].

finitions, axiomes ou théorèmes), mais aussi pour compléter des preuves ou en supprimer des parties. Au niveau des preuves, cela signifie que nous voulons manipuler explicitement la structure de la preuve (comme pour Alfa, cf. 1.3.2) contrairement à l'approche choisie dans la plupart des systèmes où l'on utilise une boucle d'interaction pour saisir les commandes de preuves, avec une fonction *undo* comme moyen de revenir en arrière[2].

Ceci a bien-sûr une influence sur l'architecture du moteur de preuve lui même, qui doit vérifier si ces modifications sont légales (on ne veut pas masquer ni supprimer un identificateur défini plus haut mais utilisé plus bas), ce qui implique de maintenir un graphe de dépendance. De plus, les données doivent être gérées de façon beaucoup plus dynamique que dans un AP traditionnel. Ceci est susceptible d'introduire des erreurs, le système devenant plus complexe, donc plus fragile. C'est pourquoi le processus de vérification doit être global et non local à une preuve : on reprend le fichier depuis le début, et on reconstruit les preuves et les définitions dans un système minimal. Cette vérification est déclenchée par l'utilisateur quand celui-ci le souhaite.

* * *

Dans l'ergonomie d'un système, il est nécessaire de prendre en compte son temps de réponse aux requêtes de l'utilisateur. Il est indispensable que le système sache répondre instantanément (ou perçu comme tel), aux requêtes simples et usuelles. En revanche, il peut prendre un peu plus de temps sur des procédures de décisions complexes ou sur des manœuvres inhabituelles (on peut admettre un temps de réponse de l'ordre de la seconde ou de quelques secondes pour ce qui est vraiment peu fréquent).

C'est un point qui nous conforte dans le choix exposé en 2.5 sur l'écriture des tactiques en deux parties. Cette méthode peut permettre d'écrire une procédure de décision rapide pour un type de problème donné, mais à partir de laquelle on n'est pas capable de construire un terme de preuve. C'est le cas notamment des procédures basées sur les BDD, qui permettent de déterminer efficacement si une formule du calcul propositionnel est une tautologie,

[2]Proof General (voir section 1.3.1) permet pour certain systèmes de compenser ces limitations en utilisant le *undo* et en "rejouant" les scripts pour simuler une édition pleine page.

mais qui ne permettent pas d'engendrer un terme de preuve.

Dans un cas comme celui-là, lorsqu'il sera nécessaire de reconstruire une preuve n'utilisant que des règles atomiques (processus de validation des preuves), il sera nécessaire de prouver que la formule est une tautologie par un autre biais, possiblement beaucoup plus lent. Pendant la construction interactive de la preuve, on a donc un temps de réponse rapide, mais on en paye le prix au moment de la validation. Or, la validation n'est pas un processus interactif, et n'est déclenchée par l'utilisateur que quand il le souhaite. Le temps de calcul n'est donc pas ici un problème.

C'est un gain qui est appréciable par rapport à d'autres systèmes, qui soit utilisent des procédures de décision comme les BDD, soit choisissent de satisfaire le critère de De Bruijn. D'autres méthodes permettent de pouvoir, comme ici, d'avoir "le beurre et l'argent du beurre", mais elles restent beaucoup plus complexes à mettre en œuvre que celle-ci. Il s'agit par exemple, de prouver dans le système même la correction des tactiques utilisées [64].

2.7 Objectifs

Le premier objectif de ces travaux est la réalisation (conception et implémentation) d'un système satisfaisant les critères donnés dans ce chapitre :

- extensible,
- sûr,
- simple pour l'utilisateur,
- ergonomique.

Notre second objectif est d'avoir une spécification des différentes parties du système (moteur, interface, protocole de communication) indépendante du formalisme choisi, de manière à pouvoir être utilisée pour d'autres systèmes.

Le dernier objectif est la description de l'implémentation, pour les parties clefs du système.

* * *

Dans le chapitre suivant, nous allons présenter le formalisme retenu pour PAF!, c'est-à-dire le langage miML, ainsi que les lan-

gages de spécification et de preuve.

Chapitre 3

Formalisation de la logique du système

Ce chapitre a pour but de décrire le formalisme mathématique de notre système de preuve. Celui-ci est plus complexe (il y a plus de constructions, plus de règles) que dans la plupart des systèmes (notamment ceux basés sur la théorie des types) pour deux raisons : tout d'abord, notre système (PAF!) est dédié à la manipulation de programmes dans un sous-ensemble de ML, et doit donc formaliser chacune des construction du langage. De plus nous utilisons des constructions plus complexes pour que celles-ci soient plus naturelles, du point de vue de l'utilisateur.

Nous distinguons trois niveaux : le niveau programmation, où l'utilisateur peut définir de nouvelles fonctions, le niveau spécification, où il définit les comportement de ces fonctions et déduit de nouvelles propriétés à partir de la spécification, et enfin, le niveau preuve où il peut prouver que les programmes écrits vérifient bien les propriétés attendues.

Au niveau programmation, on trouvera un mini langage de programmation, issu du noyau fonctionnel de ML. Ce langage comprend les constructions usuelles de ML, notamment les types algébriques définis par l'utilisateur, le `let rec` et le filtrage, se comportant de la façon habituelle. L'utilisateur définira ses fonctions ML comme il en a l'habitude, sans restrictions (pas de points fixes gardés, récurseurs, filtrages non exhaustifs).

Au niveau spécification, se trouve d'abord un système de ty-

page à la ML usuel (décidable, mais ne garantissant évidemment pas la terminaison des programmes). À cela, on ajoute un langage de formules du second ordre permettant d'exprimer les propriétés des valeurs et fonctions ("cette fonction renvoie toujours un entier pair", "cette liste est triée", etc.). Il est nécessaire d'inclure dans ce langage de quoi exprimer la terminaison des fonctions, qui était laissée de côté par le typage. On pourra par exemple exprimer la propriété de la division euclidienne, qui est de type $nat \to nat \to nat$ comme en ML, mais qui ne terminera que si son second argument est non nul. Dans l'implémentation de notre système, la terminaison ne sera pas redondante avec le typage du point de vue de l'utilisateur, car on utilisera l'inférence de type pour le type ML. Pour compléter notre langage de spécification, on ajoute les commandes permettant de définir des prédicats, de poser axiomes et théorèmes et de déclarer des constantes. Ces constructions (dites *vernaculaires*) sont à rapprocher des définitions de formules et de types par l'utilisateur dans la partie programmation.

Enfin, au niveau preuve se trouvent les règles permettant de montrer que les théorèmes énoncés sont corrects. Le système de règles que l'on propose est basé sur la déduction libre [51] et contient aussi bien des règles usuelles, que d'autres adaptées pour notre usage, d'autres encore sont dédiées à la manipulation de nos constructions.

L'utilisation de la déduction libre pourrait sembler un choix non pertinent dans notre optique de simplification pour l'utilisateur. Mais ces règles ne sont en fait que les briques de bases qui nous permettront de construire d'autre règles (appelées *tactiques*) plus adaptées aux besoin réels (cf. par exemple section 5.6). L'utilisation de la déduction libre est justifiée principalement par des raisons d'implémentation : c'est un système homogène, facilitant l'écriture des tactiques de bas niveau, de plus cela simplifie également l'écriture de l'algorithme de vérification de la preuve.

Ce qui suit décrit plus en détail le découpage par fonctionnalités de notre système.

* * *

Nous commençons par définir le langage miML, qui est le langage sur lequel nous voulons faire de la vérification de programme. L'objectif est de travailler réellement sur les termes du langage de

programmation, aussi, nous distinguerons deux parties dans le langage : le langage des termes, qui est le langage de programmation et le langage des formules (c'est-à-dire le langage logique), qui nous permet d'exprimer des propriétés sur le langage des termes.

Ce langage de formules est un langage typé du second ordre, dont les types seront les types de ML.

Nous définirons dans ce chapitre successivement :

1. le langage de termes de miML. C'est langage similaire au λ-calcul avec constantes, étendu avec un opérateur de point fixe (non gardé), un filtrage non nécessairement exhaustif (`match ... with`), une conditionnelle (`if ... then ... else`), et une fonction d'égalité (structurelle). Ce langage est défini à la section 3.2.

2. Le langage de types de miML, qui est un sous-ensemble fonctionnel (dans les deux sens du termes) du langage de types de ML. Il comprend les types polymorphes paramétrés, les types fonctionnels et la possibilité pour l'utilisateur de définir ses propres types algébriques. Le typage correspond également au typage usuel de ML. Le langage de types est traité à la section 3.2.2.

3. Le système de typage des termes de miML est défini à la section 3.2.3, c'est un mécanisme à inférence de type identique au typage usuel des constructions ML. Le typage sert à limiter les termes légaux (par exemple lors des entrées de l'utilisateur), mais il intervient dans certaines règles de preuve (instanciation d'un quantificateur). Les jugements de typage (dans un environnement) seront notés $(x_1 : \tau_1), \ldots, (x_n : \tau_n) \models_{K,L} t : \tau$.

4. L'évaluation des termes de miML, qui est proche de l'évaluation habituelle en ML est décrite en 3.2.4. Il y a une différence au niveau du traitement de la fonction d'égalité par rapport aux implémentations de ML. Lorsque l'on évalue `t1=t2` dans un langage de programmations, les termes étant clos, il suffit que ceux-ci diffèrent pour que l'on puisse répondre "faux". En revanche, dans un AP, on manipule des termes contenant des variables, et il n'est pas forcément possible de décider de l'égalité de deux termes (par exemple $\forall x \exists y. x = y$ ne doit pas être évalué en $\forall x \exists y. false$). Les jugements d'évaluation (dans un environnement) seront notés $N_{ML} \Vdash t \hookrightarrow t'$

5. Le langage de formules est comme on l'a dit un calcul des prédicats du second ordre, où les prédicats portent sur les termes de miML. L'évaluation de ces termes joue dans notre système le rôle d'une théorie équationnelle. Par rapport aux langages habituels, nous ajoutons pour chaque type ML τ un prédicat $(t\ !\ \tau)$ indiquant que l'évaluation du terme t termine et renvoie une valeur de type τ (si τ est un type fonctionnel, cela signifie que t termine quelque soient les arguments bien typés qu'on lui fournit). Les quantifications sont bornées par les types, avec la restriction que la quantification du premier ordre ne porte que sur les termes dont l'évaluation termine (donc bornés par le prédicat $(\_\ !\ \_)$). On ajoute également la possibilité de quantifier sur les variables de types. Ce langage est défini en 3.3.1.

6. Le système de typage des formules qui est similaire au typage des termes (et décidable) ne définit qu'un seul type ($Prop$) pour les formules. Ce sont les règles de typage qui permettent de restreindre le langage au second ordre en n'autorisant que les types de la forme $\tau \to Prop$, où $Prop$ n'apparaît pas dans τ. Le typage des formules est décrit à la section 3.3.2. Les jugements de typage des formules seront notés $E \models_{K,L} \varphi : Prop$.

7. Le système de preuve décrit les règles primitives permettant de prouver un séquent. Ce système est inspiré de la déduction libre, mais est étendu de manière à gérer nos constructions, et plus adapté à nos besoins (voir 3.3.3). On y trouvera également des règles permettant de remplacer un terme t par t' ou l'inverse, si t s'évalue en t'. Les jugements de preuve sont notés $\Gamma \vdash_{Env} \varphi$.

8. Nous ajoutons dans la théorie ce qui est nécessaire pour pouvoir manipuler les types ajoutés par l'utilisateur (récurrence, typage des constructeurs), ainsi que pour étendre l'utilisation de l'égalité (voir 3.4).

9. Le langage vernaculaire permet d'ajouter des déclarations, définitions, types algébriques (types ML utilisateurs), termes, axiomes et théorèmes. Il est défini en 3.5.

On ne manipulera pas ici que des fonctions totales, ce qui pose le problème de savoir quelle est l'information donnée par le type d'un terme ou d'une formule. Nous voulons un système de type

pour lequel l'inférence est décidable. Comme notre langage est suffisamment expressif, le typage ne peut pas garantir la terminaison des fonctions.

Il faut comprendre l'information donnée par les règles de typage comme "si l'évaluation de ce terme termine sur une valeur, alors le type de celle-ci est τ. Dans les cas triviaux (pas de point-fixe, filtrage exhaustif, etc.), le typage est suffisant pour garantir la terminaison. Dans les cas où ces conditions ne sont pas satisfaites, il est nécessaire de *prouver* la totalité en utilisant les règles du système de preuve.

Ceci nous permet de mettre de côté les cas simples pour les traiter automatiquement, en ne laissant à la charge de l'utilisateur que les cas complexes, par exemple la preuve de terminaison de certaines fonctions récursives.

On pourra constater que dans ce système logique, la quantification sur les fonctions ne porte en réalité que sur les fonctions totales. C'est une restriction du système qui est assez faible, mais elle interdit par exemple de passer une fonction partielle en argument à une fonctionnelle[1].

3.1 Quelques notations

Dans tout ce qui suit, on utilisera les notations habituelles suivantes. On notera la *substitution* $t[u/x]$, c'est-à-dire t dans lequel on substitue u à toutes les occurrences libres de la variable x.

On appellera *environnements* des fonctions partielles d'un ensemble (de variables ou de constantes) vers un ensemble (de termes ou de types). On notera les environnements $\{(x_1 \rightarrow t_1); \ldots; (x_n \rightarrow t_n)\}$. Les environnements de typage se présentent de la même manière, mais on les notera $\{(x_1 : t_1); \ldots; (x_n : t_n)\}$.

On définit la *surcharge* $\oplus$ sur les ensembles tel que $E \oplus t$ est l'ensemble E auquel on ajoute t et sur les fonctions tel que $f \oplus (x \rightarrow t)$ est égal à f' tel que $f'(y) = f(y)$ si $y \neq x$ et $f'(x) = t$.

On définit l'*égalité syntaxique* $\equiv$ entre terme, formule et types.

[1] Aussi appelée fonction d'ordre supérieure, c'est une fonction prenant en argument une autre fonction.

On se donne la notation $\overrightarrow{t}$ pour $t_1, \ldots, t_n$ dont on pourra préciser le sens en cas d'ambiguïté.

3.2 Définition du langage miML

Le langage que nous définissons ici est inspiré d'*Objective Caml* [16] qui est un langage développé à l'INRIA depuis 1987 (alors appelé *Caml*). Ce langage est issu d'une longue tradition de la programmation fonctionnelle, remontant (en passant par LISP), jusqu'au λ-calcul, un formalisme mathématique créé par Church en 1932.

Le lambda-calcul offre une notation pour écrire les fonctions permettant de représenter toutes les fonctions calculables. La famille des langages ML (pour méta-langage) se base sur ce formalisme, en y ajoutant des constantes, des primitives, mais aussi (contrairement à LISP), en le dotant d'un contrôle strict des types à la compilation (typage statique).

Cette famille de langage a pour avantage d'être de "haut niveau", c'est-à-dire que l'on programme de manière très abstraite, très loin du niveau "machine". De plus, le typage statique facilite l'écriture des programmes, en ce sens que beaucoup d'erreurs sont détectées dès la compilation du programme.

L'autre raison qui nous a amené à choisir un langage *à la* ML, est qu'une partie de ce langage est suffisamment formalisable pour être intégrée dans un système de preuve. Pour ACL2, le choix a été de prendre LISP pour ces même raisons. Nous avons plutôt choisi ML car le langage de type est plus riche (types algébriques), et le typage fort permet de rendre décidable une partie de ce qui relève de la preuve dans ACL2.

Dans ce qui suit, nous définissons le langage miML et ses propriétés. Nous commencerons par définir le langage de termes, puis les types, en 3.2.3, nous décrirons le typage des termes. Enfin nous donnerons les règles d'évaluation du langage en 3.2.4.

3.2.1 Syntaxe

Le langage miML est un fragment fonctionnel de ML bien connu : il contient des variables, des constructeurs de types, l'application, l'abstraction, une fonction d'égalité structurelle, une conditionnelle, un combinateur de point fixe, une définition locale (`let x = e1 in e2`) et le filtrage.

Le point fixe n'est pas gardé et le filtrage n'est pas une simple analyse par cas, mais un filtrage en profondeur, qui peut ne pas être exhaustif. Cela implique que l'évaluation d'un programme peut se bloquer ou ne pas terminer, mais cela permet d'exprimer les fonctions comme dans les langages de programmation, plutôt que d'utiliser des récurseurs (par exemple dans le système T), ou des points fixes devant satisfaire des critères syntaxiques (par exemple dans Coq). En contrepartie, des preuves de terminaison sont nécessaires pour justifier la totalité des fonctions.

Parmi les symboles que nous définissons, on trouve la *fonction d'égalité*. Celle-ci est donnée comme primitive. Si cela n'avait pas été le cas, il aurait été nécessaire de la définir pour chaque nouveau type de donnée. Plus de détails sur cette fonction seront donnés à la section 3.2.4.

La définition du langage qui suit est donnée comme rappel. Elle ne diffère pas des constructions habituelles.

Commençons par définir la syntaxe du langage. Soient X un ensemble dénombrable de variables et C_i (où i est un entier) une famille d'ensembles dénombrables de constantes d'arité i.

Nous commençons par définir l'ensemble des termes constitués uniquement de variables et de constructeurs (constantes). Nous les utilisons pour définir les motifs de filtrage.

DÉFINITION 1 (TERMES CONSTRUITS)
On définit l'ensemble des termes construits T_c *comme le plus petit ensemble tel que :*

- *si* $x \in X$, *alors* $x \in T_c$;
- *si* $c \in C_n$ *et* $t_1, \ldots, t_n \in T_c$, *alors* $c(t_1, \ldots, t_n) \in T_c$ *(on note* c *pour* $c()$ *si* $c \in C_0$*).*

Les valeurs construites sont des termes constitués uniquement de constantes.

DÉFINITION 2 (VALEURS CONSTRUITES)
On appelle valeur construite *un terme de T_c ne contenant aucune variable.*

On définit de façon mutuellement récursive les *termes* et les *cas de filtrage*.

DÉFINITION 3 (CAS DE FILTRAGE)
Un cas de filtrage *est un couple de terme noté $p \to t$ où p est un terme construit linéaire (chaque variable a au plus une occurrence) et t un* terme. *On appelle p le* motif de filtrage.

On définit maintenant l'ensemble des termes du langage miML.

DÉFINITION 4 (TERMES)
On définit l'ensemble des termes T *comme le plus petit ensemble tel que :*

- *si* $x \in X$, *alors* $x \in T$;
- *si* $c \in C_n$ *et* $t_1, \ldots, t_n \in T$, *alors* $c(t_1, \ldots, t_n) \in T$ *(on note c pour $c()$ si $c \in C_0$)* ;
- *si* $t_1, t_2 \in T$, *alors* $(t_1\ t_2) \in T$ *(application)* ;
- *si* $x \in X$ *et* $t \in T$, *alors* $(fun\ x \to t) \in T$ *(abstraction)* ;
- *si* $t_1, t_2 \in T$, *alors* $(t_1 = t_2) \in T$;
- *si* $t_1, t_2, t_3 \in T$, *alors* $(if\ t_1\ then\ t_2\ else\ t_3) \in T$ *(conditionnelle)* ;
- *si* $t \in T$, *alors* $(fix\ t) \in T$ *(combinateur de point fixe)* ;
- *si* $x \in X$ *et* $t_1, t_2 \in T$, *alors* $(let\ x = t_1\ in\ t_2) \in T$ *(définition locale)* ;
- *si* $p_1 \to t_1, \ldots, p_n \to t_n$ *sont des cas de filtrage et* $u \in T$, *alors*
$$\left(\begin{array}{llll} match & u & with & \\ & p_1 & \to & t_1 \\ | & p_2 & \to & t_2 \\ \vdots & \vdots & \vdots & \vdots \\ | & p_n & \to & t_n \end{array}\right) \in T.$$

La construction $(let\ x = t_1\ in\ t_2)$ est un lieur pour la variable x dans t_2, de même que $(fun\ x \to t)$ lie x dans t. En ce qui concerne le $match$, dans chacun des cas de filtrage, les variables présentes dans le motif (à gauche) sont liées dans le terme (à droite). Avec ces lieurs, on définit de manière usuelle le renommage et la notion de variables libres d'un terme.

Pour améliorer la lisibilité, on écrira

$$\begin{pmatrix} match & u & with & \\ & \overrightarrow{p} & \rightarrow & \overrightarrow{t} \end{pmatrix}$$

pour

$$\begin{pmatrix} match & u & with & \\ & p_1 & \rightarrow & t_1 \\ | & p_2 & \rightarrow & t_2 \\ \vdots & \vdots & \vdots & \vdots \\ | & p_n & \rightarrow & t_n \end{pmatrix}$$

quand cela sera possible.

Les termes sont considérés modulo renommage. C'est-à-dire que le nom attribué aux variables liées n'importe pas. Par exemple $(fun\ x \rightarrow x) \equiv (fun\ y \rightarrow y)$.

Substitution

Définissons maintenant la substitution (usuelle) pour les termes. Celle-ci est définie au renommage près (si une condition sur une variable liée empêche la substitution, il est possible de la renommer).

On commence par la substitution sur les *cas de filtrage*. Les variables contenues dans le motif sont liées dans la partie terme.

DÉFINITION 5 (SUBSTITUTION POUR LES CAS DE FILTRAGE)
On définit la substitution pour un cas de filtrage $(p \rightarrow t)[u/x]$ *où* p *est un motif de filtrage,* x *est une variable et* t,u *sont des termes.*

- *si* x *est une variable de* p*, alors* $(p \rightarrow t)[u/x] \equiv p \rightarrow t$ *;*
- *si* x *n'apparaît pas dans* p*, alors* $(p \rightarrow t)[u/x] \equiv p \rightarrow (t[u/x])$*, si aucune des variables de* p *n'apparaît libre dans* u*.*

On étend cette substitution à la liste de cas de filtrage.

La définition suivante est également usuelle.

DÉFINITION 6 (SUBSTITUTION POUR LES TERMES)
On définit la substitution pour les termes*, que l'on note* $t'[t/x]$ *(substitue* t *aux occurrences libres de* x *dans* t'*), récursivement sur la structure de* t*.*

- $x[t/x] \equiv t$ *;*

- $y[t/x] \equiv y$, *si* $x \not\equiv y$;
- $c(t_1, \ldots, t_n)[t/x] \equiv c(t_1[t/x], \ldots, t_n[t/x])$;
- $(t_1\ t_2)[t/x] \equiv (t_1[t/x]\ t_2[t/x])$;
- $(\mathit{fun}\ y \to t')[t/x] \equiv (\mathit{fun}\ y \to t'[t/x])$, *si* $x \not\equiv y$ *et si* y *n'apparaît pas dans* t ;
- $(t_1 = t_2)[t/x] \equiv (t_1[t/x] = t_2[t/x])$;
- $(\mathit{if}\ t_1\ \mathit{then}\ t_2\ \mathit{else}\ t_3)[t/x] \equiv (\mathit{if}\ t_1[t/x]\ \mathit{then}\ t_2[t/x]\ \mathit{else}\ t_3[t/x])$;
- $((\mathit{fix}\ t'))[t/x] \equiv ((\mathit{fix}\ (t'[t/x])))$;
- $(\mathit{let}\ y = t_1\ \mathit{in}\ t_2)[t/x] \equiv (\mathit{let}\ y = t_1[t/x]\ \mathit{in}\ t_2[t/x])$, *si* y *n'apparaît pas dans* t ;
- $(\mathit{match}\ u\ \mathit{with}\ \overrightarrow{p} \to \overrightarrow{t})[t/x] \equiv (\mathit{match}\ u[t/x]\ \mathit{with}\ (\overrightarrow{p} \to \overrightarrow{t})[t/x])$ *(on utilise la définition 5).*

On note $t[\overrightarrow{u}/\overrightarrow{x}]$ pour $((t[u_1/x_1])\ldots)[u_n/x_n]$, les substitutions étant successives.

3.2.2 Langage de types

Nous allons maintenant définir la syntaxe du langage de types de miML. Comme pour les langages ML habituels, on définit d'abord un langage de types contenant les types de bases (paramétrés par des types), les variables de types, les types fonctionnels et les n-uplets. On définit ensuite les schémas de types permettant de quantifier sur les variables de types. On distingue types et schémas de types de façon à ne pouvoir quantifier sur les variables de type qu'en tête des types (on ne peut donc pas écrire $\tau \to \forall\alpha.\tau'$).

Soit $\mathcal{X}^\tau$ l'ensemble dénombrable des variables de type et $\mathcal{C}_i^\tau$ où i est un entier, une famille d'ensembles dénombrables de constructeurs de types d'arité i. L'ensemble $\mathcal{C}_0^\tau$ contient au moins deux constantes distinctes : bool et Prop (nous n'utiliserons pas Prop avant la section 3.3.1).

DÉFINITION 7 (TYPES, SCHÉMAS DE TYPES)
On définit l'ensemble des types $\mathcal{T}_0$ *comme le plus petit ensemble tel que :*

- *si* $\alpha \in \mathcal{X}^\tau$, *alors* $\alpha \in \mathcal{T}_0$;
- *si* $c \in \mathcal{C}_n^\tau$ *et* $\tau_1, \ldots, \tau_n \in \mathcal{T}_0$, *alors* $([\tau_1, \ldots, \tau_n]\ c) \in \mathcal{T}_0$ *(on note* c *pour* $[\,]\ c$ *si* $c \in \mathcal{C}_0^\tau$*)* ;
- *si* $\tau_1, \tau_2 \in \mathcal{T}_0$, *alors* $(\tau_1 \to \tau_2) \in \mathcal{T}_0$;
- *si* $\tau_1, \ldots, \tau_n, \tau \in \mathcal{T}_0$, *alors* $\tau_1 * \cdots * \tau_n \to \tau \in \mathcal{T}_0$.

On définit l'ensemble des schémas de types $\mathcal{T}$ *comme :*
– *si* $\tau \in \mathcal{T}_0$, *alors* $\tau \in \mathcal{T}$;
– *si* $\tau \in \mathcal{T}_0$ *et* $\alpha_1, \ldots, \alpha_n \in \mathcal{X}^\tau$, *alors* $\forall \alpha_1, \ldots, \alpha_n . \tau \in \mathcal{T}$.

On peut définir la substitution dans les schémas de types, le quantificateur étant un lieur pour les variables de types.

Tout cela définit la syntaxe du langage miML. Dans la suite, nous définissons le typage de miML, puis sa sémantique opérationnelle.

3.2.3 Typage des termes

Les règles de typage permettent de restreindre les termes "légaux", par rapport à ceux que l'on peut construire avec la syntaxe de la définition 4. Le système de typage que nous présentons ici diffère peu des systèmes habituels, et est donné pour référence. La principale différence est la présence de variable de types *verrouillées*.

Soit $L \subset X$ un ensemble de variables dites *verrouillées*. Les variables verrouillées correspondent aux variables qu'il est interdit de généraliser ou d'instancier, car elles sont quantifiées universellement. Au niveau du typage, elles sont utilisées lors de la généralisation du type d'un `let ... in` (voir plus bas). Elles sont introduites par la règle "Pour-tout-type-gauche" (section 3.3.3).

Définissons maintenant l'environnement de typage. Soit $E : X \to \mathcal{T}$ une fonction partielle, l'*environnement de typage* pour les variables, et $K : (C_i)_{i \geq 0} \to \mathcal{T}$, l'*environnement de typage* pour les constantes, qui à une constante d'arité n associe son schéma de types $\forall \overrightarrow{\alpha} . \tau_1 * \cdots * \tau_n \to \tau$.

On commence par définir le typage sur les motifs de filtrage. L'objectif est de vérifier que chaque motif est typable et d'associer un type à chaque variable du motif. Ces règles de typage servent en fait à produire l'environnement pour typer les variables liées dans les motifs de filtrage.

On note le jugement de typage des motifs avec un exposant P pour le distinguer du typage des termes.

DÉFINITION 8 (TYPAGE DES MOTIFS DE FILTRAGE)
On dit qu'un terme t_c de T_c est bien typé de type τ et engendre l'environnement F si on peut dériver $F \models^P_{K,L} t : \tau$ (où $\tau \in \mathcal{T}_0$) en utilisant les règles de la figure page 75.

Les règles de typage des motifs se lisent : "t est de type τ en associant aux variables $x_1, \ldots, x_n$ les types $\tau'_1, \ldots, \tau'_n$, si les $t_1, \ldots, t_n$ sont de types $\tau_1, \ldots, \tau_n$ avec les associations variables – types $F_1, \ldots, F_n$". On les écrit :

$$\frac{F_1 \models^P_{K,L} t_1 : \tau_1 \quad \ldots \quad F_n \models^P_{K,L} t_n : \tau_n}{(x_1 : \tau'_1), \ldots, (x_n : \tau'_n) \models^P_{K,L} t : \tau}$$

On a maintenant tout ce qu'il faut pour définir le typage des termes.

DÉFINITION 9 (TYPAGE DES TERMES)
On dit qu'un terme t de T est bien typé et de type τ dans E si on peut dériver $E \models_{K,L} t : \tau$ (où $\tau \in \mathcal{T}$) en utilisant les règles des figures pages 76 et 77.

On a $gen(\tau, E, L) = \forall \overrightarrow{\alpha}.\tau$ où l'ensemble des α est l'ensemble des variables libres de τ auquel on enlève les variables de L et l'ensemble des variables libres dans le domaine de E.

La règle de typage ci-dessous se lit : "t est de type τ dans l'environnement E si $t_1, \ldots, t_n$ sont de types $\tau_1, \ldots, \tau_n$ dans $E_1, \ldots, E_n$".

$$\frac{E_1 \models_{K,L} t_1 : \tau_1 \quad \ldots \quad E_n \models_{K,L} t_n : \tau_n}{E \models_{K,L} t : \tau}$$

3.2.4 Évaluation

Après avoir défini les aspects statiques de notre langage de programmation, nous en définissons maintenant les aspects calculatoires. L'évaluation des termes de miML se comporte comme l'évaluation des termes de ML, à la différence notable de l'évaluation de la fonction d'égalité.

Une expression $t_1 = t_2$ se réduit en $true$ si t_1 et t_2 sont syntaxiquement égaux (au renommage près). Si t_1 et t_2 sont de la forme

$$\frac{}{(x:\tau) \models^P_{K,L} x : \tau}\ \text{P-Var}$$

$$\frac{K(c_n) = \forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \to \tau \quad F_1 \models^P_{K,L} t_1 : \tau_1[\overrightarrow{\tau}/\overrightarrow{\alpha}] \quad \cdots \quad F_n \models^P_{K,L} t_n : \tau_n[\overrightarrow{\tau}/\overrightarrow{\alpha}]}{\bigcup_{1 \le i \le n} F_i \models^P_{K,L} c_n(t_1, \ldots, t_n) : \tau[\overrightarrow{\tau}/\overrightarrow{\alpha}]}\ \text{P-Const}$$

FIG. 3.1 – Règles de typage des motifs de filtrage.

$$\frac{}{E, (x : \forall \overrightarrow{\alpha}.\tau_0) \models_{K,L} x : \tau_0[\overrightarrow{\tau}/\overrightarrow{\alpha}]}\ \text{Var}$$

$$\frac{K(c_n) = \forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \to \tau \quad E \models_{K,L} t_1 : \tau_1[\overrightarrow{\tau}/\overrightarrow{\alpha}] \quad \ldots \quad E \models_{K,L} t_n : \tau_n[\overrightarrow{\tau}/\overrightarrow{\alpha}]}{E \models_{K,L} c_n(t_1, \ldots, t_n) : \tau[\overrightarrow{\tau}/\overrightarrow{\alpha}]}\ \text{Const}$$

$$\frac{E \models_{K,L} t_1 : \tau_2 \to \tau \quad E \models_{K,L} t_2 : \tau_2}{E \models_{K,L} (t_1\ t_2) : \tau}\ \text{App}$$

$$\frac{E \oplus (x\ :\ \tau_1) \models_{K,L} t : \tau_2}{E \models_{K,L} (\mathit{fun}\ x \to t) : \tau_1 \to \tau_2}\ \text{Fun}$$

$$\frac{E \models_{K,L} t_1 : \tau_1 \quad E \oplus (x \mapsto \mathit{gen}(\tau_1, E, L)) \models_{K,L} t : \tau_2}{E \models_{K,L} (\mathit{let}\ x = t_1\ \mathit{in}\ t_2) : \tau_2}\ \text{Let}$$

$$\frac{E \models_{K,L} t_1 : \tau \quad E \models_{K,L} t_2 : \tau}{E \models_{K,L} t_1 = t_2 : \mathit{bool}}\ \text{Eq}$$

FIG. 3.2 – Règles de typage miML Partie 1

$$\frac{E \models_{K,L} t_1 : bool \quad E \models_{K,L} t_2 : \tau \quad E \models_{K,L} t_3 : \tau}{E \models_{K,L} (if\ t_1\ then\ t_2\ else\ t_3) : \tau}\ \textbf{If}$$

$$\frac{E \models_{K,L} t : \tau \to \tau}{E \models_{K,L} (fix\ t) : \tau}\ \textbf{Fix}$$

$$\frac{\begin{array}{llll} E & \models_{K,L} & u : \tau' & \\ F_1 & \models^P_{K,L} & p_1 : \tau' & E \oplus F_1 \quad \models_{K,L} \quad t_1 : \tau \\ & & \vdots & \vdots \\ F_n & \models^P_{K,L} & p_n : \tau' & E \oplus F_n \quad \models_{K,L} \quad t_n : \tau \end{array}}{E \models_{K,L} (match\ u\ with\ \overrightarrow{p} \to \overrightarrow{t})}\ \textbf{Match}$$

FIG. 3.3 – Règles de typage miML Partie 2

$c_1(c_2(\ldots c_n(\ldots)))$ où les c_i sont des constructeurs et que leurs "chaînes" de constructeurs diffèrent, alors on peut réduire l'égalité vers $false$. Dans les autres cas, on ne peut réduire l'expression : on ne sait pas encore décider de l'égalité. Les fonctions ne subissent pas de traitement particulier, on ne peut donc décider que de leur égalité syntaxique. Tout ceci est rendu nécessaire par le fait que dans les assistants de preuve, les termes que l'on doit évaluer ne sont, en général, pas clos. On a donné l'exemple plus haut de la formule $\forall x : nat.\exists y : nat.x = y$ pour laquelle il ne faut pas que l'évaluation puisse donner $\forall x : nat.\exists y : nat.false$ ($true$ et $false$ sont les valeurs du type $bool$).

Le traitement de la récursivité est identique à celui que l'on peut trouver dans les langages de programmations : il n'y a pas de gardes, ni de récurseurs, donc les programmes dont la récursion ne termine pas sont autorisés, et bouclent. De même, le filtrage de la construction `match ... with` n'est pas nécessairement exhaustif, donc un filtrage peut échouer. Dans ce cas, on ne peut pas évaluer l'expression plus avant.

Avant tout, nous devons définir ce qu'est le filtrage d'un terme par un motif. Le filtrage cherche une instanciation des variables du motif telle qu'il devienne égal au terme filtré.

DÉFINITION 10 (FILTRAGE)
La fonction partielle de filtrage $\nabla(p)(t)$ *où* $p \in T_c$ *et* $t \in T$ *renvoie un environnement (ensemble de paires)* $(x_i \mapsto t_i)$ *dans lequel les* x_i *et les* t_i *sont les associations variable/terme (la substitution) produite par le filtrage, au cas où celui-ci réussit.*

- $\nabla(x)(t) \equiv (x \mapsto t)$, ***si*** $x \in X$ ***;***
- $\nabla(c(p_1, \ldots, p_n))(c(t_1, \ldots, t_n)) \equiv \bigcup_{1 \leq i \leq n} \nabla(p_i)(t_i)$, ***si*** $c \in C$.

Tous les autres cas sont des cas d'échec.

Il est nécessaire de préciser que comme tous les motifs p sont linéaires, chaque variable ne peut apparaître au plus qu'une fois dans le domaine de l'environnement renvoyé.

Nous définissons d'abord une notion de valeur, c'est-à-dire de terme sur lequel on ne peut pas appliquer de règle d'évaluation. Cette définition de l'ensemble de valeur est plus complexe que la définition usuelle dans les languages de programmation. Nous avons dans notre système des variables logiques (par exemple liées

par un quantificateur) et il est nécessaire dans certains cas que l'évaluation s'arrête en les rencontrant : un exemple simple est $(if\ x\ then\ t_1\ else\ t_2)$.

DÉFINITION 11 (VALEURS)
On définit l'ensemble des valeurs *dans un environnement d'évaluation*[2] $N_{ML} : X \to T$ *comme le plus petit sous-ensemble de* T *tel que :*

- *si* x *est une variable et* x *n'est pas dans le domaine de* N_{ML}*, alors* x *est une valeur ;*
- *si* $v_1, \ldots, v_n$ *sont des valeurs et* c *un constructeur, alors* $c(v_1, \ldots, v_n)$ *est une valeur ;*
- *si* u *est une valeur et* $u \not\equiv c(t_1, \ldots, t_n)$*, alors* $(match\ u\ with\ \overrightarrow{p} \to \overrightarrow{t})$ *est une valeur ;*
- *si* u *est une valeur différente de* $true$ *et* $false$*, alors* $(if\ u\ then\ t_1\ else\ t_2)$ *est une valeur ;*
- $(fun\ x \to t)$ *est une valeur ;*
- *si* u_1 *et* u_2 *sont des valeurs et* $u_1 \not\equiv (fun\ x \to t)$*, alors* $(u_1\ u_2)$ *est une valeur ;*
- *si* u_1 *et* u_2 *sont des* valeurs non comparables *(cf. définition 12), alors* $u_1 = u_2$ *(la fonction d'égalité) est une valeur.*

La notion de *valeur comparable* est nécessaire pour l'évaluation de la fonction d'égalité. Nous définissons maintenant les *valeurs comparables*, c'est-à-dire les couples de valeurs pour lesquelles on sait décider de l'égalité ou de l'inégalité.

DÉFINITION 12 (VALEURS COMPARABLES)
On dit que deux valeurs v_1 *et* v_2 *sont* comparables *si :*

- *ou bien* $v_1 \equiv v_2$ *(égaux syntaxiquement),*
- *ou bien* v_1 *et* v_2 *sont des termes construits et ils diffèrent au niveau des constructeurs.*

Nous définissons l'évaluation comme une relation entre un environnement N_{ML} et deux termes t et t', notée $N_{ML} \models t \hookrightarrow t'$ qui se lit : le terme t est évalué en t' dans l'environnement d'évaluation N_{ML}.

Les règles données pour l'évaluation fixent la stratégie de réduction (c'est pour cette raison que l'on donne plusieurs règles pour chaque construction syntaxique) : comme pour les langages ML,

[2] On le note N_{ML} pour "environnement d'évaluation ML".

c'est une stratégie en appel par valeur, faible (on ne réduit pas sous les lieurs), et de gauche à droite pour les constructeurs et l'application. Le `if then else` est traité comme dans les langages de programmation, on ne réduit une des deux branches que lorsqu'on connaît la valeur de la condition.

Soit $N_{ML} : X \to T$ un environnement d'évaluation associant des termes aux variables.

DÉFINITION 13 (ÉVALUATION)
La relation d'évaluation $N_{ML} \Vdash t \hookrightarrow t'$ *est définie récursivement sur la structure de* t *par les règles données aux pages 81 à 83.*

Les règles d'évaluation se lisent : "t s'évalue en t' dans le contexte N_{ML} si $t_1, \ldots, t_n$ s'évaluent en $t'_1, \ldots, t'_n$ dans les contextes $N_{ML1}, \ldots, N_{MLn}$".

$$\frac{N_{ML1} \Vdash t_1 \hookrightarrow t'_1 \quad \cdots \quad N_{MLn} \Vdash t_n \hookrightarrow t'_n}{N_{ML} \Vdash t \hookrightarrow t'}$$

Les règles d'évaluations de miML préservent le typage :

PROPOSITION 1 (PRÉSERVATION DU TYPAGE)
Pour tout terme t*, si* t *est bien typé de type* τ *dans* E *et* K*, si* $N_{ML} \Vdash t \hookrightarrow t'$ *et si* N_{ML} *et* E *sont compatibles (c'est-à-dire que les valeurs des variables données par* N_{ML} *sont du type donné par* E*), alors* t' *est de type* τ *dans* E*.*

Le système de type étant similaire à celui proposé par Didier Rémy dans [58], on pourra se référer à ce document pour la preuve. Bien entendu, le système de type défini ne peut nous garantir la terminaison.

En ajoutant les définitions de type (cf. 3.4.1), ce langage forme un tout. Nous l'utilisons ici comme langage de termes dans le langage de spécification défini ci-après, mais on peut le considérer comme un langage de programmation à part entière, pour lequel on pourrait écrire un compilateur (en ajoutant évidemment des primitives d'entrée-sortie).

Nous pouvons maintenant aborder le langage de spécification en lui-même, c'est-à-dire le langage permettant d'écrire des formules parlant des termes de miML.

Les t représentent des termes et les v des valeurs.

$$\frac{N_{ML}(x) = t}{N_{ML} \models\!\Rightarrow x \hookrightarrow t}\ \text{Var-ev}$$

$$\frac{N_{ML} \models\!\Rightarrow t_1 \hookrightarrow t'_1}{N_{ML} \models\!\Rightarrow (t_1\ t_2) \hookrightarrow (t'_1\ t_2)}\ \text{App-1}$$

$$\frac{N_{ML} \models\!\Rightarrow t \hookrightarrow t'}{N_{ML} \models\!\Rightarrow (v\ t) \hookrightarrow (v\ t')}\ \text{App-2}$$

$$\frac{N_{ML} \oplus (x \mapsto v) \models\!\Rightarrow t_1 \hookrightarrow t_2}{N_{ML} \models\!\Rightarrow ((\mathit{fun}\ x \rightarrow t_1)\ v) \hookrightarrow t_2}\ \beta$$

$$\frac{N_{ML} \models\!\Rightarrow t \hookrightarrow t'}{N_{ML} \models\!\Rightarrow (\mathit{if}\ t\ \mathit{then}\ t_1\ \mathit{else}\ t_2) \hookrightarrow (\mathit{if}\ t'\ \mathit{then}\ t_1\ \mathit{else}\ t_2)}\ \text{If-cond}$$

$$\frac{N_{ML} \models\!\Rightarrow t_1 \hookrightarrow t'_1}{N_{ML} \models\!\Rightarrow (\mathit{if}\ \mathit{true}\ \mathit{then}\ t_1\ \mathit{else}\ t_2) \hookrightarrow t'_1}\ \text{If-true}$$

$$\frac{N_{ML} \models\!\Rightarrow t_2 \hookrightarrow t'_2}{N_{ML} \models\!\Rightarrow (\mathit{if}\ \mathit{false}\ \mathit{then}\ t_1\ \mathit{else}\ t_2) \hookrightarrow t'_2}\ \text{If-false}$$

FIG. 3.4 – Règles d'évaluation de miML Partie 1

$$\frac{N_{ML} \Vdash t_1 \hookrightarrow t_1'}{N_{ML} \Vdash t_1 = t_2 \hookrightarrow t_1' = t_2} \text{ Eq-ev-1}$$

$$\frac{N_{ML} \Vdash t \hookrightarrow t'}{N_{ML} \Vdash v = t \hookrightarrow v = t'} \text{ Eq-ev-2}$$

$$\frac{}{N_{ML} \Vdash v = v \hookrightarrow true} \text{ Eq-true}$$

$$\frac{}{N_{ML} \Vdash v = v' \hookrightarrow false} \text{ Eq-false}$$

si v et v' sont des *val. comparables* (déf. 12) et v différent de v'.

$$\frac{N_{ML} \Vdash t_1 \hookrightarrow t_1'}{N_{ML} \Vdash (let\ x = t_1\ in\ t_2) \hookrightarrow (let\ x = t_1'\ in\ t_2)} \text{ Let-1}$$

$$\frac{N_{ML} \oplus (x \mapsto v) \Vdash t \hookrightarrow t'}{N_{ML} \Vdash (let\ x = v\ in\ t) \hookrightarrow t'} \text{ Let-2}$$

$$\frac{N_{ML} \oplus (f \mapsto (fix\ (fun\ f \to t))) \Vdash t \hookrightarrow t'}{N_{ML} \Vdash (fix\ (fun\ f \to t)) \hookrightarrow t'} \text{ Fix-ev}$$

FIG. 3.5 – Règles d'évaluation de miML Partie 2

$$\frac{N_{ML} \models\!\Rightarrow u \hookrightarrow u'}{N_{ML} \models\!\Rightarrow (match\ u\ with\ \overrightarrow{p} \rightarrow \overrightarrow{t}) \hookrightarrow (match\ u'\ with\ \overrightarrow{p} \rightarrow \overrightarrow{t})}\ \text{Match-i}$$

$$\frac{N_{ML} \oplus \nabla(p_j)(v) \models\!\Rightarrow t_j \hookrightarrow t'_j}{N_{ML} \models\!\Rightarrow (match\ v\ with\ \overrightarrow{p} \rightarrow \overrightarrow{t}) \hookrightarrow t'_j}\ \text{Match-filtre}$$

Avec j la plus petite valeur pour laquelle p_j filtre v.

FIG. 3.6 – Règles d'évaluation de MIML Partie 3

3.3 Définition du langage de spécification

Nous commençons par définir le langage de formules que nous allons utiliser. Nous voyons ensuite les règles de typage permettant de limiter les formules bien formées. Nous décrivons enfin le comportement de ces formules à travers les règles de preuve.

3.3.1 Formules

Le langage logique que nous voulons utiliser est un calcul des prédicats du second ordre (cf. section 2.4), multisorté, dont les termes sont ceux du langage miML défini précédemment. Quelques différences rendent cependant notre système *ad hoc* pour la preuve de programme.

On ne définit pas dans le langage de prédicat $(t : \tau)$ exprimant que t est bien typé de type τ. Cela n'est pas nécessaire car on distingue le système de preuve du système de typage.

En revanche à chaque type ML, on associe un prédicat de typage *fort* $(t \mathbin{!} \tau)$ exprimant le typage *et* la terminaison. Ces prédicats peuvent être vus comme une intégration dans le langage de la notion de *terme réductible* (cf. Tait [61]), dans le sens où si τ est un type de base, cela signifie que le terme s'évalue en une valeur de ce type, et si τ est un type fonctionnel $(\tau_1 \to \tau_2)$, cela signifie que quelque soit son argument x tel que $(x \mathbin{!} \tau_1)$, on aura $((t\ x) \mathbin{!} \tau_2)$.

Contrairement au typage *à la ML* décrit en 3.2.3, on ne donne pas un jeu de règles de typage pour ces prédicats car on *prouvera* (*ie* par les méthodes habituelles, récurrence, etc.) qu'un terme termine effectivement en retournant une valeur du type adéquat. Du point de vue de l'utilisateur, cela n'est pas redondant avec le typage *à la ML*, car celui-ci peut être vérifier par un algorithme. Pour plus d'informations sur ces prédicats, on peut se référer à [4].

On différencie plusieurs quantifications, d'abord les quantifications (distinguées) au premier $\forall, \exists$ et second ordre $\mathbb{V}$. Puis une quantification universelle explicite sur les variables de type ML $\forall\_ : \mathrm{type}$. Le type de $\forall\_ : \mathrm{type}$ n'est pas lui-même un type mais une notation pour distinguer la quantification sur les variables de types.

Les quantificateurs sur les variables de termes sont bornés par un type. Ils ne quantifient que sur les termes *terminant* dans ce type, c'est-à-dire vérifiant $(t\ !\ \tau)$. Pour comprendre la raison de ce choix, donnons l'exemple de la fonction `map` prenant en argument une fonction et une liste et appliquant la fonction sur chaque élément de la liste. Le type de `map` est $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$. Nous voulons pouvoir prouver la totalité de `map`, mais cette fonction ne termine que si la fonction fournie en argument termine sur chacun des éléments de la liste. Nous devons donc n'autoriser la quantification que sur des termes dont l'évaluation termine ou sur des fonctions dont l'évaluation termine quelque soit ses arguments.

On distingue dans notre langage les booléens qui se trouvent au niveau des termes, des valeurs de vérité, au niveau formule. On ajoute au langage un prédicat $I(t)$, prenant un terme de type $bool$ (les booléens) et l'interprétant dans $Prop$ (les valeurs de vérités).

Une autre particularité de notre système est le traitement de l'égalité. Nous donnons des règles de réduction sur les termes et deux règles de preuve permettant de remplacer un terme par son réduit ou son expansé dans une preuve (cf. p. 93). C'est cela qui joue ici le rôle d'une théorie équationnelle (par exemple Leibniz) que l'on trouve dans d'autres systèmes (par exemple AF2). Nous avons vu dans la partie miML que nous avons une fonction d'égalité renvoyant une valeur de type $bool$, pour laquelle nous avons donné des règles de calcul. Nous pouvons utiliser le prédicat I pour manipuler la fonction d'égalité au niveau des formules. Typiquement, on écrira la commutativité de l'addition :
$\forall x\ !\ nat.\forall y\ !\ nat.I((add\ x\ y) = (add\ y\ x))$.

* * *

Soit $\mathbb{X}_i$ une famille d'ensembles dénombrables de variables de prédicats d'arité i. Soit $\mathbb{P}_i$ une famille d'ensembles dénombrables des constantes de prédicats d'arité i.

L'ensemble des constantes de types $\mathcal{C}_0^\tau$ contient une constante $Prop$ représentant le type des formules. On restreint l'ensemble $\mathcal{T}_0$ aux types tels que si $Prop$ apparaît dans τ, alors soit $\tau = Prop$, soit $\tau = \tau' \to Prop$ où $Prop$ n'apparaît pas dans τ'. De plus, on ne peut instancier une variable de type par $Prop$ et on ne peut utiliser $Prop$ pour typer un terme de miML.

On distinguera donc trois niveaux : le niveau des termes et variables de termes (dont le type ne contient pas $Prop$), les formules et prédicats (dont le type est de la forme $\tau_1 * \cdots * \tau_n \to Prop$) et les types et variables de types.

DÉFINITION 14 (FORMULES)
On définit l'ensemble Φ *des formules comme le plus petit ensemble tel que :*

- *si* $t_1, \ldots, t_n \in T$ *et* $X \in \mathbb{X}_n$, *alors,* $X(t_1, \ldots, t_n) \in \Phi$;
- *si* $t_1, \ldots, t_n \in T$ *et* $P \in \mathbb{P}_n$, *alors,* $P(t_1, \ldots, t_n) \in \Phi$;
- *si* $t \in T$, $I(t) \in \Phi$ *(l'interprétation d'un terme booléen)* ;
- *si* $\varphi_1, \varphi_2 \in \Phi$, *alors* $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \to \varphi_2$ *sont dans* Φ.
- *si* $x \in X$, $\tau \in \mathcal{T}$, $\varphi \in \Phi$, *alors* $(\forall x \,!\, \tau.\varphi) \in \Phi$ *et* $(\exists x \,!\, \tau.\varphi) \in \Phi$;
- *si* $\alpha \in \mathcal{X}^\tau$, $\varphi \in \Phi$, *alors* $(\forall \alpha : \mathrm{type}.\varphi) \in \Phi$;
- *si* $X \in \mathbb{X}$, $\varphi \in \Phi$, $\tau \in \mathcal{T}$, *alors* $(\mathbb{\forall} X : \tau.\varphi) \in \Phi$ *(quantification au second ordre)* ;
- *si* $t \in T$ *et* $\tau \in \mathcal{T}$, *alors* $(t \,!\, \tau) \in \Phi$ *(prédicat de typage et terminaison sur les termes).*

Les lieurs sont ici les quantificateurs $\forall$, $\exists$, $\mathbb{\forall}$ et $\forall\_ : \mathrm{type}$. On définit la notion de variable libre et de renommage dans les formules de façon usuelle. A partir de ces lieurs, on peut définir la substitution d'un terme dans une formule à toutes les occurrences libres d'une variable du premier ordre, ainsi que la substitution d'un type dans une formule à toutes les occurrences libres d'une variable de type.

On définit également les *fonctions propositionnelles* qui nous serviront pour instancier les variables du second ordre dont le type indique un prédicat ayant des arguments :

DÉFINITION 15 (FONCTIONS PROPOSITIONNELLES)
On définit l'ensemble des fonctions propositionnelles *tel que :*

- *si* $S \in \Phi$, *alors* S *est une fonction propositionnelle (d'arité 0)* ;
- *si* φ *est une formule,* $\tau_1, \ldots, \tau_n \in \mathcal{T}$ *où aucun des* τ_i *ne contient d'occurrence de* $Prop$, *alors* $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\varphi$ *est une fonction propositionnelle d'arité* n.

On peut maintenant définir la substitution (usuelle) d'une formule dans une formule. Comme pour les termes, on se place au renommage près.

DÉFINITION 16 (SUBSTITUTION DANS LES FORMULES)
On définit la substitution dans les formules, *que l'on note* $\varphi[S/X]$ *(substitue* S *à chaque occurrence libre de* X *dans* φ*), avec* S *une fonction propositionnelle d'arité* n*,* $X \in \mathbb{X}_n$ *et* $\varphi \in \Phi$ *récursivement sur la structure de* φ *:*

- $X(t_1, \ldots, t_n)[S/X] \equiv S[\overrightarrow{t}/\overrightarrow{x}]$ *où* S *est de la forme* $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\varphi$ *;*
- $Y(t_1, \ldots, t_q)[S/X] \equiv Y(t_1, \ldots, t_q)$ *où* $Y \in \mathbb{X}_q$ *et* $X \not\equiv Y$ *;*
- $P(t_1, \ldots, t_q)[S/X] \equiv P(t_1, \ldots, t_q)$ *où* $P \in \mathbb{P}_q$ *;*
- $(\varphi\ c\ \varphi')[S/X] \equiv (\varphi[S/X]\ c\ \varphi'[S/X])$ *où* $\varphi, \varphi' \in \Phi$ *et* c *un connecteur binaire ;*
- $(\neg\varphi)[S/X] \equiv \neg(\varphi[S/X])$ *où* $\varphi \in \Phi$ *;*
- $I(t)[S/X] \equiv I(t)$ *;*
- $(\forall\alpha : \text{type}.\varphi)[S/X] \equiv (\forall\alpha : \text{type}.\varphi[S/X])$ *;*
- $(\forall x\ !\ \tau.\varphi)[S/X] \equiv (\forall x\ !\ \tau.\varphi[S/X])$ *si* S *ne contient pas d'occurrence libre de* x *;*
- $(\exists x\,!\,\tau.\varphi)[S/X] \equiv (\exists x!\tau.\varphi[S/X])$ *si* S *ne contient pas d'occurrence libre de* x *;*
- $(\mathbb{\forall} X : \tau.\varphi)[S/X] \equiv (\mathbb{\forall} X : \tau.\varphi)$ *;*
- $(\mathbb{\forall} Y : \tau.\varphi)[S/X] \equiv (\mathbb{\forall} Y : \tau.\varphi[S/X])$ *si* S *ne contient pas d'occurrence libre de* X *;*
- $(t\ !\ \tau)[S/X] \equiv (t\ !\ \tau)$.

3.3.2 Typage des formules

Comme pour les termes de miML, nous définissons un système de typage pour les formules, et comme précédemment, celui-ci est décidable. Nous nous plaçons donc dans une logique multisortée avec des types polymorphes paramétriques. Les types des formules sont donc les types ML avec, comme on l'a vu dans la section précédente, un type de base *Prop* pour les valeurs de vérités. On n'autorise que les types ne contenant pas d'occurrence de *Prop*, le type *Prop* et les types de la forme $\tau \to Prop$ où τ ne contient pas d'occurrence de *Prop*. On ne peut instancier une variable de type avec un type contenant *Prop*.

Le système de typage des formules fait appel au système de typage des termes pour typer ceux-ci. L'environnement de typage complet doit donc contenir E (typage des variables de terme), K (typage des constructeurs) et L (variables verrouillées, cf. 3.2.3) pour

les propager dans le typage des termes, même si ces environnements ne sont pas utilisés dans le typage des formules proprement dit.

Soient $\mathcal{E} : \mathbb{X} \mapsto \mathcal{T}$ l'environnement de typage des variables du second ordre et $\mathcal{K} : \mathbb{P} \mapsto \mathcal{T}$ l'environnement de typage des prédicats.

DÉFINITION 17 (TYPAGE DES FORMULES)
On dit qu'une formule φ est bien typée *de type τ si l'on peut dériver $\mathcal{E}, E \Vdash_{\mathcal{K},K,L} \varphi : \tau$ à l'aide des règles pages 89 et 90.*

On dit qu'une formule est bien formée *si elle est typable de type $Prop$. On dit qu'une fonction propositionnelle $\lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\varphi$ est* bien formée *si elle est typable de type $\tau_1 * \cdots * \tau_n \to Prop$ où $Prop$ n'apparaît pas dans $\tau_1, \ldots, \tau_n$.*

Les règles de typage des formules se lisent de façon analogue aux règles de typage des termes. Quelques règles méritent cependant des explications : les règles de typage des quantificateurs du premier et second ordre (règles "Pour tout" et "Pour tout 2", figure 3.7 et 3.8) sont typées en vérifiant que la sous-formule a bien le type $Prop$, en supposant la variable du type indiqué.

La quantification sur une variable de type *verrouille* cette variable pour le typage de la sous-formule (règle "Pour tout type"). Elle ne pourra donc pas être généralisée lors du typage d'un let dans un sous-terme.

Pour le typage du prédicat de typage fort, on attend que le type du terme soit le type indiqué dans ce prédicat.

3.3.3 Système de preuve

Le système de preuve que nous utilisons est basé sur la déduction libre (classique) de Michel Parigot [51]. Les séquents manipulés sont des paires de multi-ensembles de formules (séquents multi-conclusions). Ce choix est justifié par des raisons relevant de l'implémentation : faciliter l'écriture de nouvelles tactiques à partir des règles de base du système et simplifier l'algorithme de vérification de la preuve.

On ajoute aux règles habituelles deux règles pour l'évaluation, et deux règles permettant d'ouvrir une définition, c'est-à-dire de

$$\frac{\mathcal{E}(X) = \forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \to \mathit{Prop} \quad E \models_{K,L} t_1 : \tau_1[\overrightarrow{\sigma}/\overrightarrow{\alpha}] \quad \ldots \quad E \models_{K,L} t_n : \tau_n[\overrightarrow{\sigma}/\overrightarrow{\alpha}]}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} X(t_1, \ldots, t_n) : \mathit{Prop}} \text{ Pred-Var}$$

$$\frac{\mathcal{K}(P) = \forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \to \mathit{Prop} \quad E \models_{K,L} t_1 : \tau_1[\overrightarrow{\sigma}/\overrightarrow{\alpha}] \quad \ldots \quad E \models_{K,L} t_n : \tau_n[\overrightarrow{\sigma}/\overrightarrow{\alpha}]}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} P(t_1, \ldots, t_n) : \mathit{Prop}} \text{ Pred-Const}$$

$$\frac{E \models_{K,L} t : \mathit{bool}}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} I(t) : \mathit{Prop}} \text{ I}$$

Pour chaque connecteur binaire c :

$$\frac{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} \varphi_1 : \mathit{Prop} \quad \mathcal{E}, E \Vdash_{\mathcal{K},K,L} \varphi_2 : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} \varphi_1 \ c \ \varphi_2 : \mathit{Prop}} \text{ Connecteur } c$$

$$\frac{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} \varphi : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} \neg\varphi : \mathit{Prop}} \text{ Non}$$

Pour chaque quantificateur $Q \in \{\forall, \exists\}$:

$$\frac{\mathcal{E}, E \oplus (x \mapsto \tau) \Vdash_{\mathcal{K},K,L} \varphi : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\mathcal{K},K,L} Qx \ ! \ \tau.\varphi : \mathit{Prop}} \text{ Quantificateur } Q$$

FIG. 3.7 – Typage des formules et fonctions propositionnelles

$$\frac{\mathcal{E}, E \Vdash_{\kappa,K,L\oplus\alpha} \varphi : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\kappa,K,L} \forall\alpha : \mathrm{type}.\varphi : \mathit{Prop}}\ \text{Pour tout type}$$

$$\frac{\mathcal{E} \oplus (X \mapsto \tau), E \Vdash_{\kappa,K,L} \varphi : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\kappa,K,L} \Whitestar X\ :\ \tau.\varphi : \mathit{Prop}}\ \text{Pour tout 2}$$

$$\frac{E \models_{\kappa,L} (t : \tau)}{\mathcal{E}, E \Vdash_{\kappa,K,L} (t\ !\ \tau) : \mathit{Prop}}\ \text{Terminaison Terme}$$

$$\frac{\mathcal{E} \oplus (x_1 \mapsto \tau_1) \oplus \cdots \oplus (x_n \mapsto \tau_n), E \Vdash_{\kappa,K,L} \varphi : \mathit{Prop}}{\mathcal{E}, E \Vdash_{\kappa,K,L} \lambda x_1 : \tau_1, \ldots, x_n : \tau_n.\varphi : \tau_1 * \cdots * \tau_n \to \mathit{Prop}}\ \text{Lambda}$$

FIG. 3.8 – Typage des formules et fonctions propositionnelles

substituer une formule à la variable sous le nom duquel elle a été définie (cf. “Ouvre-gauche” et “Ouvre-droit” figure 3.12 et la définition, section 3.5.2). On ajoute également les règles “XApp” et “XFun”, permettant de manipuler le prédicat de typage fort dans le cas de types fonctionnels. Les autres règles “inhabituelles” sont celles permettant de manipuler nos nouvelles constructions.

Les règles des quantificateurs montrent qu'on ne peut instancier une variable quantifiée que par un terme qui termine. On ne peut instancier une variable de type qu'avec un type. La règle permettant d'instancier une variable quantifiée au second ordre par une fonction propositionnelle invoque le système de typage des formules.

On définit l'environnement $\mathbb{N}$ associant à une variable de $\mathbb{X}_n$ une fonction propositionnelle d'arité n. Cet environnement contient les *définitions* de formules et fonctions propositionnelles (cf. 3.5.2).

Par souci d'allègement de la notation, on note Env l'environnement qui est en réalité le 7-uplet suivant : $(E, K, L, \mathcal{E}, \mathcal{K}, N_{ML}, \mathbb{N})$. On notera donc les séquents $\Gamma \vdash_{Env} \Delta$ (où Γ et Δ sont des multi-ensembles de formules), la preuve se faisant dans un environnement. Encore une fois, toutes ces fonctions ne sont pas forcément manipulées dans le système de règles de preuve, mais peuvent être répercutés dans les systèmes vus précédemment (typage, évaluation) auxquels fait appel le système de preuve.

DÉFINITION 18 (PREUVE)
On dit qu'un séquent $\Gamma \vdash_{Env} \Delta$ *est* prouvable *si* Γ *et* Δ *sont des multi-ensembles de formules bien formées (de type* $Prop$*), et que l'on peut le dériver à partir des règles des pages 93 à 96.*

On dit qu'une formule φ *est* prouvable *sous les hypothèses* Γ *si le séquent* $\Gamma \vdash_{Env} \varphi$ *est prouvable.*

Les règles de preuves se lisent : “pour prouver Δ sous les hypothèses Γ, il faut prouver $\Delta_1, \ldots, \Delta_n$ sous les hypothèses $\Gamma_1, \ldots, \Gamma_n$.

$$\frac{\Gamma_1 \vdash_{Env} \Delta \quad \cdots \quad \Gamma_n \vdash_{Env} \Delta_n}{\Gamma \vdash_{Env} \Delta}$$

Les règles venant de la déduction libre étant toutes des règles d'élimination, on ne note pas dans les règles les formules de Γ et Δ qui n'interviennent pas explicitement. Celles-ci sont simplement

recopiées des prémisses vers les conclusions (cette convention est usuelle dans la déduction libre, voir [51]).

Cela conclut la définition du système logique "pur". Dans ce système, il n'y a pas de règle de récurrence, et rien ne permet de prouver qu'un constructeur c satisfait $c\ !\ \tau$. Il n'est pas encore permis d'utiliser la fonction d'égalité comme on le ferait dans les systèmes habituels (remplacement d'un égal par un égal). Nous allons maintenant combler ces manques au niveau de la théorie.

3.4 Théorie

La théorie que l'on présente ici est "évolutive". C'est-à-dire qu'elle permet l'ajout de types et de constructeurs, ce qui aura pour effet l'ajout de nouvelles règles pour les traiter.

3.4.1 Compléments sur les types

Les définitions de types sont similaires à celles des types sommes dans ML. Une définition de type a la forme :

$$\begin{array}{lcl} \text{type } [\overrightarrow{\alpha}]\ \tau & = & \\ & & c_0 \\ & | & c_1 \\ & \vdots & \vdots \\ & | & c_i \quad \text{of} \quad \tau_{i,1} * \cdots * \tau_{i,k} \\ & \vdots & \vdots \\ & | & c_n \quad \text{of} \quad \tau_{n,1} * \cdots * \tau_{n,l} \end{array}$$

et définit le type τ paramétré par les variables de types $\overrightarrow{\alpha}$, ainsi que les constructeurs d'arité 0 (c_0, c_1,...) de type τ et des constructeurs comme c_i, k-aire de type $\tau_{i,1} * \cdots * \tau_{i,k} \rightarrow \tau$.

On donne un critère syntaxique pour admettre qu'un type récursif est bien fondé : toutes les occurrences de τ dans les types des constructeurs ne doivent apparaître qu'en position positive.

DÉFINITION 19 (POSITIVITÉ)

– *Pour un constructeur de type $\tau_1 * \cdots * \tau_n$, les $\tau_1, \ldots, \tau_n$ sont à positions positives.*

$$\frac{}{A \vdash_{Env} A}\ \text{Axiome}$$

$$\frac{\varphi[t] \vdash_{Env} \quad \vdash_{Env} \varphi[t'] \quad N_{ML} \models\!\!\Rightarrow t \hookrightarrow t'}{\vdash_{Env}}\ \text{Éval-gauche}$$

$$\frac{\vdash_{Env} \varphi[t] \quad \varphi[t'] \vdash_{Env} \quad N_{ML} \models\!\!\Rightarrow t \hookrightarrow t'}{\vdash_{Env}}\ \text{Éval-droite}$$

$$\frac{A \wedge B \vdash_{Env} \quad \vdash_{Env} A \quad \vdash_{Env} B}{\vdash_{Env}}\ \text{Et-gauche}$$

$$\frac{\vdash_{Env} A \wedge B \quad A \vdash_{Env}}{\vdash_{Env}}\ \text{Et-droite-1}\ \textit{idem}\ \text{pour B.}$$

$$\frac{A \vee B \vdash_{Env} \quad \vdash_{Env} A}{\vdash_{Env}}\ \text{Ou-gauche-1}\ \textit{idem}\ \text{pour B.}$$

$$\frac{\vdash_{Env} A \vee B \quad A \vdash_{Env} \quad B \vdash_{Env}}{\vdash_{Env}}\ \text{Ou-droite}$$

$$\frac{A \rightarrow B \vdash_{Env} \quad A \vdash_{Env} B}{\vdash_{Env}}\ \text{Implique-gauche}$$

On ne note pas dans les règles les formules de Γ et Δ qui n'interviennent pas. Celles-ci sont simplement recopiées des prémisses vers les conclusions.

FIG. 3.9 – Règles de preuve Partie 1

$$\frac{\vdash_{Env} A \to B \quad \vdash_{Env} A \quad B \vdash_{Env}}{\vdash_{Env}} \text{ Implique-droite}$$

$$\frac{\neg A \vdash_{Env} \quad A \vdash_{Env}}{\vdash_{Env}} \text{ Non-gauche}$$

$$\frac{\vdash_{Env} \neg A \quad \vdash_{Env} A}{\vdash_{Env}} \text{ Non-droite}$$

$$\frac{\forall x \mathbin{!} \tau.A \vdash_{Env} \quad (y \mathbin{!} \tau) \vdash_{Env[E \oplus (y \mapsto \tau)/E]} A[y/x]}{\vdash_{Env}} \text{ Pour tout-gauche}$$

À condition que y ne soit pas libre dans le séquent et l'environnement.

$$\frac{\vdash_{Env} \forall x \mathbin{!} \tau.A \quad A[t/x] \vdash_{Env} \quad \vdash_{Env} (t \mathbin{!} \tau)}{\vdash_{Env}} \text{ Pour tout-droite}$$

$$\frac{\exists x \mathbin{!} \tau.A \vdash_{Env} \quad \vdash_{Env} A[t/x] \quad \vdash_{Env} (t \mathbin{!} \tau)}{\vdash_{Env}} \text{ Il existe-gauche}$$

FIG. 3.10 – Règles de preuve Partie 2

$$\frac{\vdash_{Env} \exists x\ !\ \tau.A \quad (y\ !\ \tau) \vdash_{Env[E\oplus(y\mapsto\tau)/E]} A[y/x]}{\vdash_{Env}}$$ Il existe-droite

À condition que y ne soit pas libre dans le séquent et l'environnement.

$$\frac{\forall\alpha : \text{type}.A \vdash_{Env} \quad \vdash_{Env[L\oplus\gamma/L]} A[\gamma/\alpha]}{\vdash_{Env}}$$ Pour tout type-gauche

À condition que γ ne soit pas libre dans le séquent et l'environnement.

$$\frac{\vdash_{Env} \forall\alpha : \text{type}.A \quad A[\tau/\alpha] \vdash_{Env}}{\vdash_{Env}}$$ Pour tout type-droite Avec $\tau \in \mathcal{T}_0$.

$$\frac{\Bbbw X\ :\ \tau.A \vdash_{Env} \quad \vdash_{Env[\mathcal{E}\oplus(Y\mapsto\tau)/\mathcal{E}]} A[Y/X]}{\vdash_{Env}}$$ Pour tout 2-gauche

À condition que Y ne soit pas libre dans le séquent et l'environnement.

$$\frac{\vdash_{Env} \Bbbw X\ :\ \tau.A \quad A[S/X] \vdash_{Env} \quad \mathcal{E}, E \Vdash_{\mathcal{K},K,L} (S : \tau)}{\vdash_{Env}}$$ Pour tout 2-droite

S est une fonction propositionnelle.

$$\frac{I(true) \vdash_{Env}}{\vdash_{Env}}$$ I-gauche

FIG. 3.11 – Règles de preuve Partie 3

$$\frac{\vdash_{Env} I(false)}{\vdash_{Env}} \text{ I-droite}$$

$$\frac{\varphi[X] \vdash_{Env} \quad \vdash_{Env} \varphi[\mathbb{N}(X)]}{\vdash_{Env}} \text{ Ouvre-gauche}$$

Ouvre la définition d'une formule.

$$\frac{\vdash_{Env} \varphi[X] \quad \varphi[\mathbb{N}(X)] \vdash_{Env}}{\vdash_{Env}} \text{ Ouvre-droite}$$

idem.

$$\frac{\vdash_{Env} t \mathbin{!} \tau' \quad \vdash_{Env} u \mathbin{!} \tau' \to \tau}{\vdash_{Env} t\ u \mathbin{!} \tau} \text{ XApp}$$

$$\frac{x \mathbin{!} \tau' \vdash_{Env} t \mathbin{!} \tau}{\vdash_{Env} fun\ x \to t \mathbin{!} \tau' \to \tau} \text{ XFun}$$

$$\frac{K(c_n) = \forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \to \tau \quad \vdash_{Env} t_1 \mathbin{!} \tau_1[\overrightarrow{\tau}/\overrightarrow{\alpha}] \quad \ldots \quad \vdash_{Env} t_n \mathbin{!} \tau_n[\overrightarrow{\tau}/\overrightarrow{\alpha}]}{\vdash_{Env} c_n(t_1, \ldots, t_n) \mathbin{!} \tau[\overrightarrow{\tau}/\overrightarrow{\alpha}]} \text{ XConst}$$

FIG. 3.12 – Règles de preuve Partie 4

- *Si $\tau \to \tau'$ est à une position positive (resp. négative), alors τ est à une position négative (resp. positive) et τ' est à une position positive (resp. négative).*
- *Le produit préserve la polarité.*
- *Si $[\tau]\ \tau'$ est à une position positive (resp. négative), alors τ et τ' sont à des positions positives (resp. négative).*

Chaque passage à gauche d'une flèche de type fonctionnel inversant la polarité. La polarité est positive au début du parcours. Par exemple, si on définit le type τ, on admettra un constructeur de type $\tau' \to \tau$, mais pas $\tau \to \tau'$.

Les seules variables de types pouvant apparaître dans les définitions des constructeurs sont les paramètres $\overrightarrow{\alpha}$.

Lors de la définition d'un nouveau type, les constructeurs sont rangés dans le C_i correspondant à leur arité, et on associe à chacun sont type dans K.

À partir de la définition d'un type, il faut engendrer la règle d'induction correspondante, les règles de typage des constructeurs, des règles pour l'égalité.

Pour construire l'induction structurelle, on construit une formule du second ordre exprimant le principe d'induction :

- On interprète la définition de τ comme une formule :

$$[\![\text{type } [\overrightarrow{\alpha}]\tau = c_1|\cdots|c_p]\!]^{Ind} \equiv$$
$$\forall\!\!\forall X : [\overrightarrow{\alpha}]\tau \to \mathit{Prop}.\ \textstyle\bigwedge_i [\![c_i \text{ of } \tau_{i,1} * \cdots * \tau_{i,n}]\!]^{Ind}_{\epsilon} \to \forall l\ !\ [\overrightarrow{\alpha}]\tau.X(l)$$

- Si τ_1 n'est pas le type défini :
  $[\![c_i \text{ of } \tau_1 * \cdots * \tau_n]\!]^{Ind}_p \equiv \forall x_1\ !\ \tau_1.[\![c_i \text{ of } \tau_2 * \cdots * \tau_n]\!]^{Ind}_{x_1,p}$
- Si τ est le type défini :
  $[\![c_i \text{ of } \tau * \cdots * \tau_n]\!]^{Ind}_p \equiv \forall x_1\ !\ \tau_1.X(x_1) \to [\![c_i \text{ of } \tau_2 * \cdots * \tau_n]\!]^{Ind}_{x_1,p}$
- $[\![c_i \text{ of}]\!]^{Ind}_p \equiv X(c_i(p))$

Cette formule, que l'on calcule pour chaque nouveau type de donnée, est posée comme axiome. Elle correspond exactement à la règle de récurrence structurelle, sous la forme d'une formule du second ordre.

Prenons comme exemple les listes polymorphes : Le type des listes est défini comme

$$\begin{array}{rcl} \text{type } [\alpha]list & = & \\ & & \mathit{Nil} \\ & | & \mathit{Cons} \quad \text{of} \quad \alpha * [\alpha]list \end{array}$$

soit une liste est vide (Nil), soit elle est constituée d'une valeur suivie d'une liste ($Cons$).

On engendre le principe d'induction, exprimant : "si une propriété est vraie pour Nil, si en la supposant vraie pour tout l', on la démontre pour $Cons(x, l')$, alors elle est vraie pour toute liste". En pratique, nous ne souhaitons pas utiliser la formule du second ordre, mais une règle d'induction. On peut l'obtenir à partir de cette formule. On a donc la règle suivante, qui est équivalente :

$$\frac{\vdash_{Env} \varphi[Nil/l] \quad \vdash_{Env} \forall x\ !\ \alpha.\forall l'\ !\ [\alpha]\ list.\varphi[l'/l] \rightarrow \varphi[Cons(x,l')/l]}{\vdash_{Env} \forall l\ !\ [\alpha]\ list.\varphi}\ rec_{list}$$

On ajoute les axiomes suivants exprimant le fait que l'évaluation d'un constructeur termine.

$(Nil\ !\ [\alpha]\ list)$ et $(Cons\ !\ \alpha * [\alpha]list \rightarrow [\alpha]list)$.

On ajoute également les axiomes suivants pour l'égalité :

$I(Nil = Nil)$ et
$\forall x, x'!\alpha.\forall l, l'![\alpha]list.I(x = x') \wedge I(l = l') \rightarrow I(Cons(x, l) = Cons(x', l'))$.

3.4.2 Égalité

Ce système est muni d'une "théorie de l'évaluation" et d'une fonction calculant l'égalité structurelle entre deux termes. Cette fonction renvoie une valeur de type $bool$, mais peut être manipulée comme un prédicat grâce au prédicat I d'interprétation des booléens dans les valeurs de vérité ($Prop$). On ne trouve cependant pas dans la logique de base de règles permettant de manipuler cette égalité comme on a l'habitude de le faire (Leibniz). Il est donc nécessaire de poser l'axiome (du second ordre) suivant :

$$\forall \alpha : \text{type}.\Bbbw X : \alpha \rightarrow Prop.\forall x\ !\ \alpha.\forall y\ !\ \alpha.(I(x = y) \rightarrow X(x) \rightarrow X(y))$$

Cet axiome et les règles d'évaluation de l'égalité suffisent pour montrer la symétrie, la réflexivité et la transitivité. Il est inteéressant de remarquer que la quantification bornée impose que les termes substitués à x et y soient fortement typés.

Il nous reste à ajouter à tout ceci les éléments nécessaires pour pouvoir travailler dans le système : de quoi définir, déclarer, énoncer des théorèmes, etc.

3.5 Langage Vernaculaire

On appelle langage vernaculaire les extensions au langage logique permettant de modifier l'environnement. Le langage vernaculaire que nous définissons ici se compose de six constructions : type permet de définir de nouveaux types de donnée et let des termes de miML ; Axiom et Theorem énoncent respectivement un axiome et un théorème ; Declare déclare une constante et Define permet de définir une formule ou une fonction propositionnelle.

3.5.1 Déclaration

La déclaration est l'élément le plus simple du langage, elle a la forme : Declare $c : \tau$ où $\tau \in \mathcal{T}$, et soit τ est de la forme $\tau_1 * \cdots * \tau_n \rightarrow \mathit{Prop}$, soit Prop n'apparaît pas dans τ. Son effet est de créer une nouvelle constante dont on calcule l'arité à partir du type. Cette constante est ajoutée dans $\mathbb{P}_n$ ou C_n (en fonction du type, si Prop apparaît ou non) ; de même, $(c \mapsto \tau)$ est ajouté dans K ou $\mathcal{K}$.

3.5.2 Définition

La définition Define $X = \varphi$ définit X et lui lie φ. Ou bien $\varphi \in \Phi$, ou bien φ est une fonction propositionnelle

On commence par déterminer (en utilisant les règles de typage) le type τ de φ et son arité n. On range alors X dans $\mathbb{X}_n$ et on ajoute $(X \mapsto \tau)$ à $\mathcal{E}$ et $(X \mapsto \varphi)$ à $\mathbb{N}$.

Si la formule n'est pas bien typée de type $\forall \overrightarrow{\alpha}.\tau_1 * \cdots * \tau_n \rightarrow \tau$, la définition n'est pas valide.

3.5.3 Axiome

Les axiomes sont énoncés par $\mathrm{Axiom}\ n : \varphi$ où φ est une formule (de type *Prop*) et n est le nom donné à l'axiome. Les axiomes sont ajoutés dans le contexte de preuve Γ.

3.5.4 Théorème

Les théorèmes sont énoncés comme les axiomes mais on utilise la construction $\mathrm{Theorem}\ n : \varphi\,;\, \Pi$ où Π est la preuve du séquent $\Gamma \vdash_{Env} \varphi$ dans lequel Γ est la liste des axiomes et théorèmes connus.

Comme les axiomes, les théorèmes sont ajoutés dans Γ.

3.5.5 Let

Les termes de miML doivent être définis avec la construction $\mathrm{let}\ x = t$ où t est un terme bien typé de type τ. Le let se comporte comme la définition, excepté que l'on ajoute x dans X_n (n étant l'arité de t) et $(x \mapsto t)$ à N_{ML}.

3.5.6 Type

Les définitions de types permettent de créer un nouveau type somme à la ML, et pose automatiquement les axiomes correspondant (induction, typage des constructeurs et égalité) comme décrit en 3.4.1.

3.5.7 Session

Nous avons besoin de définir un objet que nous appelons *session* qui correspond à une suite de commandes du langage vernaculaire, accompagnées de leurs environnement.

DÉFINITION 20 (SESSION)
On appelle session *une suite de quintuplets*

$$(Env_0, \Gamma_0, O_0, Env'_0, \Gamma'_0), \ldots, (Env_n, \Gamma_n, O_n, Env'_n, \Gamma'_n)$$

tels que :

- Γ_0 *est un ensemble de formules bien typées,* Env_0 *est un environnement (au sens vu en 3.3.3) bien formé (tout ce qui le compose est bien typé). Tous les deux étant arbitraires.*
- O_i *est une commande du langage vernaculaire correcte, interprétée dans le contexte* (Env_i, Γ_i)*.*
- Env'_i *et* Γ'_i *sont identique à* Env_i *et* Γ_i *augmentés des effets de la commande* O_i*.*
- $(Env_{i+1}, \Gamma_{i+1}) = (Env'_i, \Gamma'_i)$*.*

Cela clos la formalisation du langage que nous implémentons dans PAF!. Nous avons un langage de termes, qui est un langage de programmation simplifié, un langage de formules, qui nous permet de spécifier et de décrire des propriétés des programmes miML. Le langage vernaculaire et la session achèvent de nous donner les outils nécessaires à l'utilisation du système.

3.6 Conclusion

Le travail d'investigation de la méta-théorie de notre système reste à faire. On peut se poser deux questions : quel est le pouvoir d'expression de la théorie ; la théorie est-elle cohérente ?

De manière évidente, ce système a au moins le pouvoir d'expression de l'arithmétique fonctionnelle du second ordre, mais dépasse-t-on cette théorie ? En effet, la quantification est possible sur les fonctions ML d'ordre supérieur. Cependant on a séparé clairement le langage de terme et le langage de formule : une fonction même polymorphe ne peut prendre en argument de valeurs de type $Prop$; de même, si on autorise la quantification sur les variables de types, on ne permet pas d'instancier ces variables avec un type contenant $Prop$. Pour être plus précis, la question qui se pose est : arrive-t-on au niveau de l'arithmétique fonctionnelle d'ordre supérieur ?

Pour ce qui est de la cohérence, si l'on se place dans un cadre restreint de notre système dans lequel tous les termes sont *fortement typables*, on est alors proche d'une logique d'ordre supérieure avec deux extensions : la plus importante est le langage de termes, pour lequel nous avons établi que le typage fort implique la terminaison (voir [4] dans un langage de termes simplifié). Pour la

confluence, la preuve reste encore à formaliser, mais, dans la mesure où notre langage est très proche de (par exemple) PCF [53], cela ne devrait pas poser de problèmes. La seconde extension est l'injection des booléens dans les valeurs de vérité. Les termes se réduisant à des valeurs, nous pouvons considérer le prédicat I comme dénotant l'ensemble des termes se réduisant en $true$. Pour ces deux extensions, il semble donc possible de construire un modèle nous donnant la cohérence du système.

La vraie question qui se pose est "que se passe-t-il lorsque l'on ajoute des termes dont l'évaluation ne termine pas ?" La réponse à cette question devra être apportée par un travail ultérieur, mais on peut déjà donner des arguments en faveur de la cohérence : on ne peut instancier les quantificateurs du premier ordre avec des termes dont l'évaluation ne termine pas (voir les règles *Pour tout droite* et *Il existe gauche*) ; il semble que l'on ne puisse rien faire "d'utile" (ou de "nuisible") avec les formules contenant un tel terme, puisqu'une formule atomique qui contient un tel terme n'apporte pas plus d'informations qu'une variable propositionnelle.

Chapitre 4

Spécification d'un système d'aide à la preuve client-serveur

Nous commençons par décrire l'architecture globale du programme, puis nous décrivons le protocole de communication entre l'interface et le moteur de preuve. À la section 4.3 nous spécifions le moteur de preuve. Enfin, nous terminons en 4.4 par la spécification de l'interface.

La spécification que nous donnons ici est indépendante du formalisme décrit au chapitre 3 : elle s'attache à décrire le comportement attendu du moteur de preuve et de l'interface utilisateur.

4.1 Modèle client-serveur

Nous faisons le choix d'une architecture client-serveur, offrant un certain nombre d'avantages. Le premier de ces avantages est méthodologique : cela oblige à bien distinguer ce qui relève de l'interface et ce qui relève du moteur de preuve en figeant les possibilités qu'ils ont de se transmettre des informations. De plus, cela offre la possibilité de disposer de plusieurs interfaces différentes, sans avoir à se plonger dans le code du serveur pour les implémenter (il suffit de lire la documentation du protocole. . .). Troisièmement, même si cette possibilité n'a pas été exploitée ici, cela permet-

trait de travailler en concurrence, à plusieurs personnes tentant de prouver un théorème, chacun s'occupant des branches de la preuve qui lui sont affectées, ou plus trivialement, de continuer à travailler sur d'autres parties d'une preuve (ou d'autres preuves), pendant qu'une procédure de décision longue est à l'œuvre sur l'un des buts. Enfin, cela permet d'utiliser le système comme "service de preuve" pour un autre programme.

Nous ne parlons pas ici de concurrence ni des aspects qui en relèvent (verrous, etc.).

La figure 4.1 montre l'architecture globale du système : d'un côté, le moteur de preuve[1] avec ses tactiques (le plus modulaire possible), de l'autre, l'interface et le protocole PEP (Proof Engine Protocol, voir 4.2) entre les deux.

FIG. 4.1 – Architecture globale

Comme vu en 3.5.7, le plus gros objet manipulé est la session, et se décompose en suites d'éléments correspondant à des commandes du langage vernaculaire que nous appelons ici *éléments de session*. Comme dans la formalisation, une session est une suite d'éléments de session. Les preuves sont vues ici comme des arbres dont les nœuds sont appelés *éléments de preuve* (cf. figure 4.2). La racine de la preuve est conservée dans un élément de session particulier.

Ce schéma représente l'agencement des éléments de preuve et des éléments de session. Les rectangles liés entre eux par des

[1] Nous utilisons indifféremment les termes "moteur", "moteur de preuve", "serveur" pour décrire cette partie du programme.

FIG. 4.2 – Exemple de session

flèches représentent les éléments de session. Ces éléments ont pour but de construire un environnement (chaîné). Un utilisateur doit pouvoir supprimer un élément de session, ou en insérer un autre à la fin, ou entre deux éléments existants.

Les éléments de preuve forment des arbres. On remarque que les branches incomplètes des preuves se terminent par un marqueur (point d'interrogation). Une preuve est complète lorsqu'il ne reste plus de sous-buts ouverts. Du point de vue de l'insertion, on ne peut ajouter un élément de preuve qu'à la place d'un point d'interrogation (ce qui a pour effet de créer des sous-buts, s'il y en a). La suppression est possible en n'importe quel point de la preuve, et elle détruit tous les descendants du nœud concerné.

À chaque élément de session on associe un identificateur, l'identificateur 0 étant réservé.

Une remarque importante que l'on peut faire sur ce modèle est qu'il rend très problématique l'utilisation de variables existentielles (ou méta-variables), c'est-à-dire de variables implicitement quantifiées existentiellement, pouvant être instanciées quand leur valeur est connue. La suppression d'un élément de preuve ayant instancié une de ces variables doit la "désinstancier" (ce problème est notoirement difficile dans le cas du *undo* pour les systèmes de preuve usuels). Un autre problème se poserait du point de vue de l'interface pleine page : doit on raffraichir l'affichage de manière à modifier la valeur de la variable partout lorsque celle-ci est instanciée ? Dans la spécification que nous donnons et dans l'implémentation, nous avons choisi de ne pas utiliser de méta-variables.

4.2 Protocole de communication

Nous commençons par donner la spécification des messages entre l'interface et le moteur de preuve. Nous décrirons ensuite en 4.3 et 4.4 chacun de ces deux éléments.

Le protocole PEP (Proof Engine Protocol) est synchrone, et c'est l'interface qui initie le dialogue. On a donc une alternance de questions de l'interface (client) et de réponses du moteur de preuve (serveur).

La description que nous faisons ici est "de haut niveau", nous ne

détaillons pas la représentation des paquets, les codes de détection d'erreurs (*checksum*), etc. Pour plus d'information à ce sujet, et pour la définition des paquets, voir [37]. Nous supposons ici que les transmissions se font sans erreurs.

Les messages du protocole de communication sont regroupés en quatre classes. La classe `Protocol` contient les messages "administratifs" échangés entre l'interface et le moteur de preuve. La classe `Config`, n'est pour l'instant pas utilisée et est dévolue aux échanges d'information sur la configuration. La classe `Session` regroupe toutes les commandes d'insertion, de suppression et de modification d'éléments de session, ainsi que les réponses correspondantes. De même que la classe `Proof` au niveau des preuves.

4.2.1 Classe `Protocol`

La classe `Protocol` définit les messages d'initialisation et de fin du dialogue, elle contient deux requêtes et deux réponses.

4.2.1.1 Requêtes

`PRO.OPEN` Commande d'ouverture de connexion.

`PRO.QUIT` Fermeture de la connexion.

4.2.1.2 Réponses

`PRO.OK` Acquittement positif de réception.

`PRO.ERROR` Acquittement négatif (erreur), contenant un code d'erreur.

L'information véhiculée par ces messages contient entre autre le numéro de version du protocole.

4.2.2 Classe `Session`

La classe session permet d'ajouter, supprimer et modifier des éléments de session. On utilise pour ces requêtes le type de l'élément de session, qui peut être `Axiom`, `Definition`, `Declaration`, `Let`, `Type` ou `Theorem`.

4.2.2.1 Requêtes

`SES.ADD` Insertion d'un élément de session. Cette requête a quatre paramètres : l'identificateur de l'élément de session (attribué par l'interface), l'identificateur de l'élément de session après lequel se fait l'insertion (ou 0 si l'insertion se fait au début), le type de l'élément inséré et les données attachées à ce type.

`SES.DEL` Suppression d'un élément de session. Les paramètres sont l'identificateur et le type de l'élément de session.

`SES.MODIFY` Modification d'un élément de session. Contient l'identificateur de l'élément modifié, le type de l'élément mis à la place et les données le concernant.

4.2.2.2 Réponses

`SES.OK` Acquittement positif d'une des requêtes ci-dessus. Contient le numéro de l'identificateur concerné et une chaîne de caractères contenant un commentaire éventuel.

`SES.ERR` Acquittement négatif d'une des requêtes ci-dessus. Contient l'identificateur de l'élément et un commentaire sous forme de chaîne de caractères.

`SES.PROOF` Acquittement positif d'une des requêtes ci-dessus avec création de la racine d'une preuve. Elle a pour arguments l'identificateur de l'élément de session, un commentaire sous forme de chaîne de caractères et un séquent.

4.2.3 Classe `Proof`

La classe `Proof` gère la construction et la destruction des éléments de preuve. La suppression d'un élément de preuve provoque la suppression de tout le sous-arbre ayant cet élément pour racine et la création d'un trou "?" dans la preuve à la place. Il n'y a pas de possibilité de modifier directement un élément de preuve.

4.2.3.1 Requêtes

`PRF.ADD` Ajout d'un élément de preuve, les paramètres sont l'identificateur de l'élément de session, la position dans l'arbre de

preuve et les données nécessaires à la construction (nom de la tactique et éventuellement ses arguments).

PRF.DEL Destruction d'un élément de preuve, contenant l'identificateur de l'élément de session et la position dans l'arbre de preuve.

4.2.3.2 Réponses

Les réponses PRF.NODE, PRF.SUBQED et PRF.ERR correspondent à une requête appliquant une tactique à un but. PRF.OK et PRF.ERR correspondent à la suppression d'un noeud de preuve.

PRF.OK Cette réponse correspond à une commande `Delete`. C'est un acquittement positif qui comporte deux paramètres, l'identificateur au niveau session et la position dans l'arbre de preuve.

PRF.ERR Erreur, contenant l'identificateur de l'élément de session, la position dans l'arbre de preuve et un commentaire sous la forme d'une chaîne de caractères.

PRF.NODE Indique la création de sous-buts en réponse à la création d'un nœud de preuve. Contient en argument l'identificateur de l'élément de session et la liste des positions dans l'arbre de preuve des séquents engendrés ainsi que ces séquents.

PRF.SUBQED Indique que le sous-arbre est prouvé (l'insertion de l'élément de preuve n'a pas créé de nouveau sous-but). Les arguments sont l'identificateur de l'élément de session et la position dans l'arbre de preuve.

La spécification du moteur de preuve nous indique quelles sont les réponses faites par le moteur aux différentes commandes de PEP en fonction de l'environnement courant.

4.3 Spécification du moteur de preuves

Dans ce qui suit, nous donnons une formalisation du noyau du système de preuve. Nous voulons décrire en détail la réaction de celui-ci à chacun des événements envoyés par l'interface. Pour cela nous formalisons l'environnement, les sessions et les preuves de façon à pouvoir observer leurs modifications, et les réponses du serveur en fonction des événements reçus. Nous avons choisi

pour cette description un niveau de granularité ne permettant pas de voir ce qui se passe au niveau des formules ni des tactiques. La raison en est que le niveau le plus fin a déjà été décrit dans le chapitre présentant la formalisation, et que de cette façon, la description qui suit reste indépendante du formalisme choisi. En revanche nous voulons traiter la problématique engendrée par le fait que l'utilisateur peut intervenir en – presque – n'importe quel point de la session, nous nous intéressons donc à la gestion des noms, notamment les problèmes de portée et de dépendances.

Nous commençons par décrire la représentation de l'environnement (état de la machine), puis nous décrivons les fonctions utilisées. Nous finissons par les transitions du système.

On se donne un ensemble d'identificateurs $\mathcal{I}$ pour identifier les éléments de session, et un ensemble d'occurrences dans un arbre $\mathcal{O}$ pour les positions dans les arbres de preuves. On appellera *lieu* un identificateur suivi optionnellement d'une occurrence. On notera un lieu $[i]$, ou $[i, o]$ (avec i un identificateur et o une occurrence), et on notera $\mathcal{L}$ l'ensemble des lieux. On se donne également un ensemble de noms $\mathcal{N}$, un ensemble de types $\mathcal{T}$ et un ensemble de formules Φ.

On note $\mathcal{E}$ l'ensemble des éléments de session, dont chaque élément est un enregistrement.

Les éléments de session sont :

$\mathrm{Declare}(n, \sigma)$ où $n \in \mathcal{N}$ et $\sigma \in \mathcal{T}$;

$\mathrm{Define}(n, \varphi, \sigma)$ où $n \in \mathcal{N}$, $\varphi \in \Phi$, $\sigma \in \mathcal{T}$;

$\mathrm{Axiom}(n, \varphi)$ où $n \in \mathcal{N}$, $\varphi \in \Phi$;

$\mathrm{Theorem}(n, \varphi, \Pi)$ où $n \in \mathcal{N}$, $\varphi \in \Phi$ et Π (la preuve) est décrite en 4.3.2.

Le let est identique au Define, la distinction entre terme et formule n'est pas importante ici. Le type est également similaire au Define si ce n'est qu'il rajoute d'autres noms (principe d'induction, etc.). Nous ne décrivons donc pas leur traitement ici.

On utilise une fonction $def : \mathcal{I} \times \mathcal{N} \times \Phi \to \mathcal{P}(\mathcal{N})$ permettant de vérifier la validité d'une définition : cette fonction prend en argument le lieu, le nom et la formule et retourne une liste de noms dont dépend la validité de la définition.

4.3.1 Environnement

Pour gérer l'environnement, on utilise trois objets : une liste d'association $s : \mathcal{I} \to \mathcal{E}$, représentant la session et associant les éléments de session aux identificateurs, une fonction $dep : \mathcal{N} \to \mathcal{P}(\mathcal{L})$ représentant le graphe de dépendance et associant à un nom l'ensemble des lieux où il est utilisé. Cela nous permettra de gérer les problèmes de portée et de dépendances. Enfin, on utilise la fonction $env : \mathcal{N} \to \mathcal{P}(\mathcal{I})$ qui à un nom associe la liste des éléments de session où ce nom à été défini ou déclaré (ce qui comprend les noms de théorèmes ou d'axiomes). C'est une liste car un nom peut être lié en plusieurs endroits.

On se donne également une relation d'ordre partiel $\prec$ sur les lieux. Au niveau session, cet ordre est défini comme l'ordre sous-jacent à la liste d'association s (si (i, e) est avant (i', e') dans la liste, alors $(i, e) \prec (i', e')$). On le prolonge dans les preuves par l'ordre partiel *est préfixe de* sur les occurrences dans l'arbre de preuve.

Pour manipuler le graphe de dépendance, nous voulons pouvoir obtenir la liste des lieux où un nom est utilisé après un lieu i donné. On utilisera $dep(x)|_i$, défini à partir de dep et de l'ordre dans les éléments. C'est-à-dire $dep(x)|_i = \{j \in dep(x) | j \succ i\}$, l'ensemble des lieux où x est utilisé après le lieu i.

On a également besoin de déterminer quel est le dernier élément de session où est défini un identificateur, de manière à savoir si l'on se trouve dans sa portée. On définit donc la fonction $lookup : \mathcal{N} \times \mathcal{I} \to \mathcal{I}$ telle que $lookup(n, i)$ est le plus grand lieu j (selon $\prec$) tel que $j \in env(x)$ et $j \preceq i$. Le mécanisme de recherche de la dernière définition d'un nom suit la flèche de la figure 4.3.

Par abus de notation, on identifiera i et $[i]$.

On notera i^+ et i^- le successeur et le prédécesseur par $\prec$ de i (lorsqu'ils existent) si i est un identificateur ou un lieu réduit à un identificateur.

4.3.2 Preuves

Les preuves seront contenues dans les éléments $\mathrm{Theorem}$. Chacune des preuves sera représentée par un arbre $\Pi : \mathcal{O} \to \Sigma \times (T \cup \{\bot\})$ où Σ est l'ensemble des séquents, T l'ensemble des tactiques et $\bot$

FIG. 4.3 – Recherche d'un nom

est une constante dénotant un trou dans la preuve. Chaque tactique est une fonction de type $\Sigma \to \mathcal{P}(\Sigma) \times \mathcal{N} \times \mathcal{N}$ prenant un séquent en argument et renvoyant la liste des séquents restant à prouver, la liste de noms utilisés dans la tactique et la liste de noms créés.

Il est à noter que la représentation sous forme d'arbre n'est pas indispensable, on peut aussi bien considérer les occurrences comme des identificateurs et les preuves, donc, comme des tables, à condition cependant de pouvoir définir l'ordre $\prec$ dans les preuves.

Dans un souci d'abstraction et de modularité, nous ne décrivons pas ici les formules, les séquents et les tactiques. Nous avons en revanche besoin d'une fonction $type : \Phi \times \mathcal{L} \to \mathcal{T} \cup \{Err\}$ donnant le type d'une formule dans l'environnement connu en un lieu. Nous avons besoin d'une fonction $FV(\varphi)$ donnant la liste des noms libres dans une formule. Cette fonction est trivialement étendue aux séquents, éléments de preuve, preuves et éléments de session. Une tactique est une fonction prenant en argument un séquent et un lieu et retournant soit un triplet (l, d, n) où l est une liste de séquents et d la liste de noms utilisés par la tactique et n la liste des nouveaux noms introduits par la tactique, soit une erreur.

Après avoir décrit l'environnement de la machine spécifiant le système d'aide à la preuve, nous allons maintenant en décrire les transitions.

Les transitions sont des opérations sur l'état, c'est à dire les fonctions s, env et dep définies plus haut. Chacune de ces transitions est déclenchée par l'arrivée d'un message envoyé par l'interface. Ce message est traité, entraînant la modification de l'environnement. Enfin, une réponse est émise vers l'interface.

Pour décrire ces transitions, nous utilisons un langage informel. Quelques précisions sont cependant nécessaires : **SI** , **SINON** et **POUR TOUT** se passent d'explications ;
INSERER . . . **APRES** i permet d'insérer un élément de session après l'élément portant l'identificateur i dans la session (qui est une liste) ; à l'inverse **SUPPRIMER** a pour effet de supprimer un élément dans la liste (ou un nœud d'un arbre) ; **EMETTRE** indique l'émission d'un message à l'attention de l'interface ; enfin, $\leftarrow$ représente l'opération d'affectation : $env(n) \leftarrow env(n) \cup \{i\}$ indique que la nouvelle valeur de $env(n)$ l'ensemble des éléments de l'ancienne valeur auquel on ajoute i.

Les commentaires sont notés en italique.

Nous commençons par décrire les transitions portant sur les éléments de sessions, puis nous décrivons celles portant sur les preuves.

4.3.3 Niveau session

Ajout d'un élément de session :

Le système reçoit la commande $\texttt{SES.ADD}(i, p, \text{Declare}, n, \sigma)$ où i et p sont des identificateurs, n est un nom et σ est un type.

SI $lookup(n, p)$ est défini et $dep(n)|_p \neq \emptyset$ (*si le nom de l'élément que l'on défini existe plus haut, et est utilisé plus bas, on ne doit pas le masquer*)

EMETTRE $\texttt{SES.ERR}(i, dep(n)|_p)$. (*on indique à l'utilisateur les éléments qui dépendent de celui que l'on masque*)

SINON (*soit le nom n'existe pas, soit il n'est pas utilisé plus bas que le point où l'on insère*)

SI pour tout x dans $FV(\sigma)$, $lookup(x, p)$ est défini (*le type est bien défini*)

INSERER $s(i) = \text{Declare}(n, \sigma)$ **APRES** p dans s ; (*ajout de la déclaration dans la session*)
$env(n) \leftarrow env(n) \cup \{i\}$; (*ajout du nom dans l'environnement*)
POUR TOUT x de $FV(\sigma)$, $dep(x) \leftarrow dep(x) \cup \{i\}$; (*enregistrement des dépendances*)
EMETTRE SES.OK(i).
SINON EMETTRE SES.ERR$(i,$ `"Erreur de type"`$)$. (*Le type donné lors de la déclaration contient des noms inconnus*)

Le système reçoit la commande SES.ADD$(i, p, \text{Define}, n, \varphi)$ où i et p sont des identificateurs, n est un nom et φ est une formule.
SI $lookup(n, p)$ est défini et $dep(n)|_p \neq \emptyset$ (*on essaie de redéfinir un identificateur utilisé*) **EMETTRE** SES.ERR$(i, dep(n)|_p)$.
SINON
soit $\tau = type(\varphi, p)$;
SI $\tau = Err$ **EMETTRE** SES.ERR$(i,$ `"Erreur de typage"`$)$.
SINON
SI $def(i, n, \varphi) = Err$ (*la définition est incorrecte*)
EMETTRE SES.ERR$(i,$ `"Def. invalide"`$)$.
SI $def(i, n, \varphi)$ retourne l'ensemble de nom X (*la définition est correcte*)
INSERER $s(i) = \text{Define}(n, \Phi, \tau)$ **APRES** p (*enregistrement de la définition*) ;
POUR TOUT x dans $FV(\varphi) \cup FV(\tau) \cup X$,
$dep(x) \leftarrow dep(x) \cup \{i\}$ (*enregistrement des dépendances de la définition*) ;
$env(n) \leftarrow env(n) \cup \{i\}$ (*enregistrement du nom*) ;
EMETTRE SES.OK(i, τ).

Le système reçoit la commande SES.ADD$(i, p, \text{Axiom}, n, \varphi)$ où i et p sont des identificateurs, n est un nom et φ est une formule.

SI $lookup(n, p)$ est défini et $dep(n)|_p \neq \emptyset$, **EMETTRE** SES.ERR$(i, dep(n)|_p)$.
SINON
– Soit $\tau = type(\varphi, p)$;
SI $\tau \neq Prop$ **EMETTRE** SES.ERR$(i,$ `"Erreur de typage"`$)$.
SINON

INSERER $s(i) = \text{Axiom}(n, \Phi)$ dans s **APRES** p ;
POUR TOUT x dans $FV(\varphi) \cup FV(\tau)$,
$dep(x) \leftarrow dep(x) \cup \{i\}$,
$env(n) \leftarrow env(n) \cup \{i\}$,
EMETTRE SES.OK(i).

Le système reçoit la commande SES.ADD$(i, p, \text{Theorem}, n, \varphi)$ où i et p sont des identificateurs, n est un nom et φ est une formule.

SI $lookup(n, p)$ est défini et $dep(n)|_p \neq \emptyset$ **EMETTRE** SES.ERR$(i, dep(n)|_p)$.
SINON
soit $\tau = type(\varphi, p)$,
SI $\tau \neq Prop$, SES.ERR$(i,$ "Erreur de typage"$)$.
SINON
INSERER $s(i) = \text{Theorem}(n, \Phi, \Pi)$ dans s **APRES** p (Π *est l'arbre de preuve réduit à la racine contenant* $(\Sigma, \bot)$, *avec* $\Sigma \equiv \vdash \varphi$) ;
POUR TOUT x dans $FV(\varphi) \cup FV(\tau)$,
$dep(x) \leftarrow dep(x) \cup \{i\}$,
$env(n) \leftarrow env(n) \cup \{i\}$,
EMETTRE SES.PROOF(i, Σ).

Suppression d'un élément de session :

Le système reçoit la commande SES.DEL(i, p) où i est un identificateur et p est Declare, Define, Theorem, Axiom.

SI $p =$ Declare, et $s(i) = \text{Declare}(n, \sigma)$:
SI $dep(n)|_i \neq \emptyset$ **EMETTRE** SES.ERR$(i, dep(n)|_i)$. (*On se trouve dans le cas où l'on essaie de supprimer un élément utilisé plus loin*)
SINON
$env(n) \leftarrow env(n) - \{i\}$ (*on supprime le nom*) ;
SUPPRIMER $s(i)$ dans s (*on supprime l'élément de session*)
POUR TOUT $x \in FV(\sigma)$, $dep(x) \leftarrow dep(x) - \{i\}$;
EMETTRE SES.OK(i).
SI p est un Define, un Axiom ou un Theorem, et $s(i)$ est un élément du même type que p, le traitement est identique à celui de la déclaration, mais ce sont les dépendances des

variables libres de φ et de $type(i^-, \varphi)$ qui seront éliminées de dep. Pour le théorème, on traitera en plus $FV(\Pi)$ où Π est la preuve.
SINON EMETTRE SES.ERR(i, "Suppression invalide").

4.3.4 Niveau preuve

Les opérations d'édition au niveau preuve sont assez différentes de celles effectuées au niveau session : alors que les manipulations sur la session on pour but d'enrichir ou d'appauvrir un contexte (linéaire), mais avec la possibilité d'effectuer les modifications en n'importe quel point de ce contexte, les opérations sur les preuves ont pour but de construire un arbre, au début réduit à une racine, en l'étendant au niveau des feuilles. Les opérations de suppression, en revanche, pourront avoir lieu sur n'importe quel noeud de l'arbre, et elles auront pour effet de remplacer le sous-arbre ayant ce noeud pour racine par une feuille.

Ajout d'une étape de preuve :

Le système reçoit la commande PRF.ADD(i, o, t, p) où i est un identificateur, o une occurrence, t une tactique et p les arguments de cette tactique.

SI $s(i) = \text{Theorem}(n, \varphi, \Pi)$, et $\Pi(o) = (\Sigma, \bot)$ est une feuille de l'arbre, avec Σ un séquent (*on vérifie que l'élément de session i contient bien une preuve, et que l'élément de preuve désigné par o existe, mais n'a pas encore été résolu*)
Soit $(l, d, z) = t(\Sigma)$ (*l est la liste de nouveaux séquents, d de noms utilisés par la tactique et z de noms créés*),
SI $l = Err$ (*l'application de la tactique provoque une erreur*) **EMETTRE** PRF.ERR(i, o, "Tactique Erronée").
SI $l = [\,]$ (*il n'y a pas de sous-buts, n doit être vide*)
POUR TOUT $x \in d$, $dep(x) \leftarrow dep(x) \cup \{[i, o]\}$ (*on enregistre les dépendances*)
$\Pi(o) \leftarrow \Pi(o) = (\Sigma, t)$ (*on enregistre la tactique utilisée*) ;
EMETTRE PRF.SUBQED(i, o).
SI $l = [\Sigma_1, \ldots, \Sigma_n]$ (*plusieurs sous-buts*)
POUR TOUT $x \in d$, $dep(x) \leftarrow dep(x) \cup \{[i, o]\}$
(*dépendances de ce qui est utilisé dans la tactique*) ;

POUR TOUT Σ_i, **POUR TOUT** $y \in FV(\Sigma_k)$,
$dep(y) \leftarrow dep(y) \cup \{[i, o.k]\}$ (*dépendances des séquents des sous-buts*) ;
POUR TOUT $x \in z$, $env(x) \leftarrow env(x) \cup \{[i, o]\}$
(*enregistrements des noms créés*)
$\Pi(o) \leftarrow \Pi(o) = (\Sigma, t)$ (*tactique utilisée*) ;
$\Pi(o.k)$ ($1 \leq k \leq n$) $\leftarrow \Pi(o.k) = (\Sigma_k, \bot)$ (*création des sous-buts*) ;
EMETTRE PRF.NODE$(i, [o.1, \Sigma_1; \cdots; o.n, \Sigma_n])$.

SINON EMETTRE PRF.ERR$(i, o,$ "Erreur"$)$ (*il n'y a pas de preuve où de nœud de preuve*).

Si les paramètres de PRF.ADD sont incorrects
EMETTRE PRF.ERR$(i, o,$ "Arguments Incorrects"$)$.

Suppression d'une étape de preuve :

Le système reçoit la commande PRF.DEL(i, o) où i est un identificateur et o une occurrence. Comme indiqué précédemment, cette opération ne supprime pas seulement l'étape de preuve, mais également le sous-arbre de preuve issu de cette étape.

SI $s(i) \neq \text{Theorem}(n, \varphi, \Pi)$
EMETTRE
PRF.ERR$(i, o,$ "Cet élément n'est pas une preuve"$)$.
SINON ($s(i) = \text{Theorem}(n, \varphi, \Pi)$ *où* Π *est un arbre de preuve et* n *un nom*)
SI $\Pi(o) = (\Sigma, \bot)$
EMETTRE PRF.ERR$(i, o,$ "Rien à supprimer"$)$ (*aucune tactique appliquée ici*).
SINON (*si* $\Pi(o) = (\Sigma, t)$).
POUR TOUTE occurrence o' strictement sous o (*on supprime chaque nœud sous* o *complètement*)
POUR TOUT x tel que $[i, o'] \in env(x)$,
$env(x) \leftarrow env(x) - \{[i, o']\}$.
POUR TOUT x tel que $[i, o'] \in dep(x)$,
$dep(x) \leftarrow dep(x) - \{[i, o']\}$.
SUPPRIMER $\Pi(o')$.
$env(x) \leftarrow env(x) - z$ et $dep(x) \leftarrow dep(x) - d$, avec d et z tels que $t(\Sigma) = (l, d, z)$;
$\Pi(o) \leftarrow (\Sigma, \bot)$; (*on supprime la tactique appliquée en* o)
EMETTRE PRF.OK(i, o).

Il nous reste à passer de l'autre côté de la barrière pour voir ce qui se passe au niveau de l'interface.

4.4 Spécification de l'interface

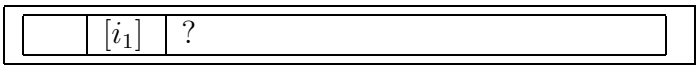
Le contenu de l'interface n'est pas identique au contenu du moteur de preuve. L'utilisateur peut entrer des commandes sans les valider, créer de nouveaux éléments de session, et tout cela ne sera pas répercuté sur le serveur. Seuls les éléments dits *validés* ont une existence pour celui-ci. Cela explique que toutes les modifications dans l'interface ne sont pas suivies d'une émission de message à destination du moteur de preuve.

L'interface est définie comme une suite de boîtes correspondant à des éléments de session dans une boîte correspondant à l'interface. Chacune des boîtes des éléments de session est pourvue d'un identificateur.

Les actions de l'utilisateurs possibles sur les boîtes sont la création d'une boîte vide avant ou après la boîte courante, l'insertion d'un texte dans une boîte (non validée), la validation, l'invalidation et la suppression d'une boîte non valide, ainsi que le changement de focus[2].

Les crochets autour de l'identificateur indiquent que le point d'insertion (curseur) est dans cette boîte. L'identificateur en gras signifie que la boîte est verrouillée (on ne peut modifier son contenu, mais il n'a pas été validé par le serveur). Si le symbole **V** apparaît, elles sont verrouillées en attente de validation, si c'est un **I**, en attente d'invalidation. Les boîtes contenant ✓ sont validées (on ne peut pas modifier le contenu non plus).

L'environnement initial est



La boîte externe représente l'interface au complet. La boîte interne représente un élément de session vide (le point d'interrogation est une *invite*. i_1 est l'identificateur de l'élément de session.

[2]Avoir le focus signifie être l'élément recevant les actions de l'utilisateur. Ce n'est pas forcément l'élément actif dans l'opération courante.

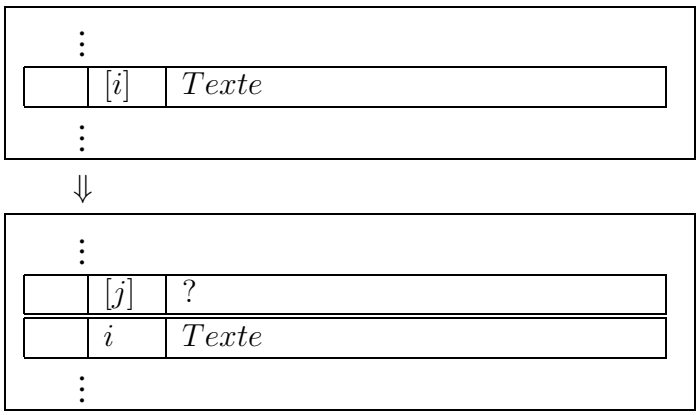
Les schémas donnés ne sont pas une représentation graphique de l'interface, mais juste une notation pour décrire ce qui se passe.

L'interface réagit à deux sources d'événements : ceux en provenance de l'utilisateur et ceux en provenance du serveur. Nous reflétons ceci en traitant d'abord les premiers, puis les seconds.

4.4.1 Niveau session, événements utilisateur

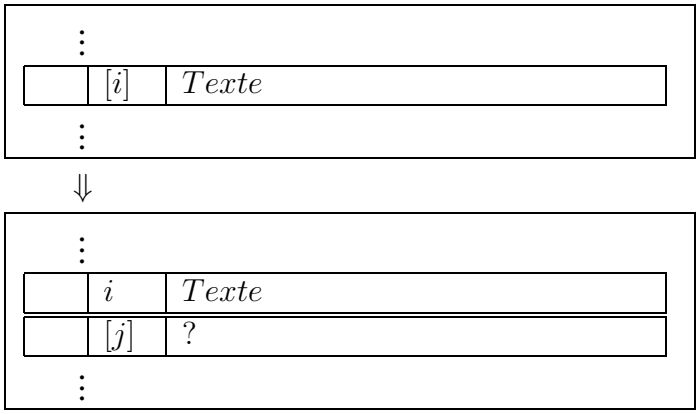
Création d'un élément avant la boîte courante

On crée une nouvelle boîte vide, ayant pour (nouvel) identificateur j. Cette nouvelle boîte ne contient qu'une invite et obtient le focus.



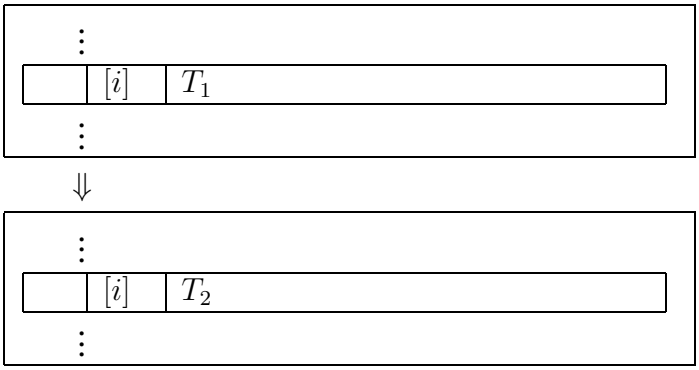
. . .après la boîte courante

Cette commande est identique à la précédente, mais la nouvelle boîte est créée après la boîte courante.



Insertion d'un texte

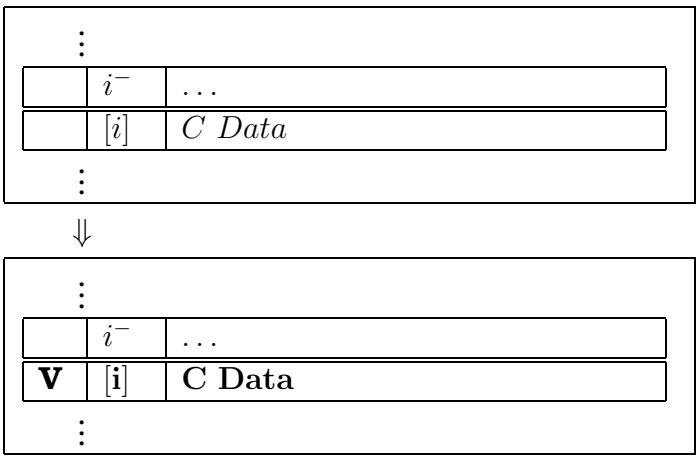
Cette transition représente l'insertion d'un texte à la place d'un autre dans un élément qui n'est ni validé ni verrouillé. T_1 et T_2 sont des textes quelconques, mais T_1 peut être l'invite "?". On peut noter que rien n'est envoyé à ce stade au serveur.



Validation d'un élément de session

L'utilisateur demande la validation d'un élément de session. L'interface envoie au serveur le message SES.ADD$(i, i^-, C, Data)$. En attendant une réponse positive ou négative, l'élément de session est bloqué (l'utilisateur ne peut le modifier ni le supprimer). C est Declare, Define, Theorem ou Axiom. $Data$ contient les données associées. Il faut envoyer l'identificateur de l'élément précédent pour que le serveur sache où insérer cet élément.

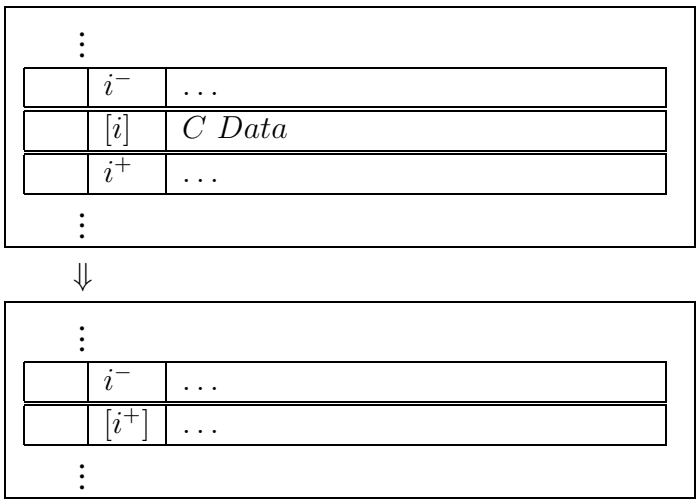
S'il n'y a pas de prédécesseur, on prend l'identificateur réservé 0 à la place de i^-.



Suppression d'un élément de session non validé

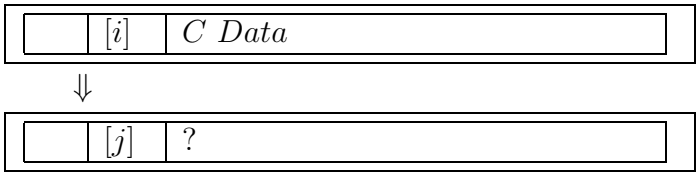
Lorsqu'un élément de session n'est ni validé, ni verrouillé (le moteur de preuve ne connaît donc pas son existence), il est possible de le supprimer. Aucun message n'est envoyé.

Dans le cas où il n'y a pas de prédécesseur ou pas de successeur, le comportement est identique (s'il n'y a pas de successeur, l'élément courant conserve le focus). S'il n'y a ni successeur ni prédécesseur, voir la règle suivante.



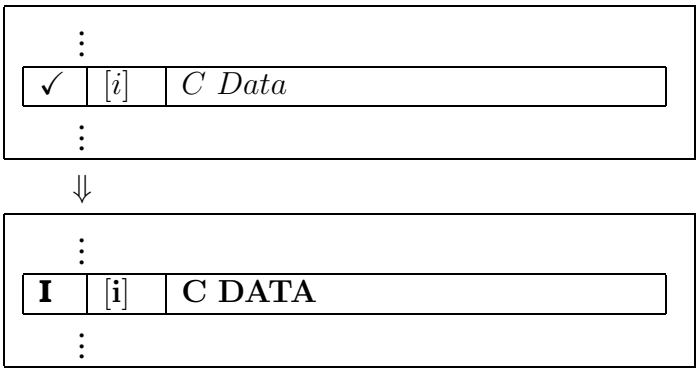
Suppression d'un élément de session seul

La suppression d'un élément de session sans successeur ni prédécesseur crée un nouvel élément vide à la place, et lui donne le focus.



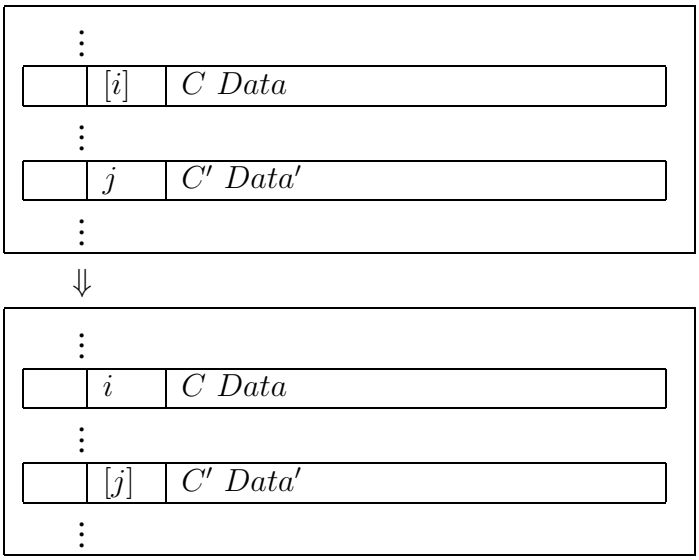
Invalidation d'un élément de session

L'invalidation d'un élément de session n'est possible que si celui-ci est validé. Cela provoque la suppression de l'élément côté moteur de preuve. On émet donc `SES.DEL`(i). l'élément est verrouillé en attendant la réponse du serveur.



Changement de focus

L'utilisateur fait passer le focus d'un élément à un autre.



Si une des boîtes est une boîte de preuve, le fonctionnement est identique.

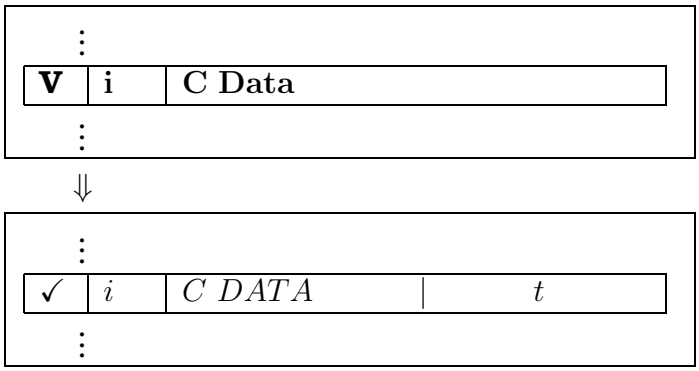
Passons maintenant aux réactions aux commandes renvoyées par le serveur.

4.4.2 Niveau session, événements moteur

Deux cas se présentent lors de l'acquittement positif d'un élément de session : le moteur accepte l'élément tel quel, dans ce cas il émet SES.OK(i, t), ou l'élément nécessite une preuve (théorème), et le moteur demande donc la création d'une boîte de preuve en émettant SES.PROOF(i, t, Σ).

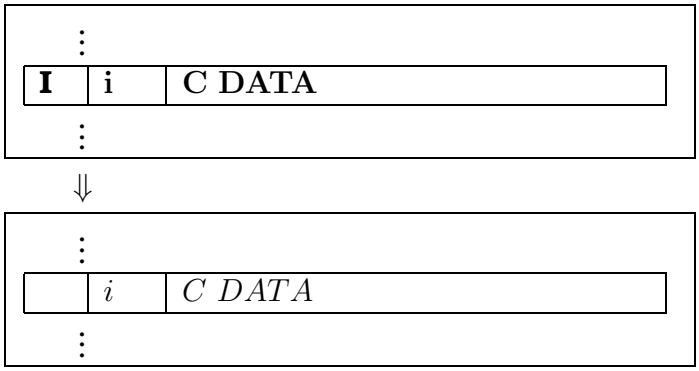
Réception de SES.OK(i, t) **en attente de validation**

Le serveur indique donc que l'élément est accepté. Il passe donc de l'état verrouillé à l'état validé. Si le serveur renvoie un texte d'accompagnement, celui-ci est affiché. Il n'est pas nécessaire que l'élément i détienne le focus pour que cet événement puisse se produire. Le focus n'est pas modifié lors de cette opération.



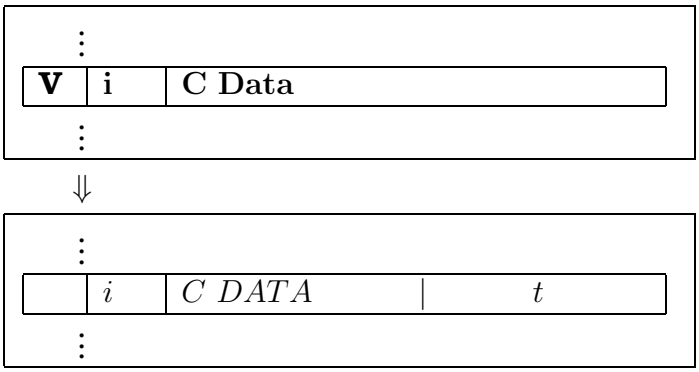
Réception de SES.OK(i) **en attente d'invalidation**

Cas où l'élément ne contient pas de preuve. Le serveur accepte l'invalidation. Du côté de l'interface, l'élément est simplement déverrouillé, mais du côté serveur, il est supprimé. Une fois de plus, le focus n'intervient pas ici.



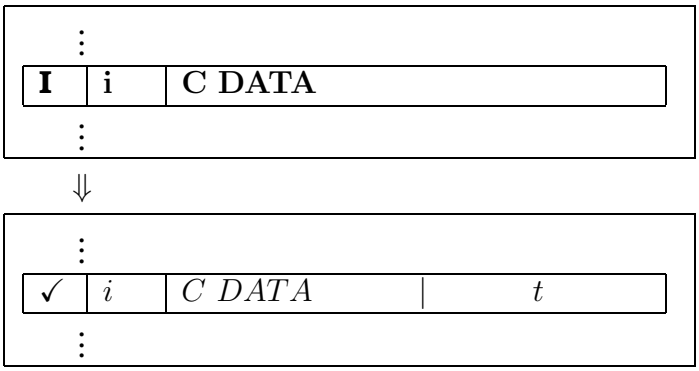
Réception de SES.ERR(i, t) **en attente de validation**

Le moteur de preuve refuse de valider cet élément. Celui-ci est déverrouillé et un message d'erreur est affiché.



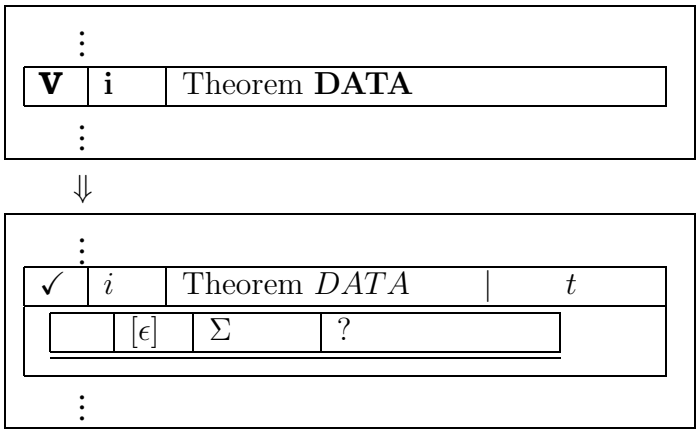
Réception de SES.ERR(i, t) **en attente d'invalidation**

Cas où l'élément ne contient pas de preuve. Le serveur refuse de supprimer cet élément. Le cas le plus courant est celui où d'autres éléments en dépendent, auquel cas le message t contient la liste des éléments rendant impossible l'invalidation.



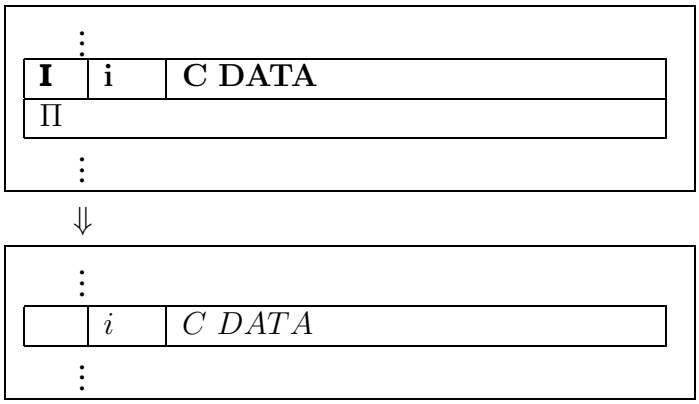
Réception de SES.PROOF(i, t, Σ)

Le moteur de preuve commande la création de la racine d'une preuve dans cet élément de session. Il envoie un séquent (des données arbitraires) à prouver. L'interface transforme la boîte pour pouvoir y installer la preuve. Une invite est mise au lieu où l'on doit entrer une tactique de preuve, et le focus est mis à cet endroit. L'occurrence de la racine de la preuve est notée ϵ.



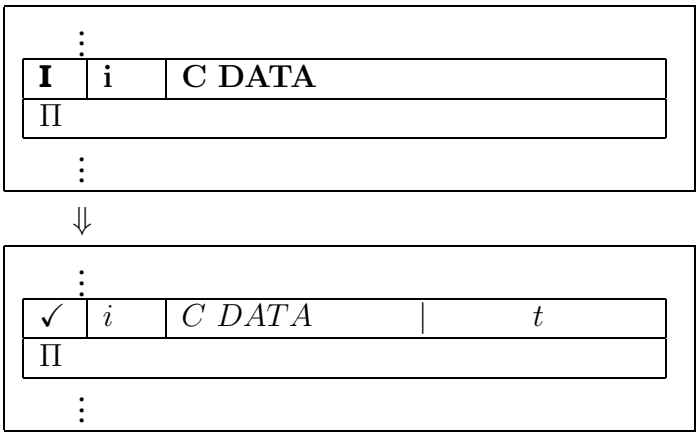
Réception de SES.OK(i) **en attente d'invalidation**

Cas où l'élément contient une preuve (Theorem). On est de nouveau dans un cas où le moteur autorise l'invalidation d'un élément, mais celui-ci contient une preuve Π. Tout se passe comme dans une suppression habituelle, mais la preuve est supprimée et on retransforme la boîte de l'élément de session en boîte standard (sans preuve).



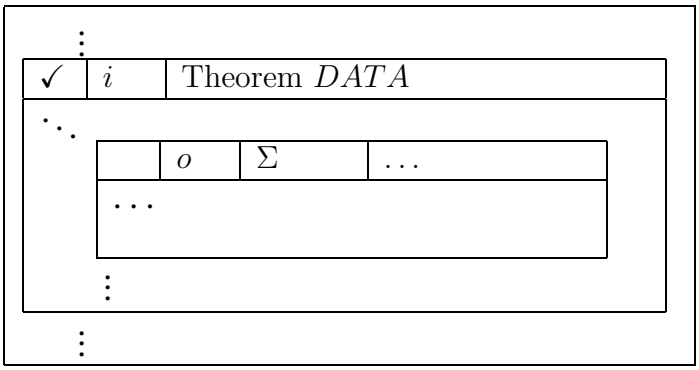
Réception de SES.ERR(i, t) **en attente d'invalidation**

Dans le cas où l'élément est un Theorem. Le moteur a refusé la suppression (un élément dépend de celui-ci). On restaure donc l'indicateur ✓ de l'élément et on affiche le message d'erreur.

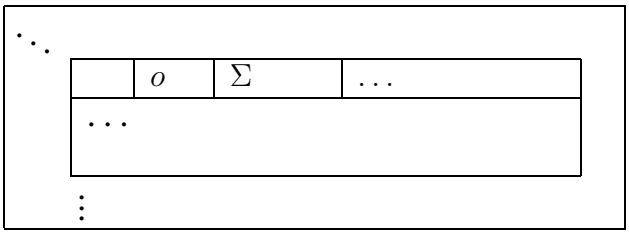


4.4.3 Niveau preuve, événements utilisateurs

Une action en mode preuve ne peut pas avoir d'influence en amont du nœud qu'elle adresse. Nous allons donc représenter l'interface de la manière suivante :

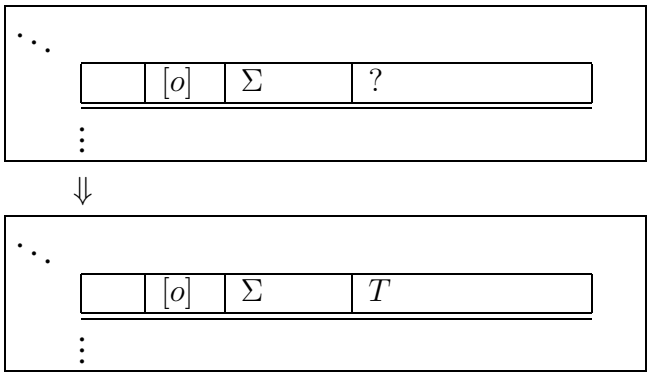


Le schéma représente l'interface contenant un élément de session (valide) i, contenant lui même une preuve où à l'occurrence o, le séquent à prouver est Σ. Pour simplifier, on représentera cela par :



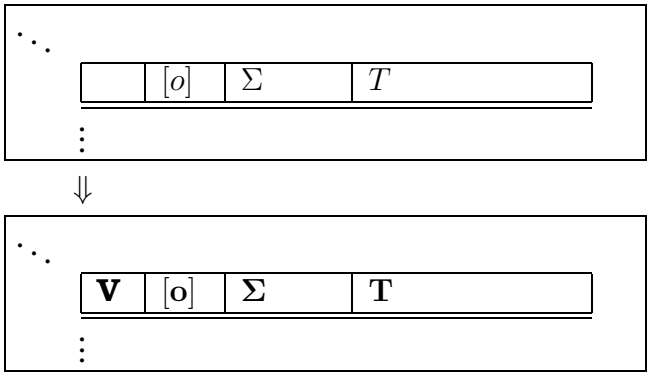
Insertion d'un texte dans un élément de preuve

L'utilisateur tape le texte T (une tactique et ses arguments) à la position o de l'arbre de preuve de l'élément de session i. Rien n'est envoyé au moteur de preuve.



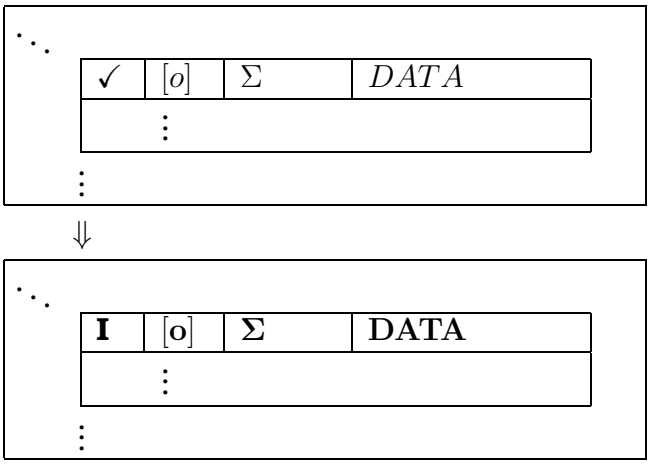
Validation d'un élément de preuve

Demande de validation d'un nœud de la preuve. Celui-ci est verrouillé en attendant la réponse. On envoie au moteur de preuve PRF.ADD(i, o, T).



Invalidation d'un élément de preuve

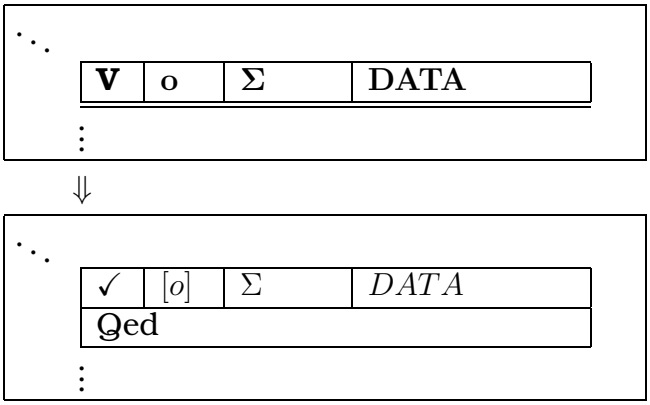
Demande par l'utilisateur d'invalidation d'une commande sur un nœud de preuve. On envoie au serveur PRF.DEL(i, o).



4.4.4 Niveau preuve, événements moteur

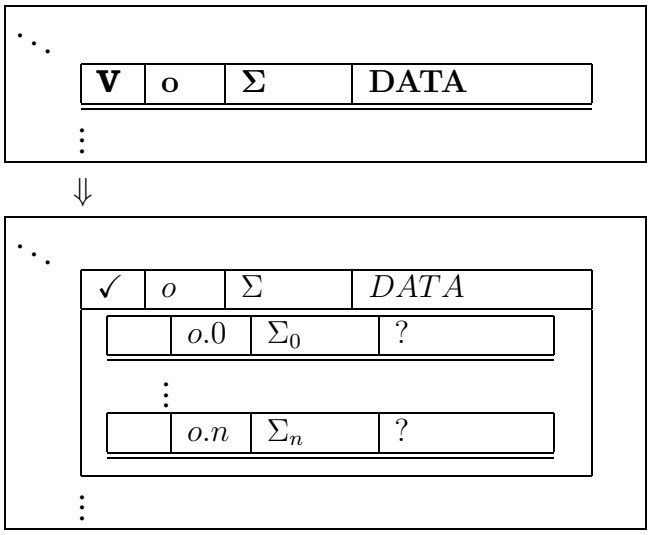
Réception de PRF.SUBQED(i, o)

Réponse du noyau, pas de sous-buts lors d'une validation. Le sous-arbre de preuve est prouvé. On valide l'élément et on ferme cette branche de la preuve.



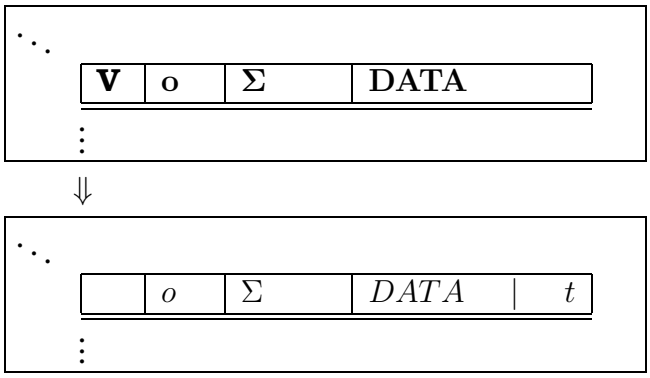
Réception de PRF.NODE$(i, o.0, \Sigma_0, \ldots, o.n, \Sigma_n)$

Réponse du noyau, création de sous-buts. L'interface crée des boîtes pour accueillir les sous-buts engendrés, qui sont munis d'une invite. Les $o.0, \ldots, o.n$ sont les occurrences des fils.



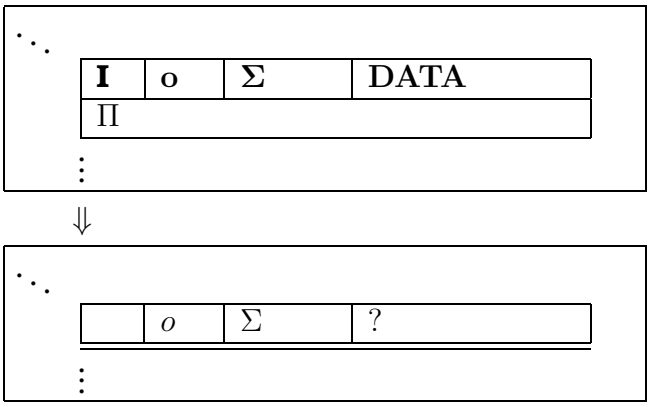
Réception de PRF.ERR(i, o, t)

Réponse négative lors de la validation. La tactique n'est donc pas la bonne. On affiche le message d'erreur t et on déverrouille la boîte.



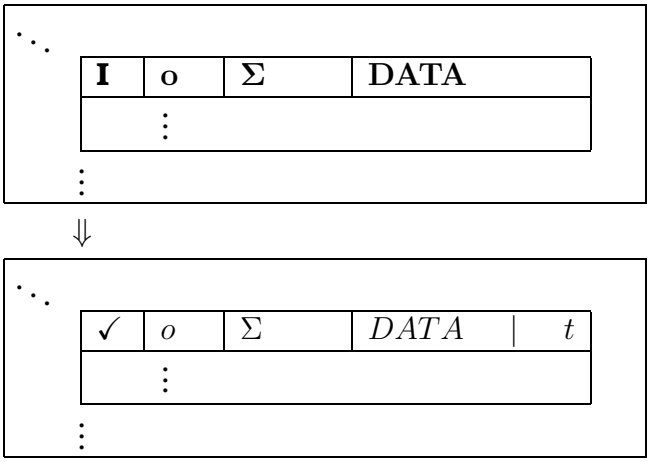
Réception de PRF.OK(i, o)

Réponse positive lors de l'invalidation. L'interface déverrouille la boîte et supprime la preuve Π qui en partait.



Réception de PRF.ERR(i, o, t)

Réponse négative lors de l'invalidation, il suffit de restaurer l'indicateur ✓. On ne touche pas à la preuve. Ce cas n'est jamais supposé se produire, sauf dans le cas d'une inconsistance entre le moteur et l'interface.



De même que pour les éléments de session, on ajoute la possibilité de se déplacer dans la preuve et de quitter celle-ci (changer le focus vers un autre élément de session).

Chapitre 5

Implémentation du moteur de preuve

Nous présentons ici l'implémentation du système PAF! destiné à la spécification et à la vérification de programmes ML. Cependant, les choix faits et décrits ci-dessous présentent un intérêt indépendamment de la logique que l'on souhaite manipuler dans l'AP.

Au cours de la réalisation de ce système, les traits principaux (architecture du serveur, représentation des preuves et des sessions, contextes chaînés) de cette implémentation ont également été expérimentés dans un AP pour une logique d'ordre supérieure à sortes simples multiples.

Dans la suite, nous décrivons le système PAF! en suivant un découpage par fonctionnalité (figure 5.1). Nous avons choisi de donner pour chacune de ces parties du programme aussi bien une description des aspects intéressants de l'implémentation de celle-ci, mais aussi de donner la liste des modules, pour référence et pour aider à la compréhension de la structure du système.

5.1 Généralités

L'architecture générale correspond à la figure page 133. Nous avons d'un côté l'interface et de l'autre le moteur de preuve. Nous ne nous attacherons ici qu'à l'implémentation du moteur. Les tactiques sont représentées séparément du système car il est néces-

Chaque boîte représente un groupe de modules.
Les flèches indiquent la dépendance.

FIG. 5.1 – Découpage par fonctionnalités

saire de pouvoir les isoler du reste du moteur. En effet, alors qu'il est probable que la plus grosse partie du système reste stable au cours des évolutions futures, il doit être aisé de pouvoir rajouter de nouvelles tactiques adaptées à l'usage que l'on a de PAF!. Nous revenons sur ce point en 5.6.2.

Nous combinons dans ce logiciel l'utilisation des types sommes pour tout ce qui est langage de termes et formules, avec l'utilisation d'objets pour représenter les éléments de session et de preuve. Le choix de l'objet pour les éléments de session est essentiellement dû au fait que la session et les éléments de session forment l'état du système et la programmation objet fournit une manière élégante de les implémenter. Les méthodes de ces objets sont essentiellement des méthodes d'accès aux données, notamment au contexte en amont et en aval de l'élément.

La complexité de la représentation du contexte, de la session et

FIG. 5.2 – Architecture globale

des preuves est une conséquence directe de la souplesse à l'utilisation : la possibilité qu'a l'utilisateur de pouvoir agir en n'importe quel point. En effet, la suite des évènements doit être conservée (et pas seulement leurs effets) et l'utilisateur peut intervenir pour détruire ou ajouter n'importe où. Cela oblige à avoir une "dynamicité" plus grande des structures de données que dans les systèmes habituels où ce type de problématique n'apparaît que pour la gestion de l'historique (*undo*).

Pour les aspects preuve, l'approche objet s'avère d'une aide précieuse compte tenu de la manière dont on veut les manipuler. Nous décrivons ceci plus précisément en 5.6.2.

Nous ne décrivons pas ici la couche objet d'*Objective Caml*, on donc peut se référer à [59].

5.2 Utilitaires

La première série de modules contient des structures de données, des fonctions utiles indépendantes de l'application développée, ainsi que des fonctions servant essentiellement au debuggage de l'application.

`XList` : Fonctions supplémentaires sur les listes.

`XString` : *idem* pour les chaînes de caractères.

`Coll` : Dictionnaire où l'ordre des entrées est significatif.

`IndexedTree` : Arbres n-aires génériques.

`Devel` : Fonctions et exceptions utilisées pour le debuggage

`Printers` : Pour le debuggage, définit l'affichage des données dans le *toplevel Objective Caml*.

5.3 Identificateurs, noms et contextes

Viennent ensuite les identificateurs, noms et opérations sur les contextes (cette partie est indépendante du système logique).

La gestion des variables est un problème délicat. Est-ce qu'une variable ne dénote qu'une place (ou un ensemble de places) dans un terme ou une formule, est-ce que son nom est significatif ?

Nous distinguons deux représentations pour les variables, nom de types et constantes. Chaque variable possède un nom (`name`) et un identificateur (`ident`). Le nom est la représentation pour l'utilisateur de la variable. Il est représenté comme une chaîne de caractères. Plusieurs variables peuvent porter le même nom. La portée lexicale et le masquage s'appliquent comme vu à la figure 4.3. En revanche, chaque variable porte un identificateur, qui est attribué par le système et qui est unique. Cet identificateur (un entier), n'est jamais présenté à l'utilisateur. Une variable change d'identificateur quand elle passe de l'état lié à l'état libre (par exemple lors de l'introduction d'un quantificateur). Elle peut également changer de nom lorsque cela risque de donner lieu à des confusions. L'utilisation d'un identificateur permet de s'affranchir d'un certain nombre de renommages.

Pour interpréter les entrées de l'utilisateur ou pour l'affichage, il est nécessaire de trouver la correspondance entre le nom et l'identificateur. Dans le sens identificateur vers nom, la table est globale (les identificateurs sont uniques). Dans l'autre sens, elle est implémentée de façon chaînée, car elle dépend du lieu où l'on se trouve (portée lexicale), plusieurs variables différentes pouvant porter le même nom. La chaîne de contexte suit le mécanisme de recherche de nom décrit à la figure 4.3. Le mécanisme de création des nouveaux nom fait également appel aux contextes chaînés, car cette création ne doit pas masquer de noms existant.

Chacune des cellules de cette chaîne de contexte sera portée par

un élément de preuve ou de session et contiendra le lien nom vers identificateur pour les noms définis dans cet élément. Le contexte se présente donc comme remontant un arbre à partir des feuilles (les feuilles des preuves et le dernier élément de session) jusqu'à la racine (l'élément de session initial). Pour implémenter ceci nous définissons dans `Names` une signature `nameContext` puis nous définissons deux classes, une correspondant au contexte initial et une correspondant à une cellule pointant vers son père.

`BasicTypes` : Type de bases (abstraits), notamment les identificateurs et les noms.

`Names` : Définition des contextes et table de correspondance des identificateurs vers les noms.

5.4 Types, termes et formules

On peut maintenant définir les langages de types, de termes et de formules, les opérations pour les manipuler ainsi que les tables permettant de les stocker. Les types, termes et formules sont implémentés par des types sommes. Les variables sont représentés par leur identificateur. Les fonctions définies sur ces types prennent en charge le renommage des variables, lorsque celui-ci est nécessaire.

Les séquents sont implémentés comme une paire de collections (`Coll`) de formules nommées. Chaque séquent contient au plus une hypothèse et une conclusion dites *actives*. Bien que les séquents soient multi-conclusions, cela n'apparaitra pas à l'utilisateur car seule la conclusion active est affichée. Le fait d'avoir plusieurs conclusion n'est utile que pour le raffinement (voir ci-dessous), car il utilise les règles de la déduction libre qui nécessite des séquents multi-conclusions.

Les langage des types, termes et formules dépendent du formalisme logique choisi (dédié à la preuve de programmes ML, dans notre cas).

`TypeML` : Langage des types ML (et schémas de types) ainsi que la substitution, l'égalité, la généralisation, la conversion sous forme de chaîne de caractères, etc. Les types étant exportés

comme types de données abstraits, on en donne les constructeurs, reconnaisseurs et destructeurs.

`TermML` : Langage des termes ML, sur lesquels on définit les mêmes opérations que ci-dessus.

`Formula` : *idem* pour le langage de formules (langage de spécification).

`Eval` : Fonctions d'évaluation sur les termes et formules.

`Filtre` : Fonctions de filtrage pour les termes et les formules.

`Sequent` : Définitions des séquents et opérations de manipulations (utilise les collections de `Coll`).

`TypeDef` : Définitions de types algébriques à la ML, ainsi que les fonctions permettant de les utiliser pour engendrer les principes de récurrence.

`Typing` : Inférence et vérification de type pour les termes et les formules.

`Tables` : Toutes les tables susceptibles d'accueillir théorèmes, définitions, etc. Les objets sont référencés par leurs identificateurs (`ident`). On utilise également une table des fonctions prouvées totales. Le module `Tables` est rangé dans cette catégorie car il dépend de `TermeML`, `TypeML`, `Formula` et `TypeDef` (les tables contiennent des éléments dont les types sont définis dans ces modules), mais le typage (dans `Typing`) dépend de ces tables.

5.5 Gestion des entrées

Les modules qui suivent servent à interpréter les entrées fournies par l'utilisateur. Ces entrées correspondent soit aux arguments des éléments de session, soit aux arguments des tactiques. Ces entrées sont fournies sous la forme de chaîne de caractère par la couche communication du programme, puis sont analysées. Soit les valeurs obtenues par l'analyse syntaxique et le traitement sont passées en argument aux fonctions créant les éléments de session, soit elles sont mises sous la forme de `psParam`, un objet encapsulant les paramètres susceptibles d'être reçus par une tactique. La tactique recevant le `psParam` a à charge de vérifier que ce qu'il

contient correspond à ce qui est attendu. Ceci est rendu nécessaire par le fait que toutes les tactiques doivent avoir le même type.

Pour ne pas avoir à mélanger la partie analyse syntaxique et gestion des noms, l'analyseur ne produit pas directement des termes et formules, mais des valeurs de type `preFormula` qui mélangent termes et formules, et dont les constantes et variables (indifférenciées à ce stade) sont identifiées par leur nom.

Ici encore, les modules dépendent du formalisme choisi pour le système.

`PreFormula` : Les préformules sont les valeurs construites au cours de l'analyse syntaxique. Ce module contient la définition du type `preFormula` et des fonctions utilies permettant de manipuler les préformules.

`MakeTerm` : Contient les fonctions permettant de transformer une préformule en un terme ou une formule. Ces fonctions prennent en argument le contexte dans lequel on se trouve, pour pouvoir convertir les noms en identificateurs.

`Parser` : Analyseur syntaxique engendré à partir d'`ocamlyacc` et produisant des `preFormula`, des types et des `psParam`.

`Lexer` : Analyseur lexical engendré à partir d'`ocamllex`.

`ParserAPI` : Fonctions utilitaires permettant de gérer tout le processus d'analyse lexicale, puis syntaxique et la transformation vers l'objet final attendu.

`PsParam` : Définit la classe `psParam` qui encapsule tous les types d'arguments qu'est susceptible de recevoir une tactique (listes de formules, termes, noms, types ou entiers).

5.6 Preuves

La gestion des preuves est rendue plus complexe que dans les systèmes habituels pour plusieurs raisons : l'utilisateur doit pouvoir agir en n'importe quel point d'une preuve ; une preuve composée de tactiques de haut niveau doit pouvoir être transformée en une preuve composée uniquement de règles élémentaires ; la création de nouvelles tactiques doit être raisonnablement simple et ne doit pas mettre en danger la fiabilité du système.

5.6.1 Raffinement

Le principe du raffinement est qu'à partir de toute preuve, il doit être possible de générer une nouvelle preuve constituée uniquement d'étapes élémentaires. De plus, le terme constitué doit être facile à vérifier (valider). Nous ne souhaitons pas ici que le terme de preuve obtenu contienne également les preuves des lemmes utilisés. Il nous suffit de connaître son nom et de savoir qu'il a été lui-même validé auparavant. Notre jeu de règles primitives contient la déduction libre, des règles structurelles, une règle permettant d'utiliser un théorème ou un axiome, une règle permettant d'ouvrir une définition et une règle permettant d'évaluer un terme (voir chapitre 3).

La déduction libre a pour avantage qu'il est facile d'y coder les règles du calcul des séquents et de la déduction naturelle, et que les preuves dans la déduction libre sont représentables naturellement sous forme d'algèbre de termes (toutes les règles sont des règles d'élimination et ont le même aspect).

Toute tactique non primitive devra donc comporter deux parties : le code permettant de déterminer si la tactique s'applique, et celui permettant de se rapporter à une preuve plus simple. Il n'est pas indispensable de se ramener en un coup à une preuve atomique, il est possible de passer par d'autres tactiques de plus en plus simples.

La distinction entre la génération des sous-buts et la fabrication du terme de preuve offre la possibilité de différer la vérification des preuves. Ceci peut avoir un intérêt lorsqu'une tactique permet d'obtenir une réponse rapidement, mais que la preuve complète est très longue à engendrer. Par exemple on peut vouloir utiliser des BDD pour une procédure de décision efficace dans le calcul propositionnel. L'utilisateur a donc une réponse rapide lors de la construction de la preuve, mais la procédure de vérification de celle-ci peut prendre plus de temps.

La figure 5.3 illustre le mécanisme de raffinement d'une règle `Intros` ayant effectué toutes les règles d'introductions possibles en une suite de trois règles effectuant l'introduction de l'implication et d'un quantificateur. Ces règles sont raccordées au reste de la preuve. Bien entendu, ce raffinement peut également produire un arbre (avec des trous au niveau des feuilles), mais les trous doivent

être identiques aux prémisses de la tactique initiale.

FIG. 5.3 – Raffinement

Le raffinement correspond au niveau de l'implémentation à la méthode `refine` des objets représentant les nœuds de preuve.

Après avoir raffiné une preuve en étapes primitives, on en extrait le *terme de preuve* proprement dit, au moyen de la méthode `realize`. La seule différence entre la preuve primitive issue du raffinement et ce terme de preuve réside dans la représentation : les preuves construites par l'utilisateur et les preuves obtenues par raffinement ont une représentation complexe (décrite dans la section suivante), tandis que le terme de preuve doit avoir une représentation simple, facile à manipuler au cours de la validation de la preuve : un type algébrique sur lequel nous pouvons travailler par filtrage. Chaque alternative du type algébrique correspondant exactement à une règle primitive.

Cette approche visant à définir une tactique comme ayant deux objectifs : engendrer les sous-buts et un "certificat" garantissant la correction est à rapprocher des tactiques de LCF [28] dans lequel

cette vision existait sous une forme un peu différente où le “certificat” retourné par la tactique est une fonction ML vérifiant la partie de la preuve correspondante.

5.6.2 Représentation

Une des particularités de notre système, mais qui n'est pas visible par l'utilisateur, réside dans la représentation des preuves. Nous voulons que cette représentation “colle” aux manipulations que l'on effectue dessus, c'est-à-dire que l'on puisse émettre vers chacun des nœuds d'une preuve un événement (suppression, application d'une tactique, etc.) correspondant à une requête du protocole vu au chapitre précédent. Nous voulons également qu'il soit aisé pour l'utilisateur de créer une nouvelle tactique, mais nous ne voulons pas que la sûreté du système soit mise en péril par celle-ci.

5.6.2.1 `ProofTree`

L'architecture retenue pour la représentation de la preuve est la suivante : on utilise une structure d'arbre n-aire `proofTree` dont les nœuds sont décorés avec des éléments de preuve. Les méthodes principales de `proofTree` sont `insert`, qui prend en argument la fonction de création d'une étape de preuve, une occurrence et les paramètres de la tactique et instancie la tactique en lui passant en argument le contexte et le but à prouver à l'occurrence donnée (ce qui, en cas de succès, a pour effet d'engendrer les sous-buts), accroche la tactique au nœud correspondant, crée des nœuds fils et y accroche les sous-buts engendrés ; on a également `deleteNode`, qui a pour effet de supprimer le nœud à l'occurrence donnée, ainsi que tout le sous-arbre de preuve partant de ce nœud, puis on place à l'occurrence indiquée un but à prouver avec le même séquent conclusion que celui de la tactique, ainsi qu'un contexte vide pointant sur le contexte du père.

On trouvera également dans `ProofTree` des méthodes permettant le raffinement et la transformation en terme de preuve, mais elles n'ont pour effet que d'appeler les méthodes équivalentes sur le nœud à la racine de la preuve, nous y reviendrons plus loin. On utilise également la méthode `delete` qui supprime complètement l'arbre de preuve.

5.6.2.2 `ProofStep`

Tous les nœuds de preuve implémentent la signature `stepSig` et sont des descendants de la classe virtuelle `proofStep`. L'arbre d'héritage est représenté à la figure 5.4 (`stepSig` étant seulement une signature, elle n'y apparaît pas). Comme on le voit, la classe

Les flèches représentent la relation d'héritage.
Les classes et méthodes abstraites sont en italiques.

Les ajouts de tactiques se font sous la classe `ProofTac`.

FIG. 5.4 – Arbre d'héritage des étapes de preuve (UML).

`proofStep` est le type de tous les objets susceptibles d'apparaître comme nœud dans une preuve. On distingue en dessous les trois principales familles d'étapes de preuves.

On trouve également dans `ProofStep` les méthodes permettant d'accéder aux données (séquents conclusion et prémisses), des itérateurs et une méthode `id` permettant de connaître l'identificateur du nœud (son occurrence).

5.6.2.3 `ProofGoal`

Tout d'abord, on trouve les `proofGoal` permettant de marquer les trous dans une preuve et contiennent le but à prouver, ainsi que le contexte (les noms connus, les types) en ce point. Les deux seules opérations légales sur un `proofGoal` sont le remplacement par un autre type de nœud de preuve, et la suppression (qui n'a jamais lieu directement, mais uniquement lorsque l'on a supprimé un nœud situé en amont dans la preuve).

5.6.2.4 `ProofRule`

La deuxième famille d'étapes de preuve est constituée par les héritiers de la classe virtuelle `proofRule` (les *règles de preuve*). Ces règles correspondent à des étapes de preuve primitives, c'est-à-dire principalement les règles de la déduction libre ainsi que les règles structurelles (activer une hypothèse, etc.). Leur particularité est que chacune de ces règles correspond directement à un terme de preuve (c'est-à-dire une valeur du langage simple de preuve dédié à la vérification). Les opérations légales sur les règles de preuves sont l'initialisation (`generate`) qui engendre les sous-buts, l'extraction[1] d'un terme de preuve (`realize`), la suppression (`delete`) et le raffinement (`refine`), qui a ici pour seul effet de se propager aux prémisses de la règle.

5.6.2.5 `ProofTac`

La dernière famille correspond à `proofTac`, la classe des tactiques composées (par opposition aux règles primitives). Toutes les tactiques qui ne correspondent pas immédiatement à un terme de preuve sont des `proofTac`. Les descendants de `proofTac` auront la même signature que les règles primitives. Le raffinement `refine` renverra un sous-arbre construit uniquement à partir de règles

[1] Ce que nous appelons extraction ici n'est pas l'extraction d'un λ-terme ou d'un programme, mais cela correspond cependant bien à un mécanisme d'extraction : à partir d'une preuve utilisant des étapes de preuves complexes et dont la représentation est complexe, nous fabriquons un terme dont chaque nœud correspond à une étape de preuve primitive, que nous appelons terme de preuve.

primitives. L'appel à la méthode `realize` déclenchera d'abord le raffinement, à partir duquel sera construit le terme de preuve.

Pour simplifier l'écriture de tactique, la méthode `refine` est implémentée dans `proofTac` et ne doit pas être redéfinie, mais le concepteur de tactique devra implémenter une fonction `refineFun` prenant en argument une liste d'arbres (déjà raffinés correspondant aux prémisses) et renvoyant l'arbre de preuve correspondant au raffinement de la tactique courante auquel on aura rattaché le raffinement des prémisses. Cela a pour avantage de ne pas avoir à gérer la récursivité du raffinement au niveau de chaque tactique. Cette approche tire partie de la liaison retardée de l'appel de `refineFun`, et simplifie beaucoup l'écriture du raffinement, qui se présente alors comme la création des règles ou tactiques plus primitives, puis le raccordement de celles-ci entre elles.

Le concepteur de la tactique devra évidemment implémenter la méthode `generate` permettant d'initialiser un nœud et d'engendrer les sous-buts et la méthode `delete` qui sera appelée lors de la suppression de l'étape de preuve.

On peut continuer à hériter de tactiques : par exemple dans le schéma d'héritage, nous présentons `SimplGoals` qui est une classe abstraite redéfinissant la méthode `initGoal` (définie dans `proofStep`, elle crée des `proofGoal` fils à partir des prémisses de la règle). Ici, elle appellera une procédure de décision sur chaque prémisse, puis la méthode `initGoal` de la super-classe seulement sur les prémisses non résolues. Toutes les tactiques en dérivant élimineront donc automatiquement les sous-buts simples.

5.6.2.6 Termes de preuves

Les termes de preuves sont représentés par un type somme dont les nœuds sont les étapes de preuve et contiennent les arguments de la règle appliquée.

Ces termes de preuves sont vérifiées par une fonction implémentée par filtrage. Il est relativement simple pour un utilisateur ayant peu confiance en l'implémentation d'en vérifier cette partie, voire de la réécrire.

* * *

Les règles et tactiques, ainsi que la définition des termes de preuve et la validation dépendent bien entendu du formalisme choisi. La structure présentée, en revanche peut être appliquée à d'autre systèmes de preuve.

Les modules utilisés pour la gestion des preuves sont les suivants :

`ProofTerm` : Langage de représentation des preuves sous forme de type somme et fonctions de vérifications des preuves. Ce module sert à la validation *a posteriori* des preuves.

`ProofObject` : La partie "abstraite" de la hiérarchie des objets représentant les étapes de preuves (voir figure page 141), ainsi que les arbres de preuve (utilisant `IndexedTree`). L'arbre de preuve est un arbre indexé dont les étiquettes sont des étapes de preuve.

`ProofRules` : Les règles de preuve, c'est-à-dire les étapes primitives.

`ProofElts` : Fonctions d'insertion et de suppression d'un élément de preuve dans un élément de session.

5.7 Session

Les éléments de session sont implémentés par des objets. À chaque type d'élément (définition, théorème, etc.) correspond une classe qui hérite de la classe `element`. En dehors de l'ajout d'éléments de sessions pour la définition de types et fonctions ML, les éléments de sessions sont indépendants de l'application.

Les éléments de sessions contiennent un contexte chaîné vers le contexte de l'élément précédent, dans lequel on a ajouté les liaisons créées par cet élément (nom – formule pour une définition, nom – type pour une déclaration, etc.). Les éléments théorèmes contiennent en plus la racine d'une preuve.

On définit également la classe `sessionMgr` (gestionnaire de session) qui prend en charge toutes les opérations sur la session (insertion, suppression d'un élément).

Les deux modules suivants gèrent la notion de session :

`SessionElts` : Définitions des éléments de session : définition, déclaration, théorème, axiome, etc.

`SessionMgr` : L'objet session et ses méthodes. Les éléments de sessions sont chaînés en suivant le contexte de nom.

5.8 Communication et initialisation

De même que les éléments de session, la partie protocole est indépendant du choix de la preuve de programme ML, si ce n'est qu'il faut pouvoir indiquer au moteur la création des éléments de session `type` et `let`.

Ce sont les modules qui prennent en charge l'initialisation du système (création d'une session vide et initialisation de la couche réseau) et les communications avec l'interface.

Le chargement dynamique des tactiques (sur lequel nous donnons plus de détail en 5.10) rend indispensable la définition d'une table des tactiques disponibles où chaque tactique vient inscrire son nom lors de l'initialisation. Cela permet de ne pas avoir à modifier la partie principale du programme lorsque l'on ajoute une tactique. Le chargement dynamique de tactiques permet de ne pas avoir à recompiler le programme (ni même à l'interrompre) lorsque l'on ajoute une nouvelle tactique. Il suffit de compiler la tactique, de la placer dans le répertoire *ad hoc* puis de l'invoquer dans le système pour qu'elle soit chargée. La recherche des tactiques est naïve est se fait par les noms de fichiers.

`LinkTactics` : Prise en charge du chargement dynamique des tactiques et de leur enregistrement, table des tactiques.

`Comm` : Communication réseau de bas niveau (gestion des sockets, initialisation).

`Pep` : Protocole de communication entre le moteur et l'interface. Décodage et encodage des paquets.

`Request` : Définit les requêtes décrites par le protocole sous forme d'un type somme. Cela permet de faire le lien entre la partie réseau et les appels au gestionnaire de session.

`Handler` : Crée un gestionnaire de session et appelle les différentes fonctions d'initialisation (tactiques, réseau, etc.).

5.9 API

Il reste le module `API`, qui importe les modules `BasicTypes`, `TermML`, `Names`, `Formula`, `TypeML`, `Sequent`, `PsParam`, `ProofObject`, `ProofTerm`, `Tables`, `Skolem` et `Typing`, et qui n'en réexporte qu'une partie (ce qui est nécessaire ou utile pour l'écriture des tactiques). Cela permet de factoriser ce qui est nécessaire et de contrôler ce qui est utilisé. Ce module définit également quelque fonctions utiles pour l'écriture des tactiques.

`API` : Contient les fonctions et types autorisés pour l'écriture de tactiques par l'utilisateur. C'est-à-dire la plupart des fonctions des modules énumérés ci-dessus, ainsi que quelque fonctions utilitaires simplifiant l'écriture des tactiques.

5.10 Tactiques

Les tactiques n'apparaissent pas dans le graphe de dépendances car elles sont liées dynamiquement, donc rien ne peut dépendre d'elles. En revanche, elle dépendent du module `API` et de `LinkTactics` (pour s'enregistrer).

Au moment où une tactique est chargée (c'est-à-dire lorsqu'on l'invoque pour la première fois), elle fait appel à la fonction `register` pour enregistrer dans une liste une fonction permettant de créer une nouvelle instance d'un nœud de preuve. Au démarrage, cette liste contient uniquement les règles primitives. Les modules compilés des tactiques doivent être placés dans un répertoire particulier. Ainsi, l'ajout d'une tactique ne nécessite pas de recompiler le système, ni même de relancer le programme. Il suffit de compiler la tactique, de placer le fichier obtenu au bon endroit et d'exécuter la tactique comme si elle existait dans le programme.

Comme nous l'avons déjà dit, il est essentiel de rendre le système évolutif. C'est-à-dire qu'un utilisateur avancé[2] puisse ajouter ses propres tactiques. Les tactiques supplémentaires sont développées en *Objective Caml* et doivent utiliser une API spécifique (cf.

[2]Un utilisateur avancé est un utilisateur connaissant le système, sachant programmer (en *Objective Caml*) et connaissant *certains* aspects internes du système (l'API, et la gestion des noms).

ci-dessus). Nous n'excluons pas, cependant, d'implémenter ultérieurement un *langage de script*, c'est-à-dire un langage de programmation simple dédié aux manipulations sur les formules, les séquents, etc. comme on en trouve dans la plupart des gros AP, parfois sous une forme limitée à de simples combinateurs de tactiques.

Pour développer une tactique, il est nécessaire de sous-classer `proofTac`, notamment d'implémenter les méthodes `generate` (qui engendre les sous-buts) et `refineFun` (qui raffine une étape en étapes plus élémentaires). La fonction `refineFun` étant appelée récursivement, il n'est pas nécessaire de raffiner immédiatement en étapes atomiques, il suffit de raffiner vers des étapes plus simples. Il est à la charge du développeur d'éviter les cycles (par exemple une tactique qui se raffine en elle-même) provoquant alors un bouclage du logiciel.

On doit donc compléter le canevas suivant :

```
let tacticName = "MaTactique"
let tacticKeyW  = "MotClefDeMaTactique"

class maTactique id goal param =
object (self : 'a)
  inherit (proofTac tacticName id goal) as super

  (* generate subgoals *)
  method generate =
         (* genere les sous buts et renvoit true en *)
         (* cas de reussite false en cas d'echec    *)
      <A completer>

  method refineFun premTacL =
         (* prend en argument la liste des sous     *)
         (* preuves et engendre l'arbre raffine     *)
      <A completer>
end

(* Enregistre la tactique *)
let _ =
  LinkTactics.register tacticKeyW (new maTactique)
```

Le `(new maTactique)` que l'on trouve à la fin n'a pas pour effet

de créer une nouvelle instance car on ne lui passe pas tous ses arguments. C'est une application partielle renvoyant une fonction de type `occ -> sequent -> psParam -> step` (qui prend en argument la position dans l'arbre de preuve, un séquent, les arguments encapsulés de la tactique et qui renvoie un `step`, c'est-à-dire une étape de preuve) que l'on enregistre dans la table des tactiques.

Pour illustrer ceci, regardons l'exemple de la tactique `IntroImpl` qui correspond à la règle d'introduction de l'implication dans la déduction naturelle (c'est une version simplifiée de la règle d'introduction de notre système qui est plus générale). Cette tactique est très simple (on ne gère pas de variables et elle ne prend pas d'arguments).

Cette tactique (ce n'est pas une règle primitive) correspond à la règle suivante :

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B} \ (\rightarrow Intro)$$

Nous allons en détailler le code par petits morceaux. Pour avoir une vision d'ensemble, se référer à l'annexe A, page 225.

Ouverture des modules nécessaires.

```
(* Intro of implication *)

open LinkTactics
open API.ProofObject
open API.Formula
open API.ProofRules
open API.PsParam
open API.BasicTypes
open API.TermML
open API.Tables
```

On définit la classe `intro` dont le constructeur prend en argument la position dans l'arbre de preuve, le séquent conclusion et des arguments (même si cette tactique n'en a pas). On donne le nom complet de la tactique, et on indique que l'on hérite de `proofTac`.

```
let tacticName = "Intro. Implication"

class intro id goal param =
```

```
object (self : 'a)
  inherit (proofTac tacticName id goal) as super
```

On commence par définir la méthode `generate`, qui ne prend pas d'arguments. La première chose à faire est de récupérer la conclusion du séquent, puis de créer une copie du séquent conclusion (`newGoal`) que nous allons modifier pour fabriquer la prémisse de la règle.

```
  (* generate subgoals *)
  method generate =
    try
      let currFormula = goal#activeConcl
      and newGoal = goal#clone
      in
      newGoal#subActiveConcl ;
```

Après avoir récupéré la conclusion (active), on la supprime du séquent et on casse l'implication en récupérant les deux formules produites dans `f1` et `f2`. Si le symbole principal de la formule n'est pas l'implication, la fonction déclenche une exception que l'on récupère pour renvoyer `false` à la fin de `generate` (échec de la tactique).

```
      begin
        try
          let f1,f2 = delImplies currFormula
          in
          newGoal#addActiveHyp f1 ;
          newGoal#addActiveConcl f2
        with Wrong_symbol -> raise WrongRule
      end ;
```

Après avoir ajouté au séquent `newGoal` `f1` dans les hypothèses et `f2` comme conclusion, il ne nous reste ci-dessous qu'à stocker dans les prémisses le séquent ainsi obtenu, et à renvoyer `true` pour indiquer que tout s'est bien déroulé.

```
      self#initGoal [newGoal] ;
      true
    with WrongRule -> false
```

La méthode `refineFun` est plus longue. Elle correspond à la traduction de la règle d'introduction de l'implication de la déduction naturelle en un arbre de preuve de la déduction libre.

On veut construire l'arbre (à trou) suivant, où `prem` indique l'endroit où l'on doit recoller la suite de la preuve :

$$\dfrac{\dfrac{}{\Gamma, A \to B \vdash A \to B}\ Ax \quad \dfrac{\dfrac{\dfrac{}{\Gamma, A, A \to B \vdash A \to B}\ Ax \quad \dfrac{\texttt{prem}}{\Gamma, A \vdash A \to B, B}}{\Gamma, A \vdash A \to B} \to g-2}{\Gamma \vdash A \to B} \to g-1$$

On peut noter que le but qu'il nous reste contient une conclusion de plus que pour la règle `intro`.

On commence par récupérer la prémisse et vérifier qu'il n'y en a qu'une.

```
method refineFun premTacL =
  let prem =
    match premTacL with
        [x] -> x
      | _ -> failwith "forallIntro/refineFun/prem"
  in
```

On casse l'implication.

```
  let (a,b) = delImplies concl#activeConcl
  in
```

Puis on crée une étape de preuve $\to$ gauche-1 de même conclusion que la conclusion courante, en lui passant A et B en argument.

```
   let i1 =
      createNodeWithFormList "LeftArrow1" goal [a;b]
   in
```

idem pour $\to$ gauche-2, mais cette fois, la conclusion sera la seconde prémisse (numérotée à partir de 0) de la règle précédente.

```
  let i2 =
      createNodeWithFormList "LeftArrow2"
       (getNthConcl i1 1) [a;b]
  in
```

On traite de la même manière les règles axiomes.

```
  let ax1 =
      createNodeWithoutParam "Axiom"
            (getNthConcl i1 0)
  in
  let ax2 =
      createNodeWithoutParam "Axiom"
```

```
                (getNthConcl i2 0)
    in
```

Il reste à construire l'arbre de preuve (en n'oubliant pas de raccorder aux prémisses. Enfin, on retourne comme résultat la base de l'arbre ainsi construit.

```
    i1#setPremisses [|ax1;i2|] ;
    i2#setPremisses [|ax2;prem|] ;
    i1
end

let _ = LinkTactics.register "IntroImpl" (new intro)
```

La dernière ligne enregistre cette tactique sous la forme d'une fonction, avec comme mot-clef `IntroImpl`. Lorsque le système invoquera la tactique à partir de cette fonction, il lui passera en argument son identificateur, sa conclusion et ses arguments.

Si au cours du raffinement, on passe par une tactique (non primitive), on instancie celle-ci de la même manière, mais il est ensuite nécessaire de la raffiner explicitement, en en fixant les prémisses. Dans l'exemple qui suit (extrait de la tactique `Rec`) on crée une instance `rand` de la classe `AndIntros` qui a pour effet de "casser" une suite de *et* dans la conclusion. On crée ensuite `rand'` qui est le sous-arbre obtenu en raffinant `rand` et on lui passe directement la liste des prémisses (`premTacL`). C'est `rand'` qui est ensuite inséré dans les prémisses d'une autre tactique.

```
let rand =
  createNodeWithoutParam "AndIntros"
     (getNthConcl ard 1)
in
let rand' = rand#refineFun premTacL
in
arr#setPremisses [|ax2;rand';fag|] ;
```

5.11 Remarques sur le langage d'implémentation

Quelques remarques sont nécessaires sur les conséquences du choix du langage d'implantation, en ce qui concerne la structura-

tion des données et le typage.

La couche objet d'*Objective Caml* [59] utilise des classes paramétrées statiques (les classes ne sont pas des objets), avec héritage multiple. Les méthodes n'autorisent pas la surcharge. La relation d'héritage est distincte du sous-typage.

Le typage de ML, contrairement à celui des langages comme Java et Eiffel, est complètement statique. Il garantit l'absence de *toute* erreur de type à l'exécution. Le prix à payer pour cela est que la conversion de type ne fonctionne que d'un sous-type vers le surtype. Cela implique que si l'on veut pouvoir faire des conversions de types dans les deux sens, toute la hiérarchie de classe doit avoir le même type (ce qui nous oblige à contraindre explicitement le type des objets). Dans notre cas, cela ne pose pas de problème : si l'on veut ajouter des méthodes dans les étapes de preuve, on peut les définir privées et les appeler dans `generate` et `refine`. Nous pensons donc que c'est un inconvénient mineur en comparaison des avantages du typage statique (pas d'exception lors de l'invocation de méthodes non trouvées, détection de plus d'erreurs à la compilation, etc.). Mais cela peut poser des problèmes pour modéliser d'autres choses[3] (par exemple interfaces graphiques, rendu 3D etc.).

Au niveau de la structuration du code, *Objective Caml* propose un langage de modules assez riche avec modules paramétrés par des modules (foncteurs). Ces traits peuvent sembler redondants avec les classes car on peut y simuler l'héritage, mais on ne bénéficiera pas de la liaison tardive qui nous est nécessaire. Nous avons choisi ici de ne pas utiliser de foncteurs.

On peut préférer, pour représenter un langage de termes, utiliser un type algébrique ou des objets. Le type algébrique assurant une écriture plus simple, permettant le filtrage (ainsi que la détection des filtrages incomplets et redondants), mais ayant l'inconvénient de ne pas être extensible.

Dans notre implémentation, le choix d'utiliser une représentation objet ou fonctionnelle pour chaque type de donnée était assez direct : pour les termes et les formules, on n'a pas besoin d'étendre les types (le langage de termes est fixé dès le début), mais il est sans cesse nécessaire d'ajouter de nouveaux traitements : il est

[3]Cependant, des solutions existent à ce problème [15].

donc plus simple d'utiliser des types algébriques. À l'opposé, pour les preuves, le type doit être extensible, mais il n'est pas nécessaire d'ajouter de nouveaux traitements globaux. Lorsque l'on veut ajouter une opération, cela peut se faire localement ou par le biais d'une méthode privée. Nous avons donc choisi d'implémenter tout ce qui est langage de terme, de formules, de types et termes de preuves sous la forme de types algébriques. En revanche pour le langage de tactiques nous avons préféré les objets.

Un autre développement mêlant objet et fonctionnel en *Objective Caml* est le projet FOC [10], ayant pour but de créer un environnement de calcul formel *certifié*. Ce projet tire partie de l'héritage et de la liaison tardive des méthodes pour représenter les structures mathématiques (ensembles, groupes, etc.).

Chapitre 6

Résultats

Ce chapitre a vocation de présenter l'état actuel du système PAF! et de servir d'introduction à son utilisation. Les sections 6.4 à 6.5 documentent le système : la grammaire du langage, suivie de la liste des règles et tactiques. Nous terminons sur deux exemples de développements pour illustrer l'utilisation de notre système. Le premier montre la spécification et la vérification d'une fonction de tri par insertion. Le second porte sur des manipulations d'arbres et fait appel à de l'arithmétique simple. Ces exemples ont été choisis en fonction de leur caractère illustratif à la fois comme tutoriel et comme présentation des divers aspects de l'utilisation de PAF!.

6.1 État actuel

Au cours de ce travail de thèse, nous avons réalisé le moteur de preuve tel qu'il est exposé dans les chapitres précédents. Quelques différences existent cependant, que nous décrivons dans ce qui suit.

6.1.1 Ce qui a été fait

La plupart de ce qui est décrit précédemment est implanté et opérationnel. Un jeu de tactiques simples a également été ajouté : cela comprend la plupart des règles de la déduction naturelle, quelque règles du calcul des séquents, des règles prenant en charge l'appli-

cation d'un théorème simple ou d'une égalité et une procédure de décision automatique simple. Le raffinement et la vérification des preuves sont également opérationnels.

Le protocole de communication avec l'interface a été défini de manière efficace par Yves Legrandgérard ([37]) en suivant la spécification (page 106) et a été implanté dans le moteur avec Pascal Manoury. Ce protocole est plus résistant et plus compact que l'implantation "texte" de cette spécification le serait. Il peut facilement prendre en compte les évolutions futures du système sans nécessiter de changements radicaux, et peut également être utilisé indépendamment dans un autre système que PAF!.

Un "toplevel" est fourni pour l'utilisation et les tests du système. La partie liaison dynamique de tactique est également fonctionnelle, bien qu'elle ne fonctionne pas correctement avec le toplevel (le problème semble venir d'*Objective Caml*).

Le code du moteur est réparti dans 41 modules, ce qui correspond à 12981 lignes. À cela, il faut ajouter 31 modules pour les tactiques (soit 2522 lignes). L'ensemble se compile en 9 secondes avec *Objective Caml* version 3.6, sur une architecture Pentium IV 1.5GHz avec 256 Mo de mémoire vive sous FreeBSD 4.6.

6.1.2 Ce qui manque

Quelques parties manquent encore au système : la plus flagrante est l'interface graphique, qui est actuellement développée en parallèle au système de preuve, dans le langage Python couplé avec Tk, notre architecture client-serveur favorisant ce type de développement multi-langage. Il serait également utile d'ajouter une interface ligne de commande pour pouvoir traiter en "batch" (non interactivement) des fichiers.

Une fois l'interface achevée, il serait nécessaire d'avoir la possibilité de sauvegarder et charger une session sans avoir à "rejouer" les preuves jusqu'au point où l'on en était resté.

L'utilisation du système montre également un manque d'automatisation, la tactique `Auto` (décrite dans la section 6.5.2.1) étant insuffisante. Nous reviendrons là-dessus plus loin. Notre volonté étant à terme d'avoir un degré d'automatisation équivalent à celui

d'ACL2. Cela passe également par l'implantation de procédures de décision et de résolution de contraintes pour l'arithmétique.

Nous pensons que notre approche modulaire (orientée "briques") des tactiques et du raffinement permettra de simplifier beaucoup l'obtention de preuves atomiques pour les tactiques automatiques complexes.

Au niveau du moteur de preuve même, il manque encore des fonctionnalités telles que la gestion du graphe de dépendances (pour la suppression d'éléments en milieu de session), qui pourrait conduire à construire une preuve fausse, mais les erreurs en découlant seront détectées dans la phase de validation des preuves.

6.2 Difficultés rencontrées

Lors de ce travail, nous avons rencontré un certain nombre de difficultés que nous avons résolues de façon simpliste, mais qui nécessitent un traitement plus poussé.

6.2.1 Contextes, noms et identificateurs

Ainsi, la gestion des variables est une tâche moins simple qu'il pourrait sembler, surtout dans un programme aussi dynamique que le notre. La solution que nous avons retenue est d'avoir une dualité nom – identificateur. Chaque variable a un nom (une chaîne de caractères) qui lui a été attribué par l'utilisateur ou par le système, mais a aussi un identificateur, qui lui, est unique, et qui n'apparaît jamais à l'utilisateur.

À chaque fois qu'une variable est liée ou déliée, un nouvel identificateur lui est attribué (moralement, ce n'est plus la même variable). Les noms ne sont utilisés que pour communiquer avec l'utilisateur.

On évite ainsi les problèmes de capture de variable, bien que dans certains cas pathologiques, l'utilisateur puisse avoir l'impression que celle-ci s'est produite (deux variables s'appellent x, l'utilisateur ne peut les distinguer, mais le système, lui, sait que ce ne sont pas les mêmes).

Il faudrait envisager un mécanisme complexe pour pouvoir éviter complètement ce problème, car on ne peut pas simplement renommer n'importe comment les variables à chaque fois que la nécessité se présente...

6.2.2 Traitement de l'égalité et de l'évaluation

Un autre problème délicat est celui soulevé par l'égalité. Nous manipulons trois sortes d'égalités : la fonction $=$ de miML (de type $\alpha \to \alpha \to bool$), la β-équivalence, que nous n'utilisons que pour remplacer t par t' lorsque $t \hookrightarrow t'$, et le prédicat d'égalité qui est dénoté par $`(t_1 = t_2)$ où $=$ est la même égalité que celle de miML et $`$ est l'interprétation des booléens dans les valeurs de vérité .

L'évaluation de la fonction d'égalité est plus complexe que dans un langage de programmation. Il ne suffit pas de comparer les deux termes et de voir s'ils diffèrent. Par exemple $\forall x \exists y.`(x = y)$ ne doit pas s'évaluer en $\forall x.\exists y.`(false)$! Nous avons donc implanté l'égalité de façon à ce que $t = t \hookrightarrow true$, $t[C_1(\overrightarrow{u})] = t[C_2(\overrightarrow{v})] \hookrightarrow false$ si t est une chaîne de constructeurs. Si l'on n'est pas dans un de ces cas, on ne réduit pas l'égalité. C'est insuffisant et c'est pourquoi il est nécessaire de rajouter des règles pour le traitement de l'égalité à la Leibniz.

L'évaluation pose d'autres problèmes. Quand doit-on ouvrir les définitions (c'est-à-dire substituer l'objet à l'identificateur) ? Cette question se pose sur deux plans, l'automatisation et le confort de l'utilisateur.

Dans le cas de l'automatisation, si on ouvre les définitions trop tôt, on ne sait plus utiliser les théorèmes et les hypothèses qui portent sur le nom et non sur l'objet. Si on doit prouver `(add x 0)` mais que l'on se retrouve avec
`(match x with 0 -> 0 | S(x) -> S(add x 0)) = x`,
il n'est plus possible d'utiliser directement le théorème de commutativité portant sur `add`. Une solution est d'ouvrir toutes les définitions, dans le séquent courant et dans le théorème. L'inconvénient étant que s'il y a des gardes dans le théorème (quelque chose à prouver pour pouvoir l'appliquer), celles-ci seront complètement dépliées, et l'utilisateur n'y comprendra plus rien.

L'autre problème de l'ouverture des définitions se pose lors de

l'évaluation d'un terme. Si l'on évalue une formule contenant `(add x y)`, on ne veut pas se retrouver avec `(match x with 0 -> y | S(x) -> S(add x y))`, qui embrouille la lecture (surtout s'il y a plusieurs niveaux d'imbrication) et ne constitue pas une étape intéressante dans l'évaluation. Ce que l'on souhaiterait en fait est de n'évaluer `add` que lorsque son premier argument est un zéro ou un successeur. Il faudrait pour cela implémenter des techniques d'évaluation partielle à base de propagation de constantes et d'analyse de flot. La réponse à ce problème est, pour l'instant, de poser et prouver les lemmes équationnels
`(Forall y : nat '((add O y)=y))` et
```
(Forall x :nat (Forall y : nat
 '((add S(x) y)=S((add x y)))))
```
et de ne plus ouvrir la définition de `add`.

6.3 Comparaison avec d'autres systèmes

Il est évidemment utile de nous comparer avec d'autres systèmes, (ceux présentés dans le chapitre 1), du point de vue de la preuve de programmes. Nous compareront également les interfaces de ces systèmes avec notre approche.

Coq L'approche de ce système est assez différente de la notre : en Coq, l'accent est mis sur l'extraction de programmes en ML[1], tandis que nous vérifions les programmes après les avoir écrits. De plus, nous offrons la possibilité de traiter des fonctions partielles, ce qui n'est pas possible dans Coq.

Au niveau de l'automatisation, le système Coq contient de nombreuses procédures de décision spécialisées tandis que notre système souffre d'un manque de tactiques de haut niveau.

Agda/Alfa Ce système et son interface partagent avec PAF! des points communs et des différences radicales. Alfa est assez proche de ce que l'on souhaite obtenir au niveau de l'interface (édition pleine page), mais il est lié à Agda tandis que

[1] Cependant, l'usage de tactiques "dédiées" permettent la preuve de programmes *a posteriori*. C'est notamment le cas avec l'utilisation de la tactique `Program` de C. Parent (voir 1.3.1.3), ou avec la tactique `Correctness` de J.C. Filliâtre permettant la formalisation et la preuve de programmes impératifs (voir [27]).

nous penchons vers une interface et un système indépendants, communiquant à travers un protocole générique.

Contrairement à la plupart des systèmes, Agda/Alfa n'utilise pas de tactiques de haut niveau, mais permet la construction interactive des programmes ou termes de preuves, guidée par le typage car l'interface n'autorise à tout moment que l'insertion d'objets de type adéquat. Preuves et programmes ne forment souvent qu'un, la partie preuve étant intégrée au langage. Il est possible de voir Agda comme un "super ML" permettant de spécifier, programmer et prouver dans le même formalisme. Dans notre système, en revanche, on distingue le langage de programmation, et le langage de spécification et preuve.

Il est possible d'écrire des fonctions ne terminant pas, mais la terminaison doit finalement être vérifiée par un appel explicite a une procédure de décision basée sur les principes d'Abel [1].

L'édition pleine page d'Alfa tire profit des méta-variables, contrairement à notre système qui ne permet pas leur usage.

Isabelle/HOL Il y a peu de points communs entre Isabelle, qui implémente un méta-langage et notre système, qui est dédié à la preuve de programmes ML. Cependant, notre façon de garantir la correction des preuves engendrées dans le système en utilisant une méthode pour engendrer les sous-buts et une pour la preuve primitive est assez similaire à celle de LCF, ancêtre d'Isabelle. Dans LCF, une tactique à deux objectifs : construire les sous-buts et engendrer un morceau de la preuve pour en garantir la correction. On retrouve cette dualité des tactiques dans PAF!, mais une preuve est effectivement construite dès le départ, et la garantie de correction est apportée par le raffinement de cette preuve sous forme de preuve atomique.

Il est à noter que les systèmes diffèrent également du point de vue de l'implémentation. LCF mettant énormément à profit la programmation fonctionnelle (ML) pour l'écriture des tactiques, tandis que nous avons justement préféré pour ce point utiliser les mécanismes offerts par la programmation objet.

PhoX Bien que PhoX ne soit pas dédié à la preuve de programme, le formalisme de notre système se rapproche du sien, basé sur une extension d'AF2 à l'ordre supérieur. Il existe plusieurs manières de définir des fonctions dans PhoX, l'une d'entre

elles étant de les définir de façon équationnelle, puis de prouver qu'elles sont bien définies et qu'elles vérifient les propriétés souhaitées. Dans notre système, les fonctions sont des programmes, écrits dans un sous-ensemble de ML, alors que dans PhoX, elles restent des objets purement mathématiques.

PVS Du point de vue de l'interface, PVS permet l'édition pleine page, mais uniquement au niveau de la "théorie" (c'est-à-dire de la suite de commandes vernaculaires : définitions, théorèmes). La phase de preuve est effectuée dans une fenêtre spécifique où l'utilisateur tape interactivement le script de preuve.

Le formalisme de PVS n'est pas très différent du notre : le langage de terme est un lambda-calcul étendu. En revanche, les fonctions doivent être totales, mais le sous-typage permet d'en décrire précisément les domaines. Un des points nous écartant de PVS est notre volonté de pouvoir engendrer un objet preuve (facilement vérifiable) afin que des erreurs au sein de notre système ne permettent pas de prouver de théorèmes faux.

ACL2 Les buts de ce système sont les mêmes que les notres : avoir un système de preuve portant sur un langage de programmation "réel". Le choix des auteurs d'ACL2 les porte vers LISP, tandis que nous avons choisi ML. Cette différence est plus importante qu'il n'y paraît, car ACL2 utilise largement le fait que la structure de donnée principale est la paire pointée. En revanche nous profitons des types algébriques de ML et de son typage statique.

Une autre différence est qu'ACL2 est basé sur un principe assez peu interactif : l'utilisateur énonce son théorème, et le système tente de le prouver automatiquement. Le bon côté étant que la procédure de décision est très efficace. La contrepartie de l'efficacité de cette procédure est qu'ACL2 ne construit pas d'objet preuve vérifiable.

Nous avons suivi une voie complètement différente, en privilégiant la construction interactive des preuves, et la facilité d'extension du système, l'automatisation restant actuellement notre point faible. Nous avons cependant espoir d'implémenter dans notre système une procédure de décision analogue à celle d'ACL2.

Il est nécessaire de préciser que contrairement à la plupart des systèmes cités, PAF! ne permet pas l'utilisation de méta-variables. En effet, celles-ci posent probème compte tenu des choix d'architecture et d'interface. Cela ne constitue cependant pas impossibilité (les méta-variables existent dans Alfa), mais complique réellement leur implémentation.

Du point de vue de l'interface utilisateur, la plupart des systèmes cités ci-dessus offrent la possibilité d'être utilisés avec Proof General (déjà décrit à la section 1.3.1). C'est notamment le cas de Coq, Isabelle et PhoX. Notre interface, telle qu'elle est spécifiée à la section 4.4, permet une interaction beaucoup plus souple que Proof General. Celui-ci permet l'édition pleine page d'un script dans une fenêtre d'un éditeur de texte, couplée avec des facilités permettant d'aller d'avant en arrière, en utilisant les fonctions d'*undo* et en rejouant le script. La preuve est donc toujours considérée comme une séquence de commandes. Notre interface, en revanche offre de meilleures possibilités de construction interactive pour les preuves (proches de celles de Alfa, par exemple). La différence avec Proof General est surtout perceptible au niveau de l'édition des preuves, qui sont considérées comme des arbres par notre interface. Il est possible de ne pas prouver certains sous-buts, voire de ne pas traiter la preuve d'un théorème et poursuivre sur les théorèmes suivants, et ne revenir compléter les preuves que lorsqu'on le souhaite.

6.4 Utilisation

Cette section et la suivante présentent les commandes disponibles dans le système et la syntaxe du langage.

Les variables `variable` et noms de type `typeNom` sont des identificateurs commençant par une minuscule. Les constructeurs `constr` et les prédicats et variables de prédicats `predicat` commencent par une majuscule et les variables de type `typeVar` par une apostrophe.

Dans ce qui suit, les mots-clefs du langage sont écrits en gras.

6.4.1 Termes

Les termes de miML sont constructibles à partir de la grammaire suivante :

```
terme   :=
            variable
        |   constr
        |   entier
        |   true
        |   false
        |   terme ... terme
        |   fun var -> terme
        |   let var = terme in terme
        |   let rec var = terme in terme
        |   terme = terme
        |   match terme with ptList
        |   constr(terme,...,terme)
        |   <terme,...,terme>                    n-uplet
        |   if terme then terme else terme
        |   fix terme
        |   (terme)
;
```

Dans ce qui précède, la forme `let rec x = t1 in t2` est une macro pour `let x = fix (fun x -> t1) in t2`.

```
ptList  :=
            pat_terme | ptList
        |   pat_terme
;
pat_term:=
            pattern -> terme
;

pattern :=
            variable
        |   constr
        |   entier
        |   true
        |   false
        |   constr(pattern,...,pattern)
        |   <pattern,...,pattern>
;
```

6.4.2 Formules

Les formules sont les suivantes :

```
formule :=
            predicat
        |   predicat(terme,...,terme)
        |   ‘(terme)                              Interprète un
                                                  booléen comme
                                                  une proposition
        |   (formule Or formule)
        |   (formule And formule)
        |   (Not formule)
        |   (formule -> formule)
        |   (Forall variable : type formule)      premier ordre
        |   (Forall predicat : type formule)      second ordre
        |   (Forall typeVar : type formule)       variable de type
        |   (Exists variable : type formule)
        |   (formule : type)
        |   (terme : type)
;
```

La quantification au second ordre est distinguée dans la syntaxe par la casse de la variable et son type. Il n'y a qu'un seul mot-clef.

Les deux formes `(formule : type)` et `(terme : type)` correspondent au prédicat “est typé de type `type` et termine” que l'on note par un point d'exclamation (cf. section 3.3.1). On n'a pas besoin ici d'une notation pour le typage ML (typage faible), dans la mesure où celui-ci est résolu automatiquement par l'inférence de type.

```
schema  :=
            [variable : type,...,variable :
            type]formule
;
type    :=
            typeVar
        |   [type,...,type] typeNom
        |   typeNom
        |   type -> type
        |   type * ··· *type
        |   (type)
;
```

6.4.3 Vernaculaire

```
commd   :=
            Define predicat = formule
        |   Declare variable : type
        |   Declare constr : type
        |   Declare predicat : type
        |   Theorem variable : formule
        |   Axiom variable : formule
        |   Total variable
        |   let variable = terme
        |   let rec variable = terme
        |   type typeprm nom type = cde-
            fList
;
```

La forme `let rec` est ici une vraie définition récursive contrairement au `let rec in` (voir plus haut).

```
typeprm :=
            ϵ
        |   [typevar,...,typevar]
;

cdefList:=
            constrdef
        |   constrdef | cdefList
;
constrdef=
            constr of type * ··· * type
;
```

6.5 Règles et tactiques

Pour invoquer les tactiques, il faut utiliser

```
invoc   :=
            nomtactique paramlist
;
parmlist:=
            param; parmlist
        |   param
        |   ϵ
;
param   :=
            formule
        |   terme
        |   nombre
        |   [type]
;
```

Dans la description qui suit, lorsque les arguments ne sont pas donnés, c'est qu'il n'y en a pas.

6.5.1 Règles

Les règles de base utilisent des séquents multi-conclusion avec au plus une hypothèse et une conclusion dites *actives*. Lorsqu'il n'est pas précisé sur quelle hypothèse ou conclusion agit une règle, c'est qu'elle agit sur les parties actives du séquent. Lorsqu'une nouvelle hypothèse et conclusion est créée, elle est activée dans le séquent.

Il est à noter que seule la conclusion active est visible par l'utilisateur. En revanche, toutes les hypothèses lui sont présentées.

6.5.1.1 Déduction libre enrichie

`Axiom` : Règle axiome : valide si l'hypothèse active et la conclusion active sont égales.

`RightNot` :

Arguments : Une formule A.

"Par l'absurde". $\frac{\vdash \neg A \quad \vdash A}{\vdash}$ Non-droite

`LeftNot` :

Arguments : Une formule A.

"Par cas". $\frac{\neg A \vdash \quad A \vdash}{\vdash}$ Non-gauche

`LeftAnd` :
Arguments : Deux formules A et B.
$\dfrac{A \wedge B \vdash \quad \vdash A \quad \vdash B}{\vdash}$ Et-gauche

`RightAnd1` :
Arguments : Deux formules A et B.
$\dfrac{\vdash A \wedge B \quad A \vdash}{\vdash}$ Et-droite-1

`RightAnd2` :
Arguments : Deux formules A et B.
$\dfrac{\vdash A \wedge B \quad B \vdash}{\vdash}$ Et-droite-2

`LeftOr1` :
Arguments : Deux formules A et B.
$\dfrac{A \vee B \vdash \quad \vdash A}{\vdash}$ Ou-gauche-1

`LeftOr2` :
Arguments : Deux formules A et B.
$\dfrac{A \vee B \vdash \quad \vdash B}{\vdash}$ Ou-gauche-2

`RightOr` :
Arguments : Deux formules A et B.
$\dfrac{\vdash A \vee B \quad A \vdash \quad B \vdash}{\vdash}$ Ou-droite

`LeftArrow1` :
Arguments : Deux formules A et B.
$\dfrac{A \to B \vdash \quad A \vdash}{\vdash}$ Implique-gauche-1

`LeftArrow2` :
Arguments : Deux formules A et B.
$\dfrac{A \to B \vdash \quad \vdash B}{\vdash}$ Implique-gauche-2

`RightArrow` :
Arguments : Deux formules A et B.
$\dfrac{\vdash A \to B \quad \vdash A \quad B \vdash}{\vdash}$ Implique-droite

`LeftForall` :
Arguments : Une formule $\forall x : \tau.A$ et une variable y n'apparaissant pas dans le séquent ni dans l'environnement.
$\dfrac{\forall x : \tau.A \vdash \quad (y \ : \ \tau) \vdash A[y/x]}{\vdash}$ Pour tout-gauche

`RightForall` :

Arguments : Une formule $\forall x : \tau.A$ et un terme t.

$$\frac{\vdash \forall x : \tau.A \quad A[t/x] \vdash \quad \vdash (t \ : \ \tau)}{\vdash}$$ Pour tout-droite

`LeftForall2` :

Arguments : Une formule $\forall X : \tau.A$ et une variable Y n'apparaissant pas dans le séquent ni dans l'environnement.

Cette règle ne s'applique que dans le cas d'une quantification au second ordre.

$$\frac{\Bbb{W}X : \tau.A \vdash \quad \vdash A[Y/X]}{\vdash}$$ Pour tout 2-gauche

`RightForall2` :

Arguments : Une formule $\forall x : \tau.A$ et un schéma de formule S.

Cette règle ne s'applique que dans le cas d'une quantification au second ordre.

$$\frac{\vdash \Bbb{W}X : \tau.A \quad A[S/X] \vdash \quad \vdash (S : \tau)}{\vdash}$$ Pour tout 2-droite

`LeftForallType` :

Arguments : Une formule $\forall \alpha : type.A$ et une variable de type γ n'apparaissant pas dans le séquent ni dans l'environnement.

$$\frac{\forall \alpha : type.A \vdash \quad \vdash A[\gamma/\alpha]}{\vdash}$$ Pour tout type-gauche

`RightForallType` :

Arguments : Une formule $\forall \alpha : type.A$ et un type τ.

$$\frac{\vdash \forall \alpha : type.A \quad A[\tau/\alpha] \vdash}{\vdash}$$ Pour tout type-droite

`LeftExists` :

Arguments : Une formule $\exists x : \tau.A$ et un terme t.

$$\frac{\exists x : \tau.A \vdash \quad \vdash A[t/x] \quad \vdash (t \ : \ \tau)}{\vdash}$$ Il existe-gauche

`RightExists` :

Arguments : Une formule $\exists x : \tau.A$ et une variable y n'apparaissant pas dans le séquent ni dans l'environnement.

$$\frac{\vdash \exists x : \tau.A \quad (y \ : \ \tau) \vdash A[y/x]}{\vdash}$$ Il existe-droite

6.5.1.2 Autres règles

`IConcl` :
$$\frac{}{\vdash \text{`}(true)}\ \text{I-concl}$$

`IHyp` :
$$\frac{}{\text{`}(false) \vdash}\ \text{I-hyp}$$

`StructHypHead` :
Arguments : Le nom d'une hypothèse
Active une hypothèse.

`StructConclHead` :
Arguments : Le nom d'une conclusion
Active une conclusion.

`Use` :
Arguments : Le nom d'un théorème ou d'un axiome.
Charge le théorème dans les hypothèses.

`Open` :
Arguments : Un nom défini.
Ouvre la définition correspondant à ce nom dans la conclusion active.

`OpenHyp` :
Arguments : Un nom défini.
Ouvre la définition correspondant à ce nom dans l'hypothèse active.

`Eval` :
Arguments : Une liste de noms définis ou 0.
Évalue la conclusion active en ouvrant les noms donnés en argument. Si aucun argument n'est donné, ouvre le plus possible de noms. Si l'argument est 0, n'ouvre aucune définition.

`EvalHyp` :
Arguments : Une liste de noms définis ou 0.
Évalue l'hypothèse active en ouvrant les noms donnés en argument. Si aucun argument n'est donné, ouvre le plus possible de noms. Si l'argument est 0, n'ouvre aucune définition.

`InstType` :
Arguments : Une formule.
Substitue la formule donnée en argument à l'hypothèse active si elle est égale modulo instanciation des types.

`Constr` :

$$\frac{\vdash (t_1 : \tau_1) \quad \cdots \quad \vdash (t_n : \tau_n)}{\vdash (C(t_1, \ldots, t_n) : \tau} \text{ constr-type}$$

Où $C : \tau_1 * \cdots * \tau_n \rightarrow \tau$ est un constructeur de type.

`CheckType` :

$$\frac{}{\vdash (t : \tau)} \text{ Type terme-gauche}$$

Si t est bien typé et termine trivialement.

`XApp` :

$$\frac{\vdash t \ : \ \tau' \quad \vdash u \ : \ \tau' \rightarrow \tau}{\vdash t \ u \ : \ \tau} \text{ XApp}$$

`XFun` :

$$\frac{x \ : \ \tau' \vdash t \ : \ \tau}{\vdash \mathit{fun} \ x \rightarrow t \ : \ \tau' \rightarrow \tau} \text{ XFun}$$

6.5.2 Tactiques

6.5.2.1 Pour l'utilisateur

`AndIntro` : Règle d'introduction du "et" de la déduction naturelle.

`AndIntros` : Casse tous les "et" imbriqués (itère la tactique ci-dessus).

`AppEq` :

Arguments : Nom d'un théorème, d'un axiome ou d'une hypothèse .
Applique une égalité conditionnelle. Celle-ci doit avoir la forme :
$\forall \overrightarrow{\alpha} : \mathit{type}. \forall \overrightarrow{x : \tau}. A_1 \wedge \cdots \wedge A_n \rightarrow \text{`}(t_1 = t_2)$ ou
$\forall \overrightarrow{\alpha} : \mathit{type}. \forall \overrightarrow{x : \tau}. A_1 \rightarrow \cdots \rightarrow A_n \rightarrow \text{`}(t_1 = t_2)$.
Cette règle substituera dans la conclusion active la partie droite de l'égalité à un sous-terme de la conclusion correspondant à la partie gauche (moyennant filtrage). Des prémisses sont crées pour vérifier les hypothèses du théorème.

`AppEqH` :

Arguments : Nom d'une hypothèse, puis nom d'un théorème, d'un axiome ou d'une hypothèse .
Identique à la tactique ci-dessus, mais l'égalité est appliquée sur l'hypothèse donnée comme premier argument.

`AppEqHRev` :

Arguments : Nom d'une hypothèse, puis nom d'un théorème,

d'un axiome ou d'une hypothèse .
Identique à la `AppEqH`, mais l'égalité est traitée de droite à gauche.

`AppEqRev` :
Arguments : Nom d'un théorème, d'un axiome ou d'une hypothèse .
Identique à la `AppEq`, mais l'égalité est traitée de droite à gauche.

`Apply` :
Arguments : Nom d'un théorème, d'un axiome ou d'une hypothèse .
Applique une formule. Celle-ci doit avoir la forme :
$\forall \overrightarrow{\alpha} : type. \forall \overrightarrow{x : \tau}. A_1 \wedge \cdots \wedge A_n \rightarrow B$ ou
$\forall \overrightarrow{\alpha} : type. \forall \overrightarrow{x : \tau}. A_1 \rightarrow \cdots \rightarrow A_n \rightarrow B$.
Cette règle vérifie que B filtre la conclusion et demande à prouver les prémisses sur lesquelles on applique la substitution.

`Assume` : Par hypothèse. Essaie d'appliquer une des règles du I ou une hypothèse.

`Auto` : Procédure de décision pour les buts simples. Cette procédure évalue la conclusion et les hypothèses, applique toutes les règles d'introduction et essaie d'appliquer les hypothèses (pas de *backtrack* en cas d'échec). Elle essaie également de résoudre les énoncés de terminaison triviaux.

`BoolIf` :
Arguments : Une fonction propositionnelle (cf. définition 15) dont le corps est la conclusion active avec un "trou" à la place du `if ... then ... else ...` de la formule à prouver.
Engendre un sous-but par branche du `if` en ajoutant en hypothèse la condition (resp. sa négation) et en substituant au `if` sa branche `then` (resp. sa branche `else`).

`ByEq` : Essaie d'appliquer une égalité (évite les égalités dont la partie droite est filtrée par la partie gauche, comme pour `AppEqNoMatch`).

`ByInduction` :
Arguments : Un nom de variable (optionnel).
Effectue une induction sur la variable indiquée (ou sur la première) quantifiée universellement dans la formule. Si la variable est précisée et qu'il y a plusieurs quantificateurs uni-

versels, cette règle les permute de manière à avoir l'hypothèse d'induction la plus forte. `Auto` est appliqué sur tous les sous-buts. En cas d'échec de celle-ci, la tactique effectue uniquement l'induction.

`ByTheorem` : Essaie d'appliquer les théorèmes connus (ayant la forme applicable pour `Apply`).

`Contrap` : Applique la contraposition sur la conclusion du séquent.

`ElimForall` :
Arguments : Un terme.
Élimination du "pour tout" de la déduction naturelle.

`EqConstr` : Élimine le même constructeur de part et d'autre d'une égalité.

`EvalAllHyps` : Évalue toutes les hypothèses en ouvrant les définitions.

`Intro` : Introduction du "pour tout" (premier et second ordre), du "pour tout type" ou de l'implication.

`IntroFaType` : Introduction du "pour tout type".

`IntroForall` : Introduction du "pour tout" (au premier et second ordre).

`Intros` : Itère `Intro` autant que possible, et au moins une fois.

`Is` : Applique les règles du I.

`LOr` :
Arguments : Un nom d'hypothèse.
Applique la règle gauche du "ou" du calcul des séquents.

`Left` :
Arguments : Un nom d'hypothèse, et soit un terme, soit une formule, soit une variable, soit un nom d'hypothèse, soit un type.
Applique la règle gauche du calcul des séquents du pour tout (premier et second ordre ou type) ou de l'implication.

`OpenAndEval` :
Arguments : Un nom défini.
Ouvre la définition correspondant à ce nom dans la conclusion active (`Open`) puis évalue sans ouvrir de noms (`Eval 0`).

`ROr` :
Arguments : 1 ou 2.
Applique la règle (1 ou 2) d'introduction du "ou".

`Rec` :

Arguments : Nom du principe d'induction (optionnel).
Effectue une récurrence sur la variable quantifiée universellement en tête. Si la règle est appliquée sans argument, on utilise l'induction structurelle sur le type de la variable. Sinon, il faut donner un théorème de la forme :
$\forall \overrightarrow{\alpha : type}.\forall X : \tau \to Prop.(A_1 \wedge \cdots \wedge A_n) \to \forall x : \tau.X(x)$.

`Step` :

Arguments : Une liste de noms définis.
Évalue d'un pas la conclusion en ouvrant les noms donnés en argument. Si aucun argument n'est donné, n'ouvre pas les définitions.

`StepHyp` :

Arguments : Une liste de noms définis.
Évalue d'un pas l'hypothèse active en ouvrant les noms donnés en argument. Si aucun argument n'est donné, n'ouvre pas les définitions.

`SwapL` :

Arguments : Un nom d'hypothèse et un entier.
Permute le premier et le n-ième quantificateur universel dans l'hypothèse spécifiée.

`SwapR` :

Arguments : Un entier.
Permute le premier et le n-ième quantificateur universel dans la conclusion.

`TypeIf` :

$$\frac{\vdash c \ : \ bool \quad \vdash e_1 \ : \ \tau \quad \vdash e_2 \ : \ \tau}{\vdash (if \ c \ then \ e_1 \ else \ e_2) \ : \ \tau} \text{ TypeIf}$$

6.5.2.2 Pour l'automatisation

Les tactiques décrites ci-dessous ne sont utiles que pour l'écriture d'autres tactiques plus complexes.

`AppEqNoMatch` : Fonctionnement identique à `AppEq`, mais échoue si la partie droite de l'égalité est filtrable par sa partie gauche (par exemple, les théorèmes de commutativité).

`EvalFail` :

Arguments : Une liste de nom

Fonctionne comme la règle primitive `Eval` sur la conclusion mais échoue s'il n'y a pas eu de modifications.

`ReadyVar` :

Arguments : Une variable liée dans la conclusion.
Introduit tous les "pour tout type". S'il reste une suite de quantifications universelles, fait passer la quantification sur la variable donnée en argument en tête. Cette tactique s'utilise pour préparer à une récurrence.

6.6 Remarque sur l'écriture des tactiques

Diverses méthodes ont été utilisées pour implémenter les tactiques décrites dans la section précédente.

Certaines tactiques utilisent une approche directe pour la création des sous-buts : les formules du séquent conclusion sont manipulées directement jusqu'à obtenir les séquents prémisses.

D'autres tactiques sont écrites en n'utilisant que des tactiques plus simples, que ce soit pour la construction des sous-buts ou pour le raffinement. `Auto`, par exemple itère sur la liste de fonctions ci-dessous jusqu'à échouer ou arriver à résoudre tous les sous-buts.

```
let taclist ctx = [
 (fun g -> createNodeWithoutParam ctx "Assume" g) ;
 (fun g -> createNodeWithoutParam ctx "ByHyp" g) ;
 (fun g -> createNodeWithoutParam ctx "CheckType" g) ;
 (fun g -> createNodeWithoutParam ctx "Intros" g) ;
 (fun g -> createNodeWithoutParam ctx "ByEq" g) ;
 (fun g -> createNodeWithoutParam ctx "Constr" g) ;
 (fun g -> createNodeWithoutParam ctx "EvalFail" g) ;
 (fun g -> createNodeWithoutParam ctx "EvalAllHyps" g)
]
```

La dernière méthode d'écriture de tactique consiste à utiliser un théorème de la bibliothèque de base ou un théorème engendré, combiné à l'utilisation de la tactique `Apply` et une remise en forme éventuelle des prémisses. Cette technique peut s'appliquer aussi bien pour la construction des sous-buts (`Rec`) ou pour le raffinement de la tactique (`BoolIf`).

6.7 Interface *toplevel*

L'interface fonctionne en manipulant une pile. En mode session, le sommet de la pile contient le dernier élément de session inséré. En mode preuve, la pile contient les sous-buts qui restent à prouver.

L'interface *toplevel* est lancée par la commande :

```
./spatop
```

Pour avoir accès aux fonctions de manipulation de session et de preuve, il est utile d'ouvrir le module `Topfun` par la commande

```
open Topfun ;;
```

Les fonctions principales de manipulation sont :

```
val p : string -> unit
```

Insère un élément de preuve à la suite. L'argument contient le nom de la tactique et ses paramètres.

Par exemple : `p "Left H1" ;;`

```
val v : string -> unit
```

Insère un élément de session à la suite. L'argument contient le type de l'élément et ses arguments.

Par exemple : `v "Theorem facile : (A -> A)" ;;`

```
val abort : unit -> event
```

Recommence une preuve du début.

```
val del : unit -> event
```

Supprime le dernier élément dans la pile.

```
val hd : unit -> event
```

Retourne le dernier élément dans la pile.

```
val next : unit -> unit
```

Affiche le prochain élément à prouver.

6.8 Exemple

6.8.1 Tri par insertion

Le premier exemple que nous donnons pour illustrer notre système est la preuve de correction d'une fonction de tri par insertion. Nous avons choisi ce problème car il est assez simple à traiter mais assez complet et illustratif, cet exemple ayant pour but de présenter l'utilisation du système, ainsi que de servir de tutoriel.

Cet exemple met en évidence une faiblesse de notre système : son manque de tactiques. Le langage de tactique de PAF! reste pour l'instant assez sommaire, le développement du moteur ayant pris beaucoup de temps, mais le système n'est encore que dans une version préliminaire.

La première partie de la preuve consiste à montrer que cette fonction retourne une liste triée. La seconde partie vérifiera que cette fonction préserve le contenu de la liste.

Toutes les preuves ne sont pas données *in extenso*. Pour tout vérifier, il a fallu utiliser 180 étapes de preuves (y compris les lemmes simples sur *inférieur ou égal*).

Ce qui suit a été prouvé dans un toplevel *Objective Caml* étendu par des fonctions de manipulation du système de preuve : insertion d'un élément, suppression, modification des preuves. Comme dans LCF ou ses descendants, on a ainsi tout le pouvoir d'expression d'*Objective Caml* à disposition.

Pour ajouter une commande vernaculaire, on passe celle-ci en argument à la fonction `v`. Par exemple `v "Declare A : Prop" ;;`. Pour les commandes de preuve, on utilise la commande `p` (par exemple `p "Apply H3" ;;`). Nous ne donnons pas en général les réponses du système.

6.8.2 Types, fonctions et lemmes de base

Nous commençons par définir les entiers et les listes (bien qu'ils soient définis dans la bibliothèque).

```
v "type nat = O | S of nat" ;;
v "type ['a] list = N | C of 'a*['a] list" ;;
```

Ensuite, nous définissons la fonction `le`, inférieure ou égale, exprimant la relation d'ordre sur les entiers.

```
v "
let rec le =
  (fun x -> (fun y ->
              (match x with
                   O -> true
                 | S(x) ->
                     (match y with
                          O -> false
                        | S(y) -> (le x y))))" ;;
```

Prouvons d'abord la terminaison de la fonction `le`. La commande `Total` permet d'engendrer la formule à prouver. engendre automatiquement la formule à prouver :

```
v "Total le" ;;

|-
(Forall z7:nat (Forall z6:nat ((le z7 z6):bool)))
```

On essaie de prouver cette formule par induction (sur la première variable). Le premier sous-but `(Forall z6:nat ((le O z6):bool))` est vérifié automatiquement par la tactique `Auto`.

```
p "ByInduction" ;;
p "Auto" ;;
```

Le second sous-but

```
(Forall x12:nat ((Forall z6:nat ((le x12 z6):bool))
             -> (Forall z6:nat ((le S(x12) z6):bool))))
```

se prouve par induction (qui résout ici automatiquement les deux sous-buts).

```
p "Intro" ;;
p "Intro" ;;
p "ByInduction" ;;


Proved.
```

Prouvons maintenant un premier lemme dont nous auront besoin ultérieurement. Ce lemme indique que si `x`≤ 0, alors `x` est nul.

On écrit `‘((le x O))` et non directement `(le x O)` car `le` est une fonction renvoyant un booléen et l'on distingue les booléens des propositions. Le prédicat `‘` interprète un booléen en une valeur de vérité dans les propositions (`‘ : bool -> Prop`). Il correspond au I défini page 86.

```
v "Theorem leO :
    (Forall x : nat (‘((le x O)) -> ‘((x = O))))" ;;
p "ByInduction" ;;


Proved.
```

Prouvons maintenant le lemme exprimant la réflexivité de `le`.

```
v "Theorem le_refl :
 (Forall x : nat
     ‘((le x x)))" ;;
p "ByInduction" ;;


Proved.
```

Les deux théorèmes suivants expriment que si $x \not\leq y$, alors $y \leq x$, et si $x \not\leq y$, alors $x \neq y$.

```
v "Theorem notle :
 (Forall x : nat
   (Forall y : nat
     ((Not ‘((le x y))) -> ‘((le y x)))))"
;;
```

La preuve se fait par induction sur x, puis, dans le cas successeur, par induction sur y.

```
p "ByInduction" ;;
p "Eval" ;;
p "Intros" ;;

p "RightNot ‘(true)" ;;
p "Assume" ;;
p "Assume" ;;

p "Intro" ;;
p "Intro" ;;
p "ByInduction" ;;

v "Theorem notle_noteq :
 (Forall x : nat (Forall y : nat
   ((Not ‘((le x y))) -> (Not ‘((x = y))))))" ;;
```

Ce lemme se prouve aisément par induction, puis en cherchant à montrer la contraposée.

```
p "ByInduction" ;;
p "Intro" ;;
p "Contrap" ;;
p "Auto" ;;

p "Intro" ;;
p "Intro" ;;
p "Intro" ;;
p "Contrap" ;;
p "Intro" ;;
```

Il nous reste à montrer

```
[H4 : ‘((S(x13)=y1))]
H3 : (y1:nat)
H2 : (Forall y:nat ((Not ‘((le x13 y))) -> (Not ‘((x13=y)))))
H1 : (x13:nat)
|-
‘((le S(x13) y1))
```

On applique l'égalité de l'hypothèse `H4` (on substitue le membre droit au membre gauche de l'équation dans la conclusion), puis on utilise la réflexivité de `le` pour prouver `'((le y1 y1))`.

```
p "AppEq H4" ;;
p "Apply le_refl" ;;

Proved.
```

6.8.3 Insertion et tri

Nous définissons la fonction `insert` qui insère un élément en place dans une liste triée, puis la fonction `sort`.

```
v "
let rec insert =
(fun x ->
 (fun l ->
  (match l with
    N -> C(x,N)
  | C(y,m) ->
        if (le x y)
        then C(x,l)
        else C(y,(insert x m)))))" ;;
```

Le mécanisme d'inférence de type répond :

```
insert defined of type (nat -> ([nat] list -> [nat] list))
```

```
v "
let rec sort =
 (fun l ->
   (match l with
      N -> N
    | C(x,l) -> (insert x (sort l))))" ;;
```

Nous déclarons maintenant le prédicat inductif `Sorted` (être trié), puis nous donnons les axiomes le définissant, ainsi que les axiomes d'inversion. Nous n'avons pas dans le système de définition de prédicat inductif, nous devons donc utiliser des axiomes.

Une alternative aurait été de donner la fonction ML `is_sorted` de type `[nat] list -> bool` et d'utiliser ‘ pour obtenir un prédicat.

```
v "Declare Sorted : ([nat] list -> Prop)" ;;

v "Axiom sorted0 : Sorted(N)" ;;
v "Axiom sorted1 : (Forall n : nat (Sorted(C(n,N))))" ;;
v "Axiom sorted2 :
  (Forall n1 : nat
   (Forall n2 : nat
    (Forall l : [nat] list
     ((Sorted(C(n2,l)) And ‘((le n1 n2)))
       ->  (Sorted(C(n1,(C(n2,l))))))))" ;;


v "Axiom sorted_inv1 :
   (Forall x : nat
    (Forall y : nat
     (Forall l : [nat] list
      (Sorted(C(x,C(y,l))) -> ‘((le x y))))))" ;;

v "Axiom sorted_inv2 :
(Forall x : nat
 (Forall l : [nat] list
  (Sorted(C(x,l)) -> Sorted(l))))" ;;
```

Nous prouvons la totalité des fonctions `insert` et `sort` (nous n'en donnons pas les preuves ici). Prouver la totalité d'une fonction n'est pas indispensable (les fonctions partielles sont autorisées dans notre système), mais cela permet de traiter aisément (et automatiquement) les énoncés de typage les concernant.

```
v "Total insert" ;;

v "Total sort" ;;
```

6.8.4 `sort` renvoie une liste triée

Le premier lemme que nous posons indique que si on insère un élément y plus grand que x dans une liste triée commençant par x, alors la liste commençant par x, suivi du résultat de l'insertion de y dans le reste de la liste est triée. Ce lemme nous servira à prouver le cas inductif de `sortedIns`.

```
v "Theorem lemme1 :
 (Forall l : [nat] list
  (Forall x : nat
   (Forall y : nat
    (‘((le x y)) ->
      (Sorted(C(x,l)) -> Sorted(C(x,(insert y l))))))))"
;;
```

Cette proprété se montre par récurrence sur l.

```
p "ByInduction l" ;;
```

Dans le cas où la liste est vide, il suffit d'évaluer après avoir "ouvert" la définition de `insert`, puis d'appliquer sur le résultat les lemmes `sorted2` et `sorted1`. Les sous-buts restant sont résolus par `Auto`.

```
p "Intros" ;;
p "OpenAndEval insert" ;;
p "Apply sorted2" ;;
p "Apply sorted1" ;;
p "Auto" ;;
p "Auto" ;;
```

Le cas où la liste n'est pas vide est plus complexe. On commence par évaluer en ouvrant la définition de `insert`.

```
p "Intros" ;;
p "OpenAndEval insert" ;;
```

On obtient :

```
[H8 : Sorted(C(x16,C(x14,x15)))]
H7 : '((le x16 y1))
H6 : (y1:nat)
H5 : (x16:nat)
H4 : (Forall x:nat (Forall y:nat ('((le x y))
     -> (Sorted(C(x,x15))
     -> Sorted(C(x,(insert y x15)))))))
H3 : (x15:[nat] list)
H2 : (x14:nat)
|-
Sorted(C(x16,(if (le y1 x14)
              then C(y1,C(x14,x15))
              else C(x14,(insert y1 x15)))))
```

La preuve se fait par cas sur `(le y1 x14)`, en utilisant la commande (`BoolIf`).

```
p "BoolIf ([v:[nat] list](Sorted(C(x16,v))))" ;;
```

Dans le cas où `(le y1 x14)`, on doit prouver :
`Sorted(C(x16,C(y1,C(x14,x15))))`.
On applique deux fois `sorted2`, puis, pour prouver
`Sorted(C(x14,x15))`,
on utilise l'inversion `sorted_inv2`.

```
p "Apply sorted2" ;;
p "Apply sorted2" ;;
p "Use sorted_inv2" ;;
```

Avant de pouvoir l'appliquer on doit instancier le "pour tout".

```
p "Left H10;x16" ;;
p "Apply H11" ;;
```

Nous devons maintenant prouver `Sorted(C(x16,C(x14,x15)))` que nous avons en hypothèse. Cela conclut cette branche de la preuve.

La suppression des hypothèses `H10` et `H11` est nécessaire pour qu'elle ne soient pas prises en compte par `Auto`. On élimine ensuite les sosu-buts résiduels.

```
p "DelHyp H10;H11" ;;
p "Auto" ;;
p "Auto" ;;
p "Auto" ;;
```

Le cas où (`le y1 x14`) est faux est similaire, mais il utilise les deux inversions du prédicat `Sorted`. Nous donnons la suite de commandes utilisées sans plus de commentaires.

```
(* cas not le *)
p "Apply sorted2" ;;
p "Apply H4" ;;

p "Use sorted_inv2" ;;
p "Left H10;x16" ;;
p "Apply H11" ;;

p "DelHyp H11;H10" ;;

p "Assume" ;;

p "Apply notle" ;;
p "Assume" ;;


p "Use sorted_inv1" ;;
p "Left H10;x16" ;;
p "Left H11;x14" ;;
p "Left H12;x15" ;;
p "Apply H13" ;;
p "Assume" ;;

Proved.
```

Le second lemme (prouvé en utilisant le précédent) montre que si l'on insère un élément dans une liste triée, alors la liste en résultant est triée. C'est le lemme d'invariance de `insert`.

```
v
```

```
"Theorem sortedIns :
  (Forall l : [nat] list
   (Forall x : nat
    (Sorted(l) -> Sorted((insert x l)))))" ;;
```

Par induction sur la liste `l`, puis dans le cas où la liste est vide, on applique la définition de `Sorted`.

```
p "ByInduction" ;;

p "Intros" ;;
```

`Eval` sans argument ouvre toutes les définitions rencontrées dans le processus d'évaluation. `Apply` élimine automatiquement les sous-buts triviaux.

```
p "Eval" ;;
p "Apply sorted1" ;;
```

Prouvons le cas de récurrence :

```
[H5 : Sorted(C(x14,x15))]
H4 : (x16:nat)
H3 : (Forall x:nat (Sorted(x15)
        -> Sorted((insert x x15))))
H2 : (x15:[nat] list)
H1 : (x14:nat)
|-
Sorted((insert x16 C(x14,x15)))
```

On commence par ouvrir la définition d'`insert`, puis on raisonne par cas sur
`(le x16 x14)`.

```
p "Intros" ;;
p "OpenAndEval insert" ;;

p "BoolIf ([v:[nat] list](Sorted(v)))" ;;
```

Dans le cas où `x16` est inférieur ou égal à `x14`, on doit prouver `Sorted(C(x16,C(x14,x15)))`.

```
(* cas le *)

p "Apply sorted2" ;;
p "Auto" ;;
p "Auto" ;;
```

Dans le cas contraire, on doit prouver
`Sorted(C(x14,(insert x16 x15)))`.
On applique le lemme 1 qui donne un premier sous-but trivial, puis le lemme `notle` pour transformer
`Not (le x16 x14)` en `(le x14 x16)` et cela nous permet de conclure la preuve.

```
p "Apply lemme1" ;;
p "Assume" ;;
p "Apply notle" ;;
p "Assume" ;;

Proved.
```

Le dernier théorème indique que pour toute liste `l`, `(sort l)` est triée. Il suffit de charger les lemmes adéquats dans les hypothèses (`Use`) et d'appliquer `ByInduction` qui résout d'elle-même les sous-buts.

```
v "Theorem th1 :
 (Forall l : [nat] list (Sorted((sort l))))" ;;
p "Use sortedIns" ;;
p "Use sorted0" ;;
p "Use sorted1" ;;
p "Use sorted2" ;;
p "ByInduction" ;;
```

6.8.5 `sort` préserve les éléments de la liste

Nous avons prouvé que `sort` triait les listes. Il nous reste maintenant à vérifier que cette fonction préserve les éléments de la liste.

Pour cela, nous définissons la fonction
`occ : nat -> [nat] list -> nat` comptant le nombre d'occurrences d'un entier dans une liste.

```
v "
let rec occ =
  (fun n ->
   (fun l ->
    (match l with
       N      -> O
     | C(x,l) -> if (n = x)
                 then  S((occ n l))
                 else (occ n l))))" ;;
```

Nous ne donnons pas les scripts de preuves des lemmes qui suivent car ils n'ajoutent rien de nouveau à ce qui a été dit précédemment.

Les deux premiers lemmes vont nous servir à montrer `insertOccYes` et `insertOccNo`. Le premier lemme indique que l'ajout en tête d'un élément `y` (différent de `x`) dans la liste n'augmente pas le nombre d'occurrences de `x`.

```
v "Theorem lemme_occ_1 :
 (Forall x : nat
  (Forall y : nat
   (Forall l : [nat] list
    ((Not '(x = y)) ->
     '( (occ x C(y,l)) = (occ x l)    )))))" ;;
```

Le second lemme indique qu'ajouter un élément en tête de liste ajoute une occurrence de cet élément dans la liste.

```
v "Theorem lemme_occ_2 :
 (Forall x : nat
  (Forall l : [nat] list
    '( (occ x C(x,l)) = S((occ x l))  )))" ;;
```

Voici maintenant les deux lemmes qui nous intéressent en regard du problème de tri : le premier indique que l'insertion (*via* la

fonction `insert`) d'un élément `y` (différent de `x`) n'augmente pas le nombre de `x`.

Le second indique que l'insertion d'un élément `x` incrémente le nombre de `x` dans la liste.

```
v "Theorem insertOccNo :
  (Forall l : [nat] list
   (Forall x : nat
    (Forall y : nat
     ((Not(‘(x = y)))
         -> ‘((occ x (insert y l)) = (occ x l))))))" ;;

v "Theorem insertOccYes :
  (Forall l : [nat] list
   (Forall x : nat
    ‘((occ x (insert x l)) = S((occ x l)))))" ;;
```

Les preuves de ces deux lemmes se font par induction, puis par cas sur le fait que l'élément inséré est plus petit que l'élément en tête de liste.

Enfin, le théorème qui nous permet de conclure : pour tout entier `n`, le nombre d'occurrences de `n` dans la liste est préservé par `sort`.

```
v "Theorem th2 :
 (Forall l : ([nat] list)
  (Forall x : nat
   ‘((occ x (sort l)) = (occ x l))))" ;;
p "ByInduction" ;;
p "Auto" ;;


[H4 : (x16:nat)]
H3 : (Forall x:nat
        ‘(((occ x (sort x15))=(occ x x15))))
H2 : (x15:[nat] list)
H1 : (x14:nat)
|-
‘(((occ x16 (insert x14 (sort x15)))
  =(occ x16 C(x14,x15))))
```

Dans le cas de récurrence, on raisonne par cas (`LeftNot`) sur l'égalité
`(x16=x14)`. Dans chacun des cas, on applique l'égalité `insertOccYes` ou `insertOccNo` (`AppEq`).

```
p "Intros" ;;
p "OpenAndEval sort" ;;

p "LeftNot ‘((x16=x14))" ;;
(* x16 != x14 *)
p "AppEq insertOccNo" ;;
p "AppEq lemme_occ_1" ;;
p "Auto" ;;

(* x16 = x14 *)
p "AppEq H5" ;;
p "AppEq H5" ;;

p "AppEq insertOccYes" ;;
p "AppEq lemme_occ_2" ;;
p "Auto" ;;
```

La preuve de ce théorème permet de conclure sur la correction de notre fonction de tri par insertion.

6.8.6 Arithmétique et manipulations d'arbre

L'exemple précédent illustrait la preuve de l'implémentation d'un algorithme simple. Nous passons maintenant à un exemple plus consistant, mélangeant arithmétique et manipulations non triviales sur des arbres.

Cet exemple est présenté en trois parties. Dans la première, nous définissons l'addition et la multiplication dans les entiers naturels et en démontrons les propriétés fondamentales. Nous définissons ensuite les opérateurs booléens `and` et `or`, puis les relions à leurs équivalents propositionnels.

Dans la dernière partie, nous définissons un langage d'arbre, ou, plus précisément, de termes représentant les expressions arithmétiques, restreintes à l'addition et à la multiplication. Nous donnons à ce langage une sémantique, sous forme d'un évaluateur. Cela fait, nous optimisons cet évaluateur et nous montrons qu'il est équivalent au précédent. Enfin, nous définissons une relation binaire sur les termes à partir d'une fonction ML retournant un booléen. Cette fonction permet de déterminer si deux arbres sont égaux, à permutation près des fils de chaque noeud. Nous finissons cet exemple en montrant que si deux arbres sont équivalents pour cette fonction, alors les résultats de leurs évaluations sont égaux.

Nous commençons par définir des fonctions dans le *toplevel Objective Caml* pour simplifier l'écriture des preuves. La fonction `multi` permet d'exécuter `n` fois la tactique `s`, et les fonctions `intro` et `auto` sont spécialisées pour les tactiques correspondantes.

```
let multi s n =
  for i = 1 to n do p s done ;;
let intro = multi "Intro" ;;
let auto = multi "Auto" ;;
```

6.8.6.1 Arithmétique

Dans la première partie de l'exemple, nous définissons l'addition et la multiplication sur les entiers. Nous posons des lemmes qui simplifieront ultérieurement les preuves, et nous montrons les propriétés fondamentales de l'addition et la multiplication : associativité, commutativité, distributivité de l'addition sur la multiplication et simplification d'une égalité $a + x = a + y$.

La définition de l'addition est usuelle :

```
v "
let rec add =
 (fun x ->
  (fun y ->
   (match x with
      O    -> y
    | S(x) -> S((add x y)))))" ;;
```

Nous montrons les propriétés de base de l'addition, que les tactiques automatiques suffisent à démontrer immédiatement. Nous avons tout d'abord la totalité de l'addition, puis les équations : $0 + x = x$, $x + 0 = x$, $(succ(x) + y) = succ(x + y)$ et $(x + succ(y)) = succ(x + y)$.

```
v "Total add" ;;
p "ByInduction" ;;

v "Theorem addOx :
     (Forall x : nat ‘((add O x) = x))" ;;
p "Auto" ;;

v "Theorem addxO :
     (Forall x : nat ‘((add x O) = x))" ;;
p "ByInduction" ;;

v "Theorem addSx :
 (Forall x : nat
   (Forall y : nat
     ‘((add S(x) y) = S((add x y)))))" ;;
p "Auto" ;;

v "Theorem addxS :
 (Forall x : nat
   (Forall y : nat
     ‘((add x S(y)) = S((add x y)))))" ;;
p "ByInduction" ;;
```

Les lemmes ci-dessus semblent être des redites par rapport à la définition de la fonction `add`, mais ils s'avèrent utiles par la suite pour nous permettre de manipuler cette fonction sans avoir à "ouvrir" sa définition. On peut les voir comme des règles d'introduction de l'addition.

Le lemme suivant exprime que $x+y = x+z$ est équivalent à $y = z$. L'égalité étant une fonction renvoyant un booléen, et les booléens étant des valeurs, on peut choisir d'exprimer cette équivalence par une égalité. Cela nous permet, par la suite, de l'utiliser pour simplifier les égalités de la forme $x + y = x + z$ en utilisant la tactique `AppEq`. Ce lemme est prouvé automatiquement par induction.

```
v "Theorem simp_left_add :
 (Forall x : nat
  (Forall y : nat
   (Forall z : nat
   '(( (add x y) = (add x z)  ) = (y = z)))))" ;;
p "ByInduction" ;;
```

Nous passons maintenant aux propriétés de l'addition qui nous intéressent : l'associativité et la commutativité. Bien que l'automatisation de notre système ne soit pas très poussée, ces lemmes sont prouvés automatiquement par la tactique `ByInduction`, qui utilise les lemmes préalablement prouvés.

```
v "Theorem add_assoc :
 (Forall x : nat
   (Forall y : nat
     (Forall z : nat
      '((add (add x y) z) = (add x (add y z))))))" ;;
p "ByInduction" ;;

v "Theorem add_comm :
 (Forall x : nat
   (Forall y : nat
      '((add x y) = (add y x))))" ;;
p "ByInduction" ;;
```

Nous passons maintenant à la multiplication, définie récursivement en utilisant la somme.

```
v "
let rec mul =
 (fun x ->
  (fun y ->
   (match x with
      O    -> O
    | S(x) -> (add y (mul x y)))))" ;;
```

Contrairement à l'addition, la totalité de la multiplication ne peut pas être prouvée par un simple appel à une tactique automatique. Notre tactique `Auto` (donc `ByInduction`, qui l'utilise) n'effectue pas de retour en arrière en cas d'échec (*backtrack*), et l'ajout

de la tactique XApp à la liste des tactiques connues par Auto aurait pour effet de la faire échouer dans des cas pour lesquels elle fonctionnait jusqu'à présent. Une des solutions possible est d'écrire quand cela est nécessaire, une autre tactique ayant le même squelette qu'Auto, mais paramétrée par une autre liste de tactiques adaptée à une autre classe de problème. Dans un deuxième temps, on voudrait pouvoir faire cela avec une tactique capable d'effectuer un retour sur ses pas. . .

Dans ce cas, nous avons choisi de prouver la totalité "à la main", et nous profitons de ce que la preuve est courte pour la décomposer en donnant les réponses du moteur de preuve.

```
v "Total mul" ;;
```

Le système nous retourne la conjecture à prouver.

```
|-
(Forall z9:nat (Forall z8:nat ((mul z9 z8):nat)))

p "ByInduction" ;;
```

La tactique ByInduction n'a pas su montrer la conjecture, et nous renvoie donc les sous-buts engendrés par l'induction. Prouvons d'abord le cas zéro.

```
[0] proofGoal:

|-
(Forall z8:nat ((mul O z8):nat))

p "Auto" ;;
```

Nous avons montré le cas zéro automatiquement, passons maintenant au cas récursif :

```
[1] proofGoal:

|-
(Forall x7:nat ((Forall z8:nat ((mul x7 z8):nat))
         -> (Forall z8:nat ((mul S(x7) z8):nat))))
```

Nous devons maintenant passer en hypothèse le quantificateur et l'hypothèse de récurrence.

```
p "Intros" ;;
```

```
[1;0] proofGoal:
[H3 : (z10:nat)]
H2 : (Forall z8:nat ((mul x8 z8):nat))
H1 : (x8:nat)
|-
((mul S(x8) z10):nat)
```

L'hypothèse de récurrence est `H2`. Avant de pouvoir l'utiliser, il faut simplifier l'expression. La commande qui suit ouvre une fois la définition de `mul`, puis évalue l'expression sans plus toucher aux définitions.

```
p "OpenAndEval mul" ;;

[1;0;0] proofGoal:
[H3 : (z10:nat)]
H2 : (Forall z8:nat ((mul x8 z8):nat))
H1 : (x8:nat)
|-
((add z10 (mul x8 z10)):nat)
```

La tactique `XApp` décompose une fois l'application dans le jugement de terminaison/typage. Elle engendre deux sous-buts, l'un pour la fonction, l'autre pour son argument. Nous ne pouvons pas encore utiliser `Auto` ici, qui ouvrirait la définition de `mul` et ne saurait pas prouver le sous-but.

```
p "XApp" ;;

[1;0;0;0] proofGoal:
[H3 : (z10:nat)]
H2 : (Forall z8:nat ((mul x8 z8):nat))
H1 : (x8:nat)
|-
((add z10):(nat -> nat))
```

Nous pouvons maintenant appliquer `Auto`, qui utilise alors la totalité de l'addition et l'hypothèse `H3` pour clore cette branche de la preuve.

```
p "Auto" ;;
```

Nous revenons alors à la deuxième branche engendrée par XApp.

```
[1;0;0;1] proofGoal:
[H3 : (z10:nat)]
H2 : (Forall z8:nat ((mul x8 z8):nat))
H1 : (x8:nat)
|-
((mul x8 z10):nat)
```

Il suffit maintenant d'appliquer Auto, qui utilise alors l'hypothèse de récurrence pour conclure la preuve.

```
p "Auto" ;;

Proved
```

Montrons maintenant les lemmes de base concernant la multiplication : $0.x = 0$, $x.0 = 0$, $succ(x).y = y + x.y$ et $x.succ(y) = y + x.y$. Les trois premiers sont montrés automatiquement.

```
v "Theorem mul0x :
        (Forall x : nat '((mul 0 x) = 0))" ;;
p "Auto" ;;

v "Theorem mulx0 :
        (Forall x : nat '((mul x 0) = 0))" ;;
p "ByInduction" ;;


v "Theorem mulSx :
 (Forall x : nat
   (Forall y : nat
      '((mul S(x) y) = (add y (mul x y))))))" ;;
p "Auto" ;;
```

Avant de montrer $x.succ(y) = y + x.y$, nous voulons avoir le petit lemme suivant : $x+(y+z) = y+(x+z)$, qui nous simplifiera l'écriture de la preuve qui suit.

```
v "Theorem lemme_add_1 :
 (Forall x : nat
  (Forall y : nat
   (Forall z : nat
```

```
    ‘((add x (add y z)) = (add y (add x z))))))" ;;
p "ByInduction" ;;
```

Nous pouvons maintenant montrer par induction sur x le dernier lemme de base :

```
v "Theorem mulxS :
 (Forall x : nat
   (Forall y : nat
     ‘((mul x S(y)) = (add x (mul x y)))))" ;;
p "ByInduction" ;;
```

Le cas de base est prouvé automatiquement.

```
p "Auto" ;;
```

Nous devons maintenant montrer le cas récursif. On commence par introduire les hypothèses dans le contexte.

```
p "Intros" ;;
```

Le but à prouver est donc

```
[1;0] proofGoal:
[H3 : (y1:nat)]
H2 : (Forall y:nat
       ‘(((mul x8 S(y))=(add x8 (mul x8 y)))))
H1 : (x8:nat)
|-
‘(((mul S(x8) S(y1))=(add S(x8) (mul S(x8) y1))))
```

Nous devons maintenant réécrire la conclusion du séquent jusqu'à pouvoir appliquer l'hypothèse `H2`.

```
p "AppEq mulSx" ;;
p "AppEq mulSx" ;;
```

La conclusion a maintenant la forme

```
...
|-
‘(((add S(y1) (mul x8 S(y1))) =
   (add S(x8) (add y1 (mul x8 y1)))))
```

On réduit donc les additions, puis on simplifie le successeur dans chacun des termes de l'égalité.

```
p "AppEq addSx" ;;
p "AppEq addSx" ;;
p "EqConstr" ;;

[1;0;0;0;0;0;0] proofGoal:
[H3 : (y1:nat)]
H2 : (Forall y:nat
            '(((mul x8 S(y))=(add x8 (mul x8 y)))))
H1 : (x8:nat)
|-
'(((add y1 (mul x8 S(y1))) =
   (add x8 (add y1 (mul x8 y1)))))
```

Il nous reste plus qu'à utiliser notre hypothèse d'induction.

```
p "AppEq H2" ;;
```

On peut alors appliquer le lemme précédent, et conclure par `Auto`.

```
p "AppEq lemme_add_1" ;;
p "Auto" ;;
```

Avant de pouvoir démontrer la commutativité et l'associativité de la multiplication, il est utile de montrer les propriétés de distribution de l'addition sur la multiplication : $x.(y+z) = x.y + x.z$ et $(y+z).x = y.x + z.x$. Nous prouvons d'abord la distributivité à gauche par induction sur x. Pour la distributivité à droite, nous raisonnons aussi par induction, puis nous appliquons la commutativité de l'addition et la distributivité à gauche.

```
v "Theorem dis_left :
(Forall x : nat (Forall y : nat (Forall z : nat
  '((mul x (add y z)) =
    (add (mul x y) (mul x z))))))" ;;
p "ByInduction" ;;
p "Auto" ;;
```

Le cas de base est prouvé automatiquement.

```
p "Intros" ;;
```

Nous devons prouver le séquent :

```
[1;0;0;0] proofGoal:
[H4 : (z10:nat)]
H3 : (y1:nat)
H2 : (Forall y:nat (Forall z:nat
        '(((mul x8 (add y z))=
           (add (mul x8 y) (mul x8 z))))))
H1 : (x8:nat)
|-
'(((mul S(x8) (add y1 z10)) =
   (add (mul S(x8) y1) (mul S(x8) z10))))
```

Nous procédons maintenant par raisonnement équationnel, jusqu'à pouvoir appliquer l'hypothèse d'induction.

```
p "AppEq mulSx" ;;
p "AppEq mulSx" ;;
p "AppEq mulSx" ;;
p "AppEq add_assoc" ;;
p "AppEq add_assoc" ;;
p "AppEq simp_left_add" ;;
p "AppEq add_comm" ;;
p "AppEq add_assoc" ;;
p "AppEq simp_left_add" ;;
p "AppEq add_comm" ;;
```

Il nous reste à montrer

```
[1;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0;0] proofGoal:
[H4 : (z10:nat)]
H3 : (y1:nat)
H2 : (Forall y:nat (Forall z:nat
       '(((mul x8 (add y z)) =
          (add (mul x8 y) (mul x8 z))))))
H1 : (x8:nat)
|-
'(((mul x8 (add y1 z10)) =
   (add (mul x8 y1) (mul x8 z10))))
```

`Auto` suffit pour conclure.

```
p "Auto" ;;
```

Deux conclusions sont à tirer de la précédente preuve : tout d'abord, cette preuve est faite de façon naturelle, par raisonnement équationnel. On constate cependant qu'il manque à notre système un moteur de réécriture permettant de montrer effciacement et automatiquement ce type de propositions.

Nous donnons sans la décrire la seconde preuve de distributivité, qui n'utilise rien de nouveau.

```
v "Theorem dis_right :
(Forall x : nat (Forall y : nat (Forall z : nat
  '((mul (add y z) x) =
    (add (mul y x) (mul z x))))))" ;;
p "ByInduction" ;;
p "Auto" ;;

p "Intros" ;;
multi "AppEq mulxS" 3 ;;
multi "AppEq add_assoc" 2 ;;
p "AppEq simp_left_add" ;;
p "AppEq add_comm" ;;
p "AppEq add_assoc" ;;
p "AppEq simp_left_add" ;;
p "AppEq add_comm" ;;
p "Auto" ;;
```

Nous pouvons maintenant conclure la partie arithmétique de l'exemple par l'associativité et la commutativité de la multiplication, qui sont maintenant prouvables automatiquement par la tactique `ByInduction`, en utilisant les résultats ci-dessus.

```
v "Theorem mul_assoc :
 (Forall x : nat
   (Forall y : nat
     (Forall z : nat
       '((mul (mul x y) z) = (mul x (mul y z))))))" ;;
p "ByInduction" ;;
(* Proved *)

v "Theorem mul_comm :
 (Forall x : nat
   (Forall y : nat
```

```
      ‘((mul x y) = (mul y x))))" ;;
p "ByInduction" ;;
(* Proved *)
```

6.8.6.2 Opérations sur les booléens

Nous définissons maintenant les opérateurs booléens “et” et “ou”.

```
v "
let or = fun x -> fun y -> (if x then true else y)
" ;;

v "
let and = fun x -> fun y -> (if x then y else false)
" ;;
```

Ces fonctions n’étant pas récursives, montrer la terminaison est trivial.

```
v "Total or" ;;
p "Auto" ;;

v "Total and" ;;
p "Auto" ;;
```

Nous voulons maintenant montrer que le `or` booléen est équivalent au connecteur `Or`.

```
v "Theorem orOr :
(Forall x : bool (Forall y : bool
 (‘((or x y)) -> (‘(x) Or ‘(y))))))" ;;
p "ByInduction" ;;
```

Dans le cas où x est faux, on simplifie l’expression.

```
p "Eval" ;;
p "Intros" ;;
```

On doit prouver :

```
[0;0;0] proofGoal:
[H2 : ‘(y1)]
H1 : (y1:bool)
|-
(‘(false) Or ‘(y1))
```

On choisit alors la branche droite du “ou”, puis on applique `H2`.

```
p "ROr 2" ;;
p "Assume" ;;
```

Dans le cas où x est vrai, on simplifie également.

```
p "Eval" ;;
p "Intros" ;;
```

On obtient :

```
[1;0;0] proofGoal:
[H2 : ‘(true)]
H1 : (y1:bool)
|-
(‘(true) Or ‘(y1))
```

On choisit alors la branche gauche du “ou”, qui est trivialement vraie.

```
p "ROr 1" ;;
p "Auto" ;;
```

Le lemme correspondant, pour le “et” est prouvé automatiquement par induction.

```
v "Theorem andAnd :
(Forall x : bool (Forall y : bool
 (‘((and x y)) -> (‘(x) And ‘(y)))))" ;;
p "ByInduction" ;;
```

La raison de la dyssymmétrie entre ces deux preuves a déjà été évoquée plus haut : notre tactique `Auto` n’effectuant pas de *backtrack*, il aurait été inutile d’y intégrer les règles correspondant au connecteur `Or`. Dans le cas du `And`, en revanche, les règles sont inversibles, et ne nécessitent donc pas de *backtrack* pour être utilisées.

6.8.6.3 Évaluateurs

Les deux parties précédentes nous permettent maintenant d'entrer dans le vif de notre exemple. Nous définissons un langage simple de termes représentant les expressions arithmétiques dans les entiers naturels, restreintes à l'addition et la multiplication. Ces termes sont représentés par le type ML `term`. Nous définissons ensuite la sémantique de ce langage de terme en posant la fonction `eval : term -> nat` calculant la valeur de l'expression.

Nous écrivons une nouvelle fonction d'évaluation "optimisée", `eval0` qui n'évalue le fils droit d'un produit que si le fils gauche est non nul. Nous montrons que cet évaluateur est équivalent à l'évaluateur précédent.

Nous définissons ensuite la fonction
`swap : term -> term -> bool`, retournant `true` lorsque les deux arbres sont égaux à permutation des branches de chaque noeud près. Cette fonction nous permet de définir une relation sur les arbres. Nous montrons ensuite que cette préserve bien la sémantique d'évaluation.

Type `term` et évaluateur
Commençons par définir le type des termes.

```
v "
type term =
   Nat of nat
 | Add of term * term
 | Mul of term * term" ;;
```

La fonction d'évaluation parcours récursivement le terme et utilise l'addition et le produit définis précédemment.

```
v "
let rec eval = fun t ->
 (match t with
   Nat(n)     -> n
 | Add(t1,t2) -> (add (eval t1) (eval t2))
 | Mul(t1,t2) -> (mul (eval t1) (eval t2)))" ;;
```

Montrons la totalité de la fonction `eval`. Nous ne détaillons pas la preuve, celle-ci n'apportant rien de nouveau par rapport aux preuves de totalité précédentes.

```
v "Total eval" ;;
p "ByInduction" ;;
p "Auto" ;;
p "Intros" ;;
p "OpenAndEval eval" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;
p "Auto" ;;
p "Intros" ;;
p "OpenAndEval eval" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;
p "Auto" ;;
```

Nous commençons par poser les deux petits lemmes suivants, aisément prouvés, qui nous permettrons d'éviter l'ouverture de la définition d'`eval`.

```
v "Theorem evalAdd :
 (Forall t1 : term  (Forall t2 : term
   '((eval Add(t1,t2)) =
      (add (eval t1) (eval t2)))))" ;;
p "Eval eval" ;;
p "Auto" ;;

v "Theorem evalMul :
 (Forall t1 : term  (Forall t2 : term
   '((eval Mul(t1,t2)) =
       (mul (eval t1) (eval t2)))))" ;;
p "Eval eval" ;;
p "Auto" ;;
```

Nous aurons besoin ultérieurement des lemmes suivants, exprimant que l'on peut permuter les branches d'une addition et d'une

multiplication. Les preuves font appel à la commutativité de l'addition et du produit, montrées plus haut.

```
v "Theorem addComm :
(Forall t1 : term
 (Forall t2 : term
  '((eval Add(t1,t2))) = (eval Add(t2,t1))))" ;;
p "OpenAndEval eval" ;;
p "Intros" ;;
p "AppEq add_comm" ;;
p "Auto" ;;

v "Theorem mulComm :
(Forall t1 : term
 (Forall t2 : term
  '((eval Mul(t1,t2))) = (eval Mul(t2,t1))))" ;;
p "OpenAndEval eval" ;;
p "Intros" ;;
p "AppEq mul_comm" ;;
p "Use eq_refl" ;;
p "Left H3;[nat]" ;;
p "Auto" ;;
```

Équivalence des évaluateurs

Nous définissons un second évaluateur, raccourcissant le calcul lorsque la branche gauche d'un produit est nulle. L'écriture de cet évaluateur est très proche de ce que l'on aurait écrit en ML. L'intérêt de cet exemple est qu'il se rapproche d'un problème typique de la preuve de programmes : montrer la correction d'une optimisation.

```
v "
let rec eval0 = fun t ->
 (match t with
   Nat(n)     -> n
 | Add(t1,t2) -> (add (eval0 t1) (eval0 t2))
 | Mul(t1,t2) ->
    (let e1 = (eval0 t1) in
      if (e1 = 0) then 0
```

```
      else
       (let e2 = (evalO t2) in
         (if (e2 = O) then O
          else (mul e1 e2)))))" ;;
```

Nous devons maintenant en prouver la totalité. La preuve se fait par induction sur l'arbre parcouru.

```
v "Total evalO" ;;
p "ByInduction" ;;
```

Le cas de base est prouvé automatiquement.

```
p "Auto" ;;
```

Le cas `Add()` est également direct.

```
p "Intros" ;;
p "OpenAndEval evalO" ;;
p "XApp" ;;
p "XApp" ;;
auto 3  ;;
```

Seul le cas `Mul()` est plus subtil.

```
p "Intros" ;;
p "OpenAndEval evalO" ;;
```

Après aménagement, le but à prouver est

```
[2;0;0] proofGoal:
[H4 : ((evalO x19):nat)]
H3 : ((evalO x18):nat)
H2 : (x19:term)
H1 : (x18:term)
|-
((if ((evalO x18)=O)
  then O else
   (if ((evalO x19)=O)
    then O else (mul (evalO x18) (evalO x19)))):nat)
```

Décomposons le `if then else` extérieur (par cas sur la condition).

```
p "BoolIf [z:nat]((z:nat))" ;;
```

Nous obtenons alors deux sous-buts. Dans le premier cas, nous devons montrer que `(O : nat)`.

```
p "Auto" ;;
```

Le deuxième sous-but correspond à la branche `else`.

```
[2;0;0;1] proofGoal:
[H5 : (Not '(((evalO x18)=O)))]
H4 : ((evalO x19):nat)
H3 : ((evalO x18):nat)
H2 : (x19:term)
H1 : (x18:term)
|-
((if ((evalO x19)=O)
  then O
  else (mul (evalO x18) (evalO x19))):nat)
```

Raisonnons maintenant par cas sur la condition de second `if`.

```
p "BoolIf [z:nat]((z:nat))" ;;
```

Le premier sous-but est toujours immédiat.

```
p "Auto" ;;
```

Il nous reste maintenant à conclure en prouvant

```
[2;0;0;1;1] proofGoal:
[H6 : (Not '(((evalO x19)=O)))]
H5 : (Not '(((evalO x18)=O)))
H4 : ((evalO x19):nat)
H3 : ((evalO x18):nat)
H2 : (x19:term)
H1 : (x18:term)
|-
((mul (evalO x18) (evalO x19)):nat)
```

Nous décomposons l'expression avand de conclure par la totalité de `mul` et les hypothèses de récurrence.

```
p "XApp" ;;
p "XApp" ;;
auto 3 ;;
```

Passons maintenant à la preuve qui nous intéresse : la correction de cet évaluateur par rapport au précédent.

```
v "Theorem correc :
   (Forall t : term ‘((evalO t) = (eval t)))" ;;
```

Pour prouver cet énoncé, nous procédons par induction sur `t`, puis, dans le cas de la multiplication, par cas sur le fait que l'arbre de gauche se réduise en 0, et si ce n'est pas le cas, par cas sur le fait que l'arbre de droite se réduise en 0.

```
p "ByInduction" ;;
```

Le cas des feuilles est trivial.

```
p "Auto" ;;
```

Dans le cas où l'arbre est une somme, on commence par évaluer `evalO`.

```
p "Intros" ;;
p "OpenAndEval evalO" ;;

[1;0;0] proofGoal:
[H4 : ‘(((evalO x19)=(eval x19)))]
H3 : ‘(((evalO x18)=(eval x18)))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((add (evalO x18) (evalO x19))=(eval Add(x18,x19))))
```

Il ne reste qu'à utiliser les hypothèses de récurrence, puis à réduire à droite de l'équation pour conclure.

```
p "AppEq H4" ;;
p "AppEq H3" ;;
p "AppEq evalAdd" ;;
p "Auto" ;;
```

Nous arrivons donc au cas de la multiplication.

```
p "Intros" ;;
p "OpenAndEval evalO" ;;
```

Nous devons alors prouver

```
[2;0;0] proofGoal:
[H4 : ‘(((evalO x19)=(eval x19)))]
H3 : ‘(((evalO x18)=(eval x18)))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((if ((evalO x18)=O)
    then O
    else (if ((evalO x19)=O)
          then O
          else (mul (evalO x18) (evalO x19)))) =
               (eval Mul(x18,x19))))
```

Commeçons par ouvrir le `if` extérieur.

```
p "BoolIf [z:nat]‘(z=(eval Mul(x18,x19)))" ;;
```

Dans le cas où le sous-arbre gauche est nul, nous devons montrer

```
[2;0;0;0] proofGoal:
[H5 : ‘(((evalO x18)=O))]
H4 : ‘(((evalO x19)=(eval x19)))
H3 : ‘(((evalO x18)=(eval x18)))
H2 : (x19:term)
H1 : (x18:term)
|-
‘((O=(eval Mul(x18,x19))))
```

On procède alors par raisonnement équationnel (nous ne sommes pas ici dans le cadre de la simple réécriture, on utilise `H3` de droite à gauche).

```
p "AppEq evalMul" ;;
p "AppEqRev H3" ;;
p "AppEq H5" ;;
p "Auto" ;;
```

Dans le cas où le sous-arbre gauche est non nul, on ouvre le second `if`.

```
p "BoolIf [z:nat]‘(z=(eval Mul(x18,x19)))" ;;
```

Nous sommes donc dans le cas où le sous-arbre droit est nul.

```
(...)
|-
‘((O=(eval Mul(x18,x19))))
```

On procède comme plus haut.

```
p "AppEq evalMul" ;;
p "AppEqRev H4" ;;
p "AppEq H6" ;;
p "AppEq mulxO" ;;
p "Auto" ;;
```

Il reste à montrer le cas où les sous-arbres ne sont pas nuls.

```
[2;0;0;1;1] proofGoal:
[H6 : (Not ‘(((evalO x19)=O)))]
H5 : (Not ‘(((evalO x18)=O)))
H4 : ‘(((evalO x19)=(eval x19)))
H3 : ‘(((evalO x18)=(eval x18)))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((mul (evalO x18) (evalO x19))=(eval Mul(x18,x19))))
```

Encore une fois, on procède équationnellement, et on conclut par la réflexivité de l'égalité.

```
p "AppEq H4" ;;
p "AppEq H3" ;;
p "AppEq evalMul" ;;
p "Use eq_refl" ;;
p "Left H7;[nat]" ;;
p "Auto" ;;
```

Préservation de l'évaluation par les permutations

La dernière partie de l'exemple consiste à montrer que l'évaluation de l'arbre est préservée par la permutation des deux fils d'une branche.

Commençons par définir ce qu'est une permutation. La fonction `swap : term -> term -> bool` est définie telle que si `t1` est une feuille, alors `t2` doit être une feuille et porter la même valeur.

Si t1 est de la forme Add(l1,r1) alors t2 doit être de la forme Add(l2,r2) tel que soit l1 est une permutation de l2 et r1 une permutation de r2, soit l1 est une permutation de r2 et r1 une permutation de l2. Le cas de la multiplication est identique.

```
v "
let rec swap = fun t1 -> fun t2 ->
 (match t1 with
   Nat(n1) ->
    (match t2 with
      Nat(n2) -> (n1 = n2)
    | foo -> false)
 | Add(l1,r1) ->
    (match t2 with
      Add(l2,r2) ->
       (or (and (swap l1 l2) (swap r1 r2))
           (and (swap l1 r2) (swap r1 l2)))
    | foo -> false)
 | Mul(l1,r1) ->
    (match t2 with
      Mul(l2,r2) ->
       (or (and (swap l1 l2) (swap r1 r2))
           (and (swap l1 r2) (swap r1 l2)))
    | foo -> false))
" ;;
```

Prouvons d'abord la totalité de swap. Nous donnons la preuve mais ne la décrivons pas, car elle est similaire aux preuves de totalité effectuées précedemment, bien que plus longue (plus de cas). Elle se fait par induction sur t1, puis t2.

```
v "Total swap" ;;
p "ByInduction" ;;
p "Auto" ;;

(* Add *)
intro 4 ;;
p "ByInduction" ;;
p "Auto" ;;

(* Add/Add *)
```

```
p "Intros" ;;
p "OpenAndEval swap" ;;
p "XApp" ;;

p "XApp" ;;
p "Auto" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;

p "Auto" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;
p "Auto" ;;

(* Add/Mul *)
p "Auto" ;;

(* Mul *)
intro 4 ;;
p "ByInduction" ;;
p "Auto" ;;

(* Mul/Add *)
p "Auto" ;;

(* Mul/Mul *)
p "Intros" ;;
p "OpenAndEval swap" ;;
p "XApp" ;;

p "XApp" ;;
p "Auto" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;
```

```
p "Auto" ;;
p "XApp" ;;
p "XApp" ;;
p "Auto" ;;
p "Auto" ;;
p "Auto" ;;
```

Nous utilisons maintenant la fonction `swap` comme une relation, pour énoncer le théorème suivant :

```
v "Theorem swap_eval :
(Forall t1 : term
 (Forall t2 : term
  (‘((swap t1 t2)) -> ‘((eval t1) = (eval t2)))))" ;;
```

Pour montrer ce théorème “à la main” on procèderait de la façon suivante : Par induction sur `t1`, puis sur `t2`.

1. Si l'un des deux termes est de la forme `Nat(n)`, nous concluons aisément.
2. *Idem* si les deux termes n'ont pas la même racine.
3. Si `t1` et `t2` sont de la forme `Add(l1,r1)` et `Add(l2,r2)`, on distingue trois cas :
   (a) `l1` est une permutation de `l2` et `r1` une permutation de `r2`. On conclut alors en utilisant les hypothèses d'induction.
   (b) Si `l1` est une permutation de `r2` et `r1` une permutation de `l2`. On conclut alors en utilisant les hypothèses d'induction et la commutativité de l'addition.
   (c) `t1` n'est pas une permutation de `t2`. On conclut alors par l'absurde.
4. *Idem* si `t1` et `t2` sont de la forme `Mul(l1,r1)` et `Mul(l2,r2)`, *mutatis mutandis*.

L'utilisation d'un système formel nous oblige à respecter une structure plus contrainte et de plus bas niveau. Mais dans la perspective d'une interface d'édition pleine page, l'utilisateur a la possibilité de traiter les buts dans un ordre proche de la preuve informelle. On peut malgré tout remarquer que la preuve formelle suit les mêmes axes que la preuve ci-dessus.

```
p "ByInduction" ;;
```

t1 est une feuille (cas 1), la tactique ByInduction suffit pour conclure.

```
p "Intro" ;;
p "ByInduction" ;;
```

Dans le cas où t1 est une addition, on procède à l'induction sur t2.

```
intro 4 ;;
p "ByInduction" ;;
```

Le cas où t2 est un Nat est résolu automatiquement.

```
p "Auto" ;;
```

Nous sommes maintenant dans le cas où t1 et t2 sont des Add (cas 3).

```
p "Intros" ;;

[1;0;0;0;0;1;0] proofGoal:
[H9 : ‘((swap Add(x18,x19) Add(x26,x27)))]
H8 : (‘((swap Add(x18,x19) x27)) ->
         ‘(((eval Add(x18,x19))=(eval x27))))
H7 : (‘((swap Add(x18,x19) x26)) ->
         ‘(((eval Add(x18,x19))=(eval x26))))
H6 : (x27:term)
H5 : (x26:term)
H4 : (Forall t2:term (‘((swap x19 t2)) ->
         ‘(((eval x19)=(eval t2)))))
H3 : (Forall t2:term (‘((swap x18 t2)) ->
         ‘(((eval x18)=(eval t2)))))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((eval Add(x18,x19))=(eval Add(x26,x27))))
```

Nous commençons par ouvrir la définition de swap dans l'hypothèse H9, puis l'évaluer.

```
p "OpenHyp swap" ;;
p "EvalHyp H10" ;;
```

Nous obtenons l'hypothèse suivante :

```
H11 : ‘((or (and (swap x18 x26) (swap x19 x27))
            (and (swap x18 x27) (swap x19 x26))))
```

Nous voulons modifier cette hypothèse pour qu'elle utilise le connecteur `Or` afin de pouvoir raisonner par cas.

Nous effectons d'abord une coupure sur la formule contenant `Or`.

```
p "Cut (‘((and (swap x18 x26) (swap x19 x27)))
   Or ‘((and (swap x18 x27) (swap x19 x26))))" ;;
```

Nous prouvons ensuite la formule de la coupure.

```
p "Apply orOr" ;;
p "Auto" ;;
```

Il nous reste à passer cette formule dans le contexte.

```
p "Intro" ;;
```

Nous effectuons maintenant la règle gauche du “ou”, correspondant au raisonnement par cas. Cela engendre alors deux sous-buts.

```
p "LOr H12" ;;

[1;0;0;0;0;1;0;0;0;1;0;0] proofGoal:
[H13 : ‘((and (swap x18 x26) (swap x19 x27)))]
(...)
|-
‘(((eval Add(x18,x19))=(eval Add(x26,x27))))
```

L'hypothèse `H13` nous indique que nous sommes dans le cas 3a : il n'y a pas de permutation des branche à cet étage.

Nous procédons de la même manière que pour le `or` pour pouvoir utiliser le `and` comme connecteur, avant de pouvoir scinder l'hypothèse en deux.

```
p "Cut (‘((swap x18 x26)) And ‘((swap x19 x27)))" ;;
p "Apply andAnd" ;;
p "Auto" ;;
p "Intro" ;;
p "LAnd H14" ;;
```

Nous devons donc prouver le séquent suivant :

```
[1;0;0;0;0;1;0;0;0;1;0;0;1;0;0] proofGoal:
[H16 : ‘((swap x19 x27))]
H15 : ‘((swap x18 x26))
H13 : ‘((and (swap x18 x26) (swap x19 x27)))
H11 : ‘((or (and (swap x18 x26) (swap x19 x27))
            (and (swap x18 x27) (swap x19 x26))))
H9 : ‘((swap Add(x18,x19) Add(x26,x27)))
H8 : (‘((swap Add(x18,x19) x27)) ->
      ‘(((eval Add(x18,x19))=(eval x27))))
H7 : (‘((swap Add(x18,x19) x26)) ->
      ‘(((eval Add(x18,x19))=(eval x26))))
H6 : (x27:term)
H5 : (x26:term)
H4 : (Forall t2:term (‘((swap x19 t2)) ->
                      ‘(((eval x19)=(eval t2)))))
H3 : (Forall t2:term (‘((swap x18 t2)) ->
                      ‘(((eval x18)=(eval t2)))))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((eval Add(x18,x19))=(eval Add(x26,x27))))

p "OpenAndEval eval" ;;
```

L’évaluation nous amène à prouver (nous avons supprimé les hypothèses inutiles) :

```
(...)
H4 : (Forall t2:term (‘((swap x19 t2)) ->
                      ‘(((eval x19)=(eval t2)))))
H3 : (Forall t2:term (‘((swap x18 t2)) ->
                      ‘(((eval x18)=(eval t2)))))
|-
‘(((add (eval x18) (eval x19)) =
   (add (eval x26) (eval x27))))
```

Il nous reste plus qu'à instancier les hypothèses d'induction, puis à les utiliser comme égalités.

```
p "Left H3;x26" ;;
p "Left H4;x27" ;;
p "AppEq H17" ;;
p "AppEq H18" ;;
```

Les sous-buts résiduels ont étés éliminés automatiquement par `AppEq`. Il nous reste alors à montrer :

```
(...)
|-
‘(((add (eval x26) (eval x27)) =
   (add (eval x26) (eval x27))))

p "Auto" ;;
```

`Auto` conclut cette branche de la preuve.

Nous nous retrouvons maintenant dans le second cas de figure (3b).

```
[H13 : ‘((and (swap x18 x27) (swap x19 x26)))]
H11 : ‘((or (and (swap x18 x26) (swap x19 x27))
            (and (swap x18 x27) (swap x19 x26))))
H9 : ‘((swap Add(x18,x19) Add(x26,x27)))
H8 : (‘((swap Add(x18,x19) x27)) ->
      ‘(((eval Add(x18,x19))=(eval x27))))
H7 : (‘((swap Add(x18,x19) x26)) ->
      ‘(((eval Add(x18,x19))=(eval x26))))
H6 : (x27:term)
H5 : (x26:term)
H4 : (Forall t2:term (‘((swap x19 t2)) ->
                      ‘(((eval x19)=(eval t2)))))
H3 : (Forall t2:term (‘((swap x18 t2)) ->
                      ‘(((eval x18)=(eval t2)))))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((eval Add(x18,x19))=(eval Add(x26,x27))))
```

Les branches du noeud courant ont été permutées.

On procède encore de la même façon pour utiliser le `and`.

```
p "Cut (‘((swap x18 x27)) And ‘((swap x19 x26)))" ;;
p "Apply andAnd" ;;
p "Auto" ;;
p "Intro" ;;
p "LAnd H14" ;;
```

Nous devons maintenant prouver le séquent suivant :

```
[1;0;0;0;0;1;0;0;0;1;0;1;1;0;0] proofGoal:
[H16 : ‘((swap x19 x26))]
H15 : ‘((swap x18 x27))
H13 : ‘((and (swap x18 x27) (swap x19 x26)))
H11 : ‘((or (and (swap x18 x26) (swap x19 x27))
            (and (swap x18 x27) (swap x19 x26))))
H9 : ‘((swap Add(x18,x19) Add(x26,x27)))
H8 : (‘((swap Add(x18,x19) x27)) ->
      ‘(((eval Add(x18,x19))=(eval x27))))
H7 : (‘((swap Add(x18,x19) x26)) ->
      ‘(((eval Add(x18,x19))=(eval x26))))
H6 : (x27:term)
H5 : (x26:term)
H4 : (Forall t2:term (‘((swap x19 t2)) ->
                      ‘(((eval x19)=(eval t2)))))
H3 : (Forall t2:term (‘((swap x18 t2)) ->
                      ‘(((eval x18)=(eval t2)))))
H2 : (x19:term)
H1 : (x18:term)
|-
‘(((eval Add(x18,x19))=(eval Add(x26,x27))))
```

```
p "OpenAndEval eval" ;;
p "Left H3;x27" ;;
p "Left H4;x26" ;;
p "AppEq H18" ;;
p "AppEq H17" ;;
```

Après avoir procédé exactement de la même manière que précédemment, on se retrouve avec le but suivant :

```
(...)
|-
‘(((add (eval x27) (eval x26)) =
   (add (eval x26) (eval x27))))
```

Il nous reste plus alors qu'à faire commuter l'une des additions, et à conclure.

```
p "AppEq add_comm" ;;
p "Auto" ;;
```

Le cas suivant est celui où `t2` est un produit, et il est résolu automatiquement.

```
p "Auto" ;;
```

Reste à montrer le cas de la multiplication, qui est effectué de la même manière, si ce n'est que l'on est obligé d'appliquer explicitement la réflexivité de l'égalité avant d'utiliser `Auto`, sinon celle-ci ouvre la définition de la multiplication et échoue.

```
intro 4 ;;
p "ByInduction" ;;
(* Mul/Nat *)
p "Auto" ;;
(* Mul/Add *)
p "Auto" ;;
(* Mul/Mul *)

p "Intros" ;;
p "OpenHyp swap" ;;
p "EvalHyp H10" ;;
p "Cut (‘((and (swap x18 x26) (swap x19 x27)))
   Or ‘((and (swap x18 x27) (swap x19 x26))))" ;;
p "Apply orOr" ;;
p "Auto" ;;
p "Intro" ;;
p "LOr H12" ;;
 (* cas (and (swap x18 x26) (swap x19 x27)) *)
p "Cut (‘((swap x18 x26)) And ‘((swap x19 x27)))" ;;
p "Apply andAnd" ;;
p "Auto" ;;
p "Intro" ;;
p "LAnd H14" ;;
p "OpenAndEval eval" ;;
p "Left H3;x26" ;;
p "Left H4;x27" ;;
```

```
p "AppEq H17" ;;
p "AppEq H18" ;;
p "Use eq_refl" ;;
p "Left H19;[nat]" ;;
p "Apply H20" ;;
(* cas ((and (swap x18 x27) (swap x19 x26))) *)
p "Cut (‘((swap x18 x27)) And ‘((swap x19 x26)))" ;;
p "Apply andAnd" ;;
p "Auto" ;;
p "Intro" ;;
p "LAnd H14" ;;
p "OpenAndEval eval" ;;
p "Left H3;x27" ;;
p "Left H4;x26" ;;
p "AppEq H18" ;;
p "AppEq H17" ;;
p "AppEq mul_comm" ;;
p "Use eq_refl" ;;
p "Left H19;[nat]" ;;
p "Apply H20" ;;
(* Proved *)
```

6.9 Conclusion sur les exemples

Ces deux exemples nous ont permis de nous convaincre que, bien qu'encore à l'état de prototype, notre système est déjà utilisable.

Dans la partie arithmétique, même si l'automatisation reste très limitée, nos tactiques simples donnent déjà de bons résultats. Il reste cependant beaucoup à améliorer : la tactique `Auto` est paramétrable par une liste de tactiques, mais ne sait cependant pas revenir en arrière. Il manque également un moteur efficace de réécriture, et ne parlons pas de procédures de décision sur des fragments de l'arithmétique !

Dans les parties plus algorithmique, notre système se comporte “correctement”, les preuves n'étant peut-être pas simple, mais elles ne sont en revanches pas compliquées ni obscurcies par un surcoût de codage dans le formalisme de PAF!. Enfin, dans les ma-

nipulations sur des arbres, même subtiles, les preuves suivent le schéma de la preuve telle qu'on l'aurait effectuée à la main. Encore une fois, le formalisme (que cela soit celui de ML ou celui de PAF!) est particulièrement adapté à cette classe de problèmes. Nous pensons donc avoir atteint un des objectifs que nous nous étions fixé : pouvoir raisonner formellement sur des programmes fonctionnels définis par filtrage sur les types algébriques.

Conclusion

Nous avons vu dans le chapitre précédent deux exemples d'utilisation de la version actuelle de notre système. Bien que les tactiques existantes soient encore de bas niveau et que le manque d'automatisation se fasse sentir, ce système propose des solutions originales.

Tout d'abord, nous fournissons une base extensible pour la preuve de programmes ML. L'autre solution existant pour travailler sur ce langage est Coq, mais celui-ci est résolument tourné vers l'extraction de programme, tandis que nous souhaitions pouvoir manipuler directement des termes du langage cible. Notre système se situe donc plus du côté de ACL2, de Kaufmann et Moore.

Des systèmes comme celui de J.-C. Filliâtre [27] ou C. Parent [47] permettent la preuve de programmes en Coq sans recourir à l'extraction, mais cela se fait à travers un codage dans le Calcul des Construction, tandis que dans notre système, l'utilisateur manipule directement les constructions de ML.

Nous comparer à ACL2, ne semble pas à notre avantage, de prime abord. La stratégie de recherche de preuve de ce système est très efficace et notre système ne rivalise pas dans ce domaine. Mais l'architecture que nous avons choisie devrait nous permettre à terme d'implémenter des stratégies similaires dans PAF!, sans pour autant avoir les inconvénients que l'on constate dans ACL2 lorsque sa procédure échoue (manque d'interactivité, difficulté de la lecture de la trace de la preuve pour comprendre ce qui s'est passé), mais avec comme avantage une extensibilité plus importante et le gain en fiabilité offert par l'extraction et la vérification des preuves atomiques. La création d'une procédure de décision telle que celle-là sort cependant du cadre de cette thèse. Il semble également raisonnable de dire que nqthm (l'ancêtre d'ACL2) est dé-

veloppé depuis plus de vingt ans, tandis que notre système est encore jeune. . .

La première chose que nous apportons à la communauté des systèmes d'aide à la preuve est l'architecture originale de notre logiciel, qui tire profit de ce qu'apporte la programmation objet (héritage, liaison tardive, sous-typage) pour simplifier le passage progressif d'une preuve utilisant des tactiques complexes à une preuve *atomique*, ne reposant que sur des étapes primitives. Cela simplifie nettement l'écriture de tactiques, donc, à terme, permet d'augmenter la richesse de notre système. De plus, cette architecture autorise un degré d'interaction beaucoup plus élevé entre le système et l'utilisateur que dans la plupart des autres AP, basés sur des interfaces en ligne de commandes.

L'autre apport de ce travail est une description en profondeur d'un système de preuve, des spécifications aux détails de l'implémentation. Cette description est utile sur plusieurs plans, elle décrit précisément des parties du système pouvant être adaptées sur des AP voués à d'autres finalités. Elle spécifie l'interface sans pour autant figer sa présentation, et elle fournit un protocole susceptible d'être implémenté dans d'autres systèmes, ce qui donnerait la possibilité d'interchanger moteurs de preuves ou interfaces librement.

Ce travail constitue également un exemple de développement riche et complexe mêlant programmation fonctionnelle, impérative et objet (un autre exemple est le projet FOC [10], ayant pour but de créer un environnement de calcul formel certifié).

* * *

Un certain nombre de perspectives s'offrent dans la continuité de ces travaux. Comme on l'a vu, il est nécessaire d'augmenter le degré d'automatisation du système, ainsi que des procédures de décision ou de simplification dans l'arithmétique.

Il serait également intéressant d'étendre miML, de façon à avoir accès aux exceptions et aux aspects impératifs de ML. L'enrichissement de la syntaxe de miML de manière à la rendre réellement compatible avec *Objective Caml* est un objectif à cours terme.

Du point de vue de la théorie, les travaux de fondation de notre système restent encore à formaliser.

Ces travaux s'inscrivent également dans un projet plus large

au sein de l'équipe PPS, MathOS, un AP pour les mathématiques usuelles, pour lequel des éléments de l'architecture ou du protocole définis pour PAF! pourront être utilisés.

Annexe A

Introduction de l'implication

```
open LinkTactics
open API.ProofObject
open API.Formula
open API.ProofRules
open API.PsParam
open API.BasicTypes
open API.TermML
open API.Tables

let tacticName = "Intro. Implication"
let keyword = "IntroImpl"
class intro id goal param =
object (self : 'a)
  inherit (tacticStep tacticName id goal) as super

  (* generate subgoals *)
  method generate =
    try
      let currFormula = goal#activeConcl
      and newGoal = goal#clone
      in
      newGoal#subActiveConcl ;
      (* Is it an arrow *)
      begin
        try
          let f1,f2 = delImplies currFormula
          in
```

```
          newGoal#addActiveHyp f1 ;
          newGoal#addActiveConcl f2
        with Wrong_symbol -> raise WrongRule
      end ;
      self#initGoal [newGoal] ;
      true
    with WrongRule -> false


  method refineFun premTacL =
          let prem =
            match premTacL with
                [x] -> x
              | _ ->
                failwith "forallIntro/refineFun/prem"
          in
          let (a,b) = delImplies concl#activeConcl
          in
          let i1 =
            createNodeWithFormList "LeftArrow1"
                                    goal [a;b]
          in
          let i2 =
            createNodeWithFormList "LeftArrow2"
                             (getNthConcl i1 1) [a;b]
          in
          let ax1 =
            createNodeWithoutParam "Axiom"
                                   (getNthConcl i1 0)
          in
          let ax2 =
            createNodeWithoutParam "Axiom"
                                   (getNthConcl i2 0)
          in
          i1#setPremisses [|ax1;i2|] ;
          i2#setPremisses [|ax2;prem|] ;
          i1
end

let _ = LinkTactics.register keyword (new intro)
```

Bibliographie

[1] Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proof and Programs, International Workshop, TYPES '99*, volume 1956 of *LNCS*. Springer-Verlag, 2000.

[2] Jean-Raymond Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge Univ Press, 1996.

[3] David Aspinall, H. Goguen, T. Kleymann, and D. Sequeira. Proof general manual, 2001. `http://www.proofgeneral.org/~proofgen/`.

[4] S. Baro and P. Manoury. Un système X. Raisonner formellement sur les programmes ML. In *Journées Francophones des Langages Applicatifs*, 2003.

[5] Sylvain Baro. Une plate forme logique de spécification et de preuve de programmes ML. Technical report, PPS, 2002. `http://www.pps.jussieu.fr/PPS-prepub/index.html`.

[6] Stefano Berardi. Pruning simply typed lambda-terms. Technical report, Turin University, 1993.

[7] Yves Bertot. The CtCoq System : Design and Architecture. Technical Report RR-3540, INRIA, octobre 1998.

[8] Yves Bertot, Thomas Kleymann-Schreiber, and Dilip Sequeira. Implementing Proof by Pointing without a Structure Editor. Technical Report RR-3286, INRIA, octobre 1997.

[9] Richard Bornat and Bernard Sufrin. Jape's quiet interface. In *User Interfaces for Theorem Provers*, 1996.

[10] Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Polymorphic data types, objects, modules and functors : is it too much ? Technical report, LIP6, 2000.

[11] R. Boyer and J Moore. Proof-checking, theorem-proving and program verification. Technical report, Institute for Computing Science and Computer Applications, University of Texas, Austin, January 1983.

[12] R. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

[13] Bishop Brock, Matt Kaufmann, and J Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Proceedings of Formal Methods in Computer-Aided Design*, pages 275–293. Springer-Verlag, 1996.

[14] De Bruijn. A survey of the project Automath. In J.R. Hindley J.P. Seldin, editor, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and formalisms*, 1980.

[15] E. Chailloux. Cast objet pour la construction de hiérarchies de classes en objective caml. In *Journées Francophones des langages applicatifs*, 2002.

[16] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. Éditions O'Reilly, 2000.

[17] Catarina Coquand. *Agda*, 2000. `http://www.cs.chalmers.se/~catarina/agda/`.

[18] Thierry Coquand. *Une théorie des constructions*. PhD thesis, Paris 7, 1985.

[19] Philippe Curmin. *Marquage des preuves et extraction de programmes*. PhD thesis, Paris 7, 1999.

[20] H.B. Curry. Functionality in combinatorial logic. In *Proceedings of the National Academy of Sciences*, 1934.

[21] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Principles of Programming Languages*, ACM, 1982.

[22] N.G de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In *Proceedings of the Workshop on Programming Logic, Marstrand, Sweden*, 1987.

[23] David Delahaye. A tactic language for the system Coq. In *Proceedings of Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS/LNAI*. Springer-Verlag, 2000.

[24] Joëlle Despeyroux. Sémantique Naturelle : Spécifications et Preuves. Technical Report RR-3359, INRIA, 1998.

[25] E. W. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[26] DOD-STD-2167A : Defense System Software Engineering Environment. U.S. Department of Defense, Washington D.C., 1988.

[27] J.C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris XI, 1999.

[28] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF : A mechanised logic of computation. volume 78 of *LNCS*. Springer-Verlag, 1979.

[29] Thomas Hallgren. *Alfa*, 2001. `http://www.cs.chalmers.se/~hallgren/Alfa/`.

[30] C. A .R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[31] W.A. Howard. The formulae-as-types notion of construction. In J.R. Hindley J.P. Seldin, editor, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and formalisms*, 1980.

[32] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML : A gentle introduction. *Theoretical Computer Science*, 173(2) : 445–484, 1997.

[33] Matt Kaufmann and J Strother Moore. A Precise Description of the ACL2 Logic. Technical report, Computational Logic, Inc., 1997.

[34] Jean-Louis Krivine. *Lambda-Calcul types et modèles*. Masson, 1990.

[35] Jean-Louis Krivine. *Théorie des Ensembles*. Cassini, 1998.

[36] Jean-Louis Krivine and Michel Parigot. Programming with proofs. In *Journal of Information Processing and Cybernetics*, 1990.

[37] Yves Legrandgérard. Proof Engine Protocol, version 1 (pepv1) specification. manuscrit.

[38] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type discipline. In *Foundations of Computer Science*, pages pp. 460–469, 1983.

[39] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, 1990.

[40] Xavier Leroy. *The Caml Light System*. INRIA, 0.73 edition, 1997.

[41] Xavier Leroy. *The Objective Caml system, documentation and user's guide*. INRIA, 3.04 edition, 2001. `http://pauillac.inria.fr/ocaml/htmlman/`.

[42] Nestor Lopez. *Spécification de systèmes complexes : méthodes et techniques*. PhD thesis, CNAM – CEDRIC, 2002.

[43] Pascal Manoury. Preuve de correction de programmes fonctionnels de tris dans le système Coq. Technical Report 96/22, LITP, Juin 1996.

[44] Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.

[45] C. Morgan. The specification statement. *ACM Trasactions on Programming Languages and Systems*, 1988.

[46] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. SRI International, `http://pvs.csl.sri.com`, v2.4 edition, 2001.

[47] Catherine Parent. Developing certified programs in the system Coq - the Program tactic. In *Types for Proofs and Programs*, volume 806 of *LNCS*, 1993.

[48] Catherine Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In *Mathematics for Programs Constructions*, volume 947 of *LNCS*, 1995.

[49] Michel Parigot. $\lambda\mu$-calculus : an algorithmic interpretation of classical natural deduction. In Springer, editor, *Lecture Notes in Computer Science 624*, 1992.

[50] Michel Parigot. Recursive programming with proofs. In *Theoretical Computer Science*, pages 335–356, 1992.

[51] Michel Parigot. Church-Rosser property in classical free deduction. In G. Huet and G. Plotkin, editors, *Logical Environments*. 1993.

[52] Lawrence C. Paulson. *Introduction to Isabelle*. Computer Laboratory, University of Cambridge, 2002. `http://isabelle.in.tum.de`.

[53] Gordon Plotkin. LCF considered as a programming language. In *Theoretical Computer Science*, number 5, pages 223–256, 1977.

[54] Randy Pollack. *The LEGO Proof Assistant*, 1998. `http://www.dcs.ed.ac.uk/home/lego/`.

[55] Christophe Raffalli. The PhoX proof assistant, 2002. `http://www.lama.univ-savoie.fr/sitelama/Membres/pages_web/RAFFALLI/phox.html`.

[56] Christophe Raffalli and René David. Apprentissage du raisonnement assisté par ordinateur. prépublication 01-09c, LAMA, université de Savoie, 2001.

[57] Eric Raymond and al. The jargon file. `http://www.tuxedo.org/~esr/jargon/html/`, Novembre 2000. version 4.2.3.

[58] Didier Rémy. Typage et programmation. Notes de cours, 1998.

[59] Didier Rémy and Jérôme Vouillon. Objective ML : A simple object-oriented extension of ML. In *Proc. of the 24th ACM Conference on Principles of Programming Languages*, 1997.

[60] J. M. Spivey. *The Z notation : a reference manual*. International Series in Computer Science. Prentice Hall, 1989.

[61] W. W. Tait. Intentional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32, 1967.

[62] LogiCal Project The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 1999-2001. v7.2 `http://coq.inria.fr`.

[63] Laurent Thery, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. Technical Report RR-1684, INRIA, 1992.

[64] Kumar Neeraj Verma and Jean Goubault-Larrecq. Reflecting BDDs in Coq. Technical Report RR-3859, INRIA, 2000.

[65] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, mai 1994.

[66] Freek Wiedijk. The De Bruijn factor. non publié, disponible à `http://citeseer.nj.nec.com/466229.html`.

[67] Xemacs. `http://www.xemacs.org`.

Index

Résumé

Pouvoir vérifier la conformité d'un programme avec sa spécification représente un enjeu important. Pouvoir le faire à l'aide d'une machine offre en plus la possibilité de *vérifier cette vérification*. On utilise pour cela un *assistant de preuve* permettant la description du problème, la construction des preuves et leur vérification.

Plusieurs approches existent pour parvenir à cette fin : engendrer le programme à partir de sa spécification ; utiliser un programme annoté avec sa spécification et des éléments de preuves pour construire la preuve complète de correction ; ou encore partir de la spécification et du programme, puis montrer la conformité du second vis-à-vis du premier. C'est cette dernière approche que nous avons choisi.

Dans le système que nous avons implémenté, l'utilisateur décrit la spécification du programme dans un formalisme logique *ad hoc*, puis donne le programme dans le sous-ensemble fonctionnel de ML (comprenant filtrage, définitions récursives et fonctions partielles), puis construit interactivement les preuves de correction nécessaires pour prouver la validité du programme.

Le travail que nous avons fourni se découpe en trois volets :

- la formalisation d'une logique dédiée à la preuve de programmes dans le fragment fonctionnel de ML ;
- la spécification détaillée du moteur de preuve, de son interface, et du protocole qu'ils utilisent pour communiquer ;
- l'implémentation du moteur de preuve en *Objective Caml*, qui utilise une architecture particulièrement originale, mélangeant programmation objet et programmation fonctionnelle.

Tous ces éléments sont présents dans ce document, y compris une description détaillée de l'implémentation, et des raisons qui nous ont poussés à effectuer certains choix, plutôt que d'autres. Le lecteur y trouvera une description de l'utilisation de notre système, ainsi que des exemples de mise en œuvre de celui-ci.