THÈSE

présentée au

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

pour obtenir le titre de

Docteur en Sciences - Spécialité Informatique

par

Olivier PONS

Titre de la thèse :

CONCEPTION ET RÉALISATION D'OUTILS D'AIDE AU DÉVELOPPEMENT DE GROSSES THÉORIES DANS LES SYSTÈMES DE PREUVE INTERACTIFS

Soutenue le 13 Juillet 1999 au Conservatoire National des Arts et Metiers devant le jury composé de :

M.	**Yves LEDRU**	Président
M.	**Gilles DOWEK**	Rapporteur
M.	**Jean GOUBAULT-LARRECQ**	Rapporteur
Mme.	**Véronique DONZEAU-GOUGE**	Directeur
M.	**Yves BERTOT**	Examinateur
M.	**Pierre CASTERAN**	Examinateur
M.	**Jean Francois MONIN**	Examinateur
Mme.	**Laurence RIDEAU**	Examinateur

Table des matières

1 Présentation générale **1**
1.1 Introduction . . . 1
1.2 Du génie logiciel à l'ingénierie de preuve . . . 2
1.2.1 Preuve et programme . . . 3
1.3 Cadre de ce travail . . . 4
1.3.1 Logique et preuves formelles . . . 4
1.3.2 Quelques critères importants . . . 5
1.3.3 Présentation de quelques systèmes de preuves . . . 7
HOL . . . 7
Isabelle . . . 8
Lego . . . 10
Nuprl . . . 11
Alf . . . 11
PVS . . . 11
1.3.4 Quelque autres systèmes . . . 12
1.4 Remarques générales . . . 12

2 Présentation détaillée de Coq et de l'environnement CtCoq **15**
2.1 Présentation de Coq . . . 15
2.1.1 Terme de preuve versus script de preuve. . . . 15
2.1.2 Structure du script. . . . 18
2.1.3 Présentation de l'environnement de preuve CtCoq . . . 20
Point de vue de l'utilisateur . . . 20
Point de vue du concepteur . . . 22

I Etude locale des théorèmes **25**

3 Compréhension et modification de preuves. **27**
3.1 Un modèle d'arbre de preuve . . . 28
3.2 Visualisation de l'arbre et navigation dans le script . . . 31
Visualisation graphique de l'arbre de preuve . . . 32
Visualisation textuelle . . . 35

3.2.1 La relation sous-but/tactique . . . 38
3.3 Mécanisme de retour-arrière . . . 39
3.3.1 La commande "Undo-Redo" . . . 40
3.3.2 Problèmes de cohérence . . . 41
3.3.3 Fonctionnement du retour arrière logique. . . . 42
3.4 Aperçu de l'implémentation . . . 42
3.4.1 Instrumentation du script . . . 42
3.4.2 Interaction avec le prouveur . . . 44
3.4.3 Gestion de la mémoire . . . 44
3.5 Arbre de preuve et variables existentielles . . . 45
3.5.1 Pourquoi utiliser des variables existentielles? . . . 45
3.5.2 Manipulation des variables existentielles en Coq . . . 47
3.5.3 Influence des variables existentielles sur l'indépendance des branches de l'arbre de preuve . . . 47
3.5.4 Interaction avec le retour-arrière . . . 49
3.6 Solution du problème . . . 51
3.6.1 Principe de base . . . 51
3.6.2 Extension du modèle d'arbre de preuve . . . 52
3.7 Idée de l'implémentation . . . 58
3.7.1 Conclusion sur les variables existentielles . . . 59
3.8 Autres études du mécanisme de retour-arrière . . . 59

4 Contraction et expansion de l'arbre de tactiques 61
4.1 Introduction . . . 61
4.2 Contraction . . . 64
4.2.1 Principe de fonctionnement . . . 64
4.2.2 Variantes et réflexions . . . 64
4.3 Expansion . . . 68
4.3.1 Principe de fonctionnement . . . 68
4.3.2 La sécurisation du nommage des hypothèses . . . 71
4.3.3 La détection de code mort . . . 72
4.3.4 La localisation d'erreurs . . . 73
4.4 Conclusion . . . 73

5 Lemmification et factorisation de lemmes 77
5.1 Lemmification . . . 77
5.1.1 Principe de fonctionnement . . . 79
5.1.2 Idée de l'implémentation dans Coq et CtCoq . . . 80
5.1.3 Faiblesse de notre approche . . . 81
Problème de nommage des hypothèses . . . 81
Autres problèmes . . . 84
5.2 Factorisation de lemme . . . 84
5.2.1 Principe de fonctionnement . . . 88
La tactique Similar . . . 89
5.3 Conclusion . . . 90

II Etude globale des théorèmes **91**

6 Calcul et applications des dépendances au niveau des théories **97**
6.1 Introduction . . . 97
6.2 Modèle abstrait de théorie . . . 97
6.3 Manipulation d'environnements . . . 100
6.4 Implémentation du calcul des dépendances entre objets d'une théorie . . . 100
6.4.1 Implémentation dans Coq . . . 102
6.4.2 Mode interactif . . . 103
6.4.3 Mode batch . . . 104
6.4.4 Implémentation des graphes . . . 104
6.5 Brève présentation des principales applications réalisées en utilisant le graphe de dépendance . . . 104
6.5.1 Visualisation et interface . . . 105
Le Reset Logique: . . . 105
6.5.2 La coupe (ou épuration) de théorie . . . 106
6.5.3 Modification de théorie . . . 107
6.5.4 Réorganisation de développement . . . 107
6.6 Dépendances entre théories et autres outils de visualisation . . . 108

7 La coupe de théorie **113**
7.1 Implémentation . . . 113
7.1.1 Mécanisme de coupe de théorie . . . 113
7.2 Insuffisance des dépendances calculées sur l'objet preuve. . . . 114
7.2.1 Une première solution...inadéquate . . . 115
7.2.2 Calcul sur un terme non normalisé . . . 117
7.3 Réorganisation du script de preuve et suppression de code mort . . . 117

8 Une méthode interactive de propagation des modifications **119**
8.1 Graphe de dépendance et ordre des modifications . . . 119
8.2 Méthode générale . . . 120
8.2.1 Principe de propagation des modifications . . . 121
8.2.2 Exemple . . . 122
8.2.3 Propriétés . . . 124
8.2.4 Remarques . . . 124
8.2.5 Propriétés . . . 124
8.2.6 Preuves . . . 124
8.2.7 Stockage en cours de modification . . . 125
8.3 Spécificités des dépendances entre théorèmes . . . 125
8.3.1 Dépendances opaques et dépendances transparentes . . . 125
Une approximation de la nature des dépendances logiques . . . 127
8.3.2 Adaptation de la méthode de propagation des dépendances . . . 128
8.4 Mise en œuvre . . . 130
8.4.1 Résumé du principe d'utilisation . . . 130

9 Problèmes liés aux procédures de décision automatique **133**
9.1 Présentation du problème . . . 133

9.2 Minimiser les problème liés aux procédures de décision automatique 138
9.3 Une aide à la modification de théorèmes 139

10 Généralisation et réutilisation de preuves 141
10.1 Introduction 141
10.2 Le principe de base sur un exemple 142
10.3 Un outil d'aide à la généralisation 144
10.3.1 Principe de fonctionnement 144
10.3.2 Choix de nom des identificateurs abstraits 145
10.4 Où l'on abstrait encore... 145
10.5 Limitations 146
10.6 Robustesse et problèmes 149
10.7 Restriction dans l'utilisation de l'abstraction. 149
10.8 Conclusion et perspectives 150

11 Conclusion 151

A Exemple de sessions de preuve dans différents systèmes 155
A.1 HOL 155
A.2 Isabelle 159
A.3 Lego 160
A.4 PVS 161
A.5 Alf 163

B Une interface utilisant daVinci et Ocaml 167
B.1 Interfacer daVinci et Caml 167
Le visualisateur daVinci 167
B.1.1 Une idée de l'implémentation 167
Représentation des graphes 168
Communication daVinci-Ocaml 168
Utilisation du package. 170
B.2 Application 170
B.2.1 Un mini éditeur de graphe 170
B.2.2 Affichage de dépendances entre fichiers. 171
B.2.3 Une interface graphique pour *Coq* 171
B.2.4 Conclusion 172

C Exemple d'utilisation de la tactique Similar 173

D Preuves et exemples divers 175
D.1 Exemple d'utilisation de l'algorithme d'expansion 175
D.2 Proprietés de la fonction *rg'* 176

Table des figures

2.2 Un script de preuve de l'associativité à gauche de l'addition 17
2.3 Une étape de construction de la preuve 18
2.4 Autre script de preuve de l'associativité à gauche de l'addition 18
2.5 L'arbre de preuve de l'associativité à gauche de l'addition 19
2.6 Version plus compacte du script de preuve de l'associativité à gauche de l'addition 19
2.7 L'arbre de preuve compacté 20
2.1 Descriptif des tactiques les plus utilisées dans ce document 23
2.8 Environnement de travail avec CtCoq 24

3.1 Représentation graphique de l'arbre de tactique d'une preuve incomplète . . 33
3.2 Représentation graphique de l'arbre de tactique d'une preuve complète . . . 33
3.3 Un script de preuve de l'associativité à gauche de l'addition 35
3.4 Un script indenté "en peigne" 36
3.5 Script "restructuré" 36
3.6 Exemple de navigation dans un script 37
3.7 Liaison tactique/sous-but 38
3.8 Relation entre les fenêtres d'édition et d'état 39
3.9 Après un undo classique 40
3.10 Un script après application de la commande "Undo-Redo" 40
3.11 Un script devenu incohérent 42
3.12 L'environement CtCoq après un Undo-Redo 43
3.13 Le lemme1 48
3.14 L'arbre de preuve après application du théorème de transitivité de l'égalité . 48
3.15 L'arbre de preuve du lemme 1 49
3.16 L'arbre de preuve du lemme 2 54
3.17 L'arbre de preuve du lemme 3 57

4.1 Décomposition d'un nœud de l'arbre 63
4.2 Résultat de la contraction du script expansé correspondant à l'arbre de la figure 4.1. 66
4.3 Algorithme de contraction avec factorisation 67
4.4 Corps de l'algorithme d'expansion 70

4.5 Corps de l'algorithme d'expansion gérant les erreurs 74

5.1 Illustration du principe de factorisation 89

6.1 Graphe de dépendance d'une partie de la théorie des listes 103
6.2 Dépendance entre objets de la théorie des listes au format Make 105
6.3 Dépendances du développement de preuve de l'algorithme de Buchberger . . 109
6.4 Dépendance du développements de preuve d'un mini compilateur 110
6.5 Le graphe de coercion de la théorie des anneaux 111

8.1 Le graphe de départ 120
8.2 Début de traitement après modification de A et B 122
8.3 Après avoir propagé la modification au nœud D 122
8.4 Après avoir propagé la modification au nœud F 123
8.5 Après avoir propagé la modification au nœud E 123
8.6 Après avoir propagé la modification au nœud G 124
8.7 Optimisation de la propagation des modifications 129
8.8 Exemple de développement sur plusieurs fichiers 131

A.1 Environnent de travail du système PVS 164
A.2 Environnent de travail du système Alf 165

B.1 Communication `daVinci-Ocaml` 169
B.2 L'interface Coq/daVinci 172

Chapitre 1

Présentation générale

1.1 Introduction

Dès les années 1950, les pionniers de ce qui allait s'appeler par la suite l'Intelligence Artificielle tentèrent d'utiliser les machines pour automatiser le raisonnement et les preuves mathématiques [1]. Depuis les balbutiements des travaux de Newell, Shaw et Simon [88, 89], dans lesquels ils prouvèrent 38 des 52 premiers théorèmes du chapitre 2 des Principia Mathematica [143] de Russell et Whitehead, les systèmes se sont multipliés et énormément développés. Cela débouche aujourd'hui sur de puissants systèmes automatiques comme Gandalf [133], LeanTAP [11, 10], METEOR [9], Otter [85] [2], SATURATE [53], ou SETHEO [54, 72, 122] [3].

D'un autre côté dans les années 70, face aux difficultés du tout automatique (liées aux problèmes d'indécidabilité) apparurent les systèmes dits "d'aide à la preuve", qui nous intéressent plus particulièrement, et dans lesquels l'utilisateur guide la démonstration. Les pionniers en la matière furent sans conteste Automath [33, 34] de N.G. de Bruijn qui est basé sur la théorie des types puis parallèlement LCF [56] de Robin Milner qui introduisit les notions de tactique et de combinateur de tactiques (tacticielle). Pour écrire ces tactiques, Milner introduisit un meta-langage qui évolua par la suite séparément pour donner naissance aux langages de la famille ML.

En parallèle au développement des systèmes de preuve, avec la croissance de l'industrie logicielle, apparut la nécessité de prouver mathématiquement la correction de certains programmes. C'est-à-dire le besoin d'exprimer formellement ce qu'on attend d'un programme (sa spécification) et de vérifier que ce programme correspond à la spécification que l'on a donnée. Cela allait étendre considérablement le champ d'application de la démonstration

1. Pour une histoire de la démonstration automatique, on pourra se référer à MacKenzie [78] ou [59]

2. Pour une liste des principaux résultats prouvés avec otter on pourra consulter: http://www-unix.mcs.anl.gov/AR/new_results/

3. Ces systèmes dont la liste n'est en rien exhaustive ont fait l'objet d'une compétition aux deux dernières conférences Cade [131]

sur ordinateur. Aux preuves de résultats mathématiques venaient s'ajouter les preuves de programmes, de protocoles, etc. Un des pionniers en la matière fut le système Nqthm [23, 24] de Boyer et Moore.

C'est aux démonstrateurs interactifs et principalement aux héritiers de la famille LCF que nous nous sommes intéressés. La plupart de ces systèmes ont maintenant atteint un niveau qui permet d'envisager la vérification mécanique de preuves relativement complexes.

Ainsi, dans le système Coq, avec lequel nous avons principalement travaillé, un exemple récent développé à Sophia-Antipolis a permis de prouver la correction d'un algorithme de calcul formel sur les polynômes [136]. Le résultat de cette preuve se traduit par des scripts Coq de 9000 lignes, 120 définitions et 400 théorèmes et le développement s'est étalé sur un an pour un chercheur. L'une des principales conclusions de cette étude est que l'organisation de la librairie de théorèmes revêt une importance majeure. Ainsi, l'ensemble des théorèmes relevant de l'arithmétique de base des polynômes doit être placé au même endroit, les propriétés topologiques à un autre endroit, etc. . L'organisation des théories peut influencer de façon notable sur la vitesse de développement des preuves et leur réutilisabilité, deux facteurs importants pour que le système de preuve soit praticable en milieu industriel.

Des développements importants dans d'autres systèmes conduisent à des conclusions similaires. Ainsi dans [30] Paul Curzon discute l'importance des notions de maintenance et de réutilisation de preuve en HOL en se basant sur ses propres expériences de preuve de matériel [31].

Le récent travail de Tobias Nipkow et David von Oheimb [90] qui formalise en Isabelle/HOL un important sous-langage de Java et prouve la correction du système de type, montre que la vérification sur machine de la conception de langages de programmation non triviaux est devenue une réalité. Ils remarquent par ailleurs que:

> "The adaptation of old proof after changing formalisation is a tedious job. Although the changes in the proofs are usually quite local, there tend to be many. [...] A dedicated mechanism for exploring and fixing the impact of modification would also help"

Ainsi, le développement d'une grosse preuve ou d'une grosse théorie n'est pas une activité parfaitement linéaire. Le développement d'une preuve, surtout sur machine se fait souvent par tâtonnements et tentatives multiples, et comporte de nombreux essais, erreurs et retours en arrière. De plus il est parfois nécessaire d'affiner le contexte et la spécification du problème. Il importe de fournir les outils qui permettent de faciliter ces retours en arrière et d'analyser la portée d'une modification, pour éviter que l'utilisateur ne préfère construire une théorie bancale plutôt que de faire l'effort de la nettoyer.

1.2 Du génie logiciel à l'ingénierie de preuve

Les difficultés inhérentes aux gros développements sur ordinateurs sont bien sûr d'abord apparues dans les activités de programmation. Les efforts pour surmonter les difficultés de

compréhension, de modification, de réutilisation etc. constituent ce que l'on appelle le génie logiciel. Grossièrement deux directions sont possibles.

1° Concevoir de "meilleurs" langages: cela conduit à l'apparition des notions de bibliothèque, de structuration, de modularité, de compilation séparée, d'objets ...
2° Concevoir des outils pour manipuler les langages existants: cela conduit aux environnements de programmation, aux outils d'aide à la gestion de versions, aux outils d'analyse de code, aux outils de transformation de programme ...

En fait ces deux démarches sont bien souvent complémentaires car les outils développés dans la seconde optique améliorent généralement le confort d'utilisation des langages conçus dans la première optique, tandis que la bonne structuration permise par la première permet l'utilisation effective de la seconde.

1.2.1 Preuve et programme

La réalisation de preuves formelles sur machine est très proche de l'activité de programmation. Dans les langages de preuve basés sur une théorie constructive on pourra même extraire un programme de la preuve de sa spécification [95]. Comme un développement logiciel, une preuve a un cycle de vie, on la conçoit, on la code, on la met au point. Le premier pas consiste à formaliser les objets sur lesquels on veut raisonner, puis à formaliser les propriétés que l'on veut prouver, la dernière étape consiste à trouver la structure de la preuve, les lemmes dont on aura besoin, la méthode de preuve.

Ainsi, dès lors que les développements de preuve deviennent importants, on est confronté à des problèmes semblables à ceux que l'on rencontre en programmation. Des solutions similaires à celles mises en œuvre par le génie logiciel, peuvent donc être envisagées. D'un coté, on cherche à améliorer les systèmes en introduisant, dans le cadre des assistants de démonstration, des concepts apparus pour les langages de programmation comme les notions de bibliothèque, de compilation séparée ou de module. De l'autre on adapte les outils de développement et de manipulation de programme.

Le *Synthesizer Generator* [116] et le système *Centaur* [21, 65] sont des outils génériques permettant de créer, à partir d'une spécification formelle d'un langage de programmation (syntaxe et sémantique), des environnements d'édition et des interfaces spécifiques à ce langage. Un environnement interactif de programmation peut comprendre des éléments syntaxiques comme un éditeur structuré ou des outils de formatage de l'affichage, et des éléments sémantiques comme un évaluateur ou un compilateur. L'adaptation des concepts d'environnement de programmation aux assistants de preuve, dont Bertot, Théry et Kahn [135] ont mis en évidence l'intérêt, a ouvert de nouveaux horizons comme les notions de preuve par sélection [15], ou de "drag-and-drop" [13].

Dans cette thèse nous présentons un panorama d'outils d'aide à la compréhension, au développement et à la maintenance de preuves et de théories formelles. Certains s'inspirent d'idées mises en œuvre dans le cadre des langages de programmation comme les techniques

d'analyse de programme ou de gestion et de visualisation des dépendances [130, 73] d'autres comme le retour-arrière logique ou la généralisation de preuve sont plus spécifiques aux systèmes d'aide à la démonstration.

Dans une première partie, nous considérons les problèmes à leur niveau "microscopique", c'est-à-dire que l'on étudie les problèmes liés à la construction d'une preuve particulière et que l'on présente quelques outils permettant d'assister l'utilisateur dans cette tâche [107]. Le travail effectué s'articule principalement autour de quatre aspects: la navigation dans les preuves dirigées par les buts, le retour en arrière logique, l'expansion et la contraction de preuve et l'isolation de fragments de preuve et de lemmes.

Dans une seconde partie nous considérons le niveau "macroscopique" c'est-à-dire que nous étudions les problèmes liés à la construction, la représentation et la maintenance de théories [108]. On retrouve ici de nombreuses similarités avec les concepts et les outils de génie logiciel mais ils prendront parfois une signification différente dans le cadre des systèmes d'aide à la preuve.

Enfin nous étudions un certain nombre d'outils qui ont besoin d'information des deux niveaux (microscopique, macroscopique) comme l'aide à la généralisation de théorèmes et à leur réutilisation.

Nous présentons maintenant le cadre qui a servi de base à notre travail.

1.3 Cadre de ce travail

1.3.1 Logique et preuves formelles

En logique formelle, une preuve est un enchaînement de règles de déduction. On peut procéder déductivement (par chaînage avant), c'est-à-dire en déduisant récursivement d'un théorème d'autres théorèmes jusqu'à l'obtention du résultat recherché. La plupart des systèmes de preuve fonctionnent cependant par chaînage arrière. On commence par un but, constitué de l'énoncé du théorème que l'on veut prouver et on le transforme récursivement en des buts plus simples. On dit que la preuve est dirigée par les buts. Dans un tel cadre, une preuve est un arbre dont les nœuds sont des règles du système d'inférence utilisé. La conclusion de la règle portée par la racine est le but initial. La preuve est complète si toutes les feuilles sont des axiomes ou des théorèmes connus.

Depuis LCF, les fonctions qui réduisent un but en sous-buts sont appelées des tactiques. Ces tactiques peuvent être des règles de base de la logique sous-jacente ou des fonctions plus compliquées construites à partir des règles primitives. On dispose également d'opérateurs appelés tacticielles[4], qui permettent de combiner des tactiques pour obtenir de nouvelles tactiques.

Dans LCF, une tactique est une fonction ML qui prend un but et renvoie un couple formé d'une liste de sous-buts restant à prouver et d'une validation. La validation est une

4. ce néologisme est calqué sur le terme "fonctionnelle"

fonction permettant de générer la preuve complète à partir des preuves des sous-buts. Tactiques et fonctions de validation ont donc les types suivants:

```
tactic : goal -> (list goal) * validation
validation  : (list theorem) -> theorem
```

La preuve du but de départ peut donc être reconstruite à partir de la preuve des sous-buts grâce à la fonction de validation.

1.3.2 Quelques critères importants

Les systèmes que nous qualifions d'héritiers de LCF implémentent tous ces notions de tactique et de tacticielle. Ils se différencient essentiellement par la logique sur laquelle ils sont basés ainsi que par les caractéristiques de leur implémentation.

Les critères importants pour notre travail sont les suivants:

- **Existence d'un "objet preuve"**.
  Un objet preuve est une représentation rémanente d'une preuve. Cette représentation peut être un arbre de dérivation dont les nœuds sont des règles de base de la logique d'implémentation ou, ce qui dans le cas des systèmes basés sur une théorie typée est équivalent, un λ-terme. Dans ce dernier cas on parlera souvent de terme de preuve. Dans les systèmes qui n'ont pas d'objet preuve, un théorème, une fois prouvé, est ajouté dans une base de faits comme un axiome de la théorie que l'on est en train de construire.
  Remarquons par ailleurs que l'existence d'un objet preuve permet d'augmenter la confiance que l'on a en une preuve, en la faisant vérifier (valider) par un vérificateur de preuve indépendant du système l'ayant générée. L'idée est qu'un simple vérificateur de preuve étant un système plus simple, il pourra plus facilement être vérifié qu'un système de preuve complet. (Dans le système Coq par exemple, seul le vérificateur de type est crucial).
- **Présence de variables existentielles dans les buts.**
  Certains systèmes utilisent des méta-variables pour représenter en interne la preuve de résultats qui n'ont pas encore été prouvés. De plus, il peut être utile d'autoriser l'utilisateur à introduire des méta-variables au niveau des sous-buts. En effet, sans cette possibilité, pour prouver un résultat de la forme "$\exists x$ tel que $P(x)$", il faut être capable de déterminer un témoin, c'est-à-dire un terme t tel que $P(t)$ soit vrai, dès le départ pour pouvoir éliminer l'existentielle. Si l'on a la possibilité d'introduire des méta-variables dans les sous-buts, on parle alors de variables existentielles, on pourra transformer un but de la forme précédente en $P(X)$ où X est une variable existentielle. On peut ensuite continuer à raisonner sur $P(X)$ avant d'instancier X.
  Soulignons qu'en l'absence de variables existentielles, tous les sous-buts générés par une tactique sont indépendants. C'est-à-dire que, quelle que soit la façon dont ils seront prouvés, la validation permettra de reconstruire à partir de leur preuve une preuve du but de départ. Cela n'est plus forcément vrai en présence de variables existentielles. Lorsque deux sous-but partage une variable existentielle, elle devra

bien sûr être instanciée de la même manière dans la preuve de chacun des sous-buts. On dira alors qu'il existe des contraintes entre les sous-buts.
L'utilisation des variables existentielles et les problèmes qui y sont liés sont discutés en détail au chapitre 3.

- **Manière dont sont référencés les sous-buts restant à prouver.**
  Lorsque l'application d'une tactique produit plusieurs sous-buts, il faut à l'étape suivante désigner le sous-but que l'on désire prouver. Ce choix peut être fait par le système ou laissé à l'utilisateur. Dans ce dernier cas, la façon dont l'utilisateur désigne le sous-but sur lequel il désire travailler dépend du système. Il peut, par exemple, fournir le chemin de ce sous-but dans l'arbre de preuve. Cette méthode est très robuste, car ce chemin reste inchangé dans de nombreuses manipulations de l'arbre, mais devient vite contraignant si on manipule de grosses preuves.
  Généralement le système ordonne les sous-buts par un parcours de l'arbre de preuve. L'utilisateur peut alors selon les systèmes, changer l'ordre des sous-buts et placer en premier celui sur lequel il désire travailler, ou désigner directement le sous-but qui l'intéresse en donnant sa position dans la liste ordonnée des sous-buts. On parle alors de l'index du sous-but ou plus précisément de l'index de la tactique qui est appliquée à ce sous-but.
- **Manière dont sont référencées les hypothèses.**
  Une étape de preuve très courante consiste à transformer un but de la forme $A \rightarrow B$ en un but $A \vdash B$. On dit alors que l'on introduit l'hypothèse A dans le contexte local (on dira parfois la base locale d'hypothèses). Elle pourra être utilisée dans la preuve de B. Pour utiliser des hypothèses du contexte local, il faut être capable d'y faire référence. Cela peut se faire de plusieurs façons selon le système utilisé, par exemple : par leurs positions dans la liste d'hypothèses, en les citant explicitement, en les nommant. Ce problème est développé au chapitre 5 quand nous présentons le mécanisme de "lemmification".
- **Le fonctionnement de l'automatisation.**
  Certains systèmes comportent des procédures de décision automatique. Comme nous le verrons au chapitre 9, le comportement de ces procédures peut être très dépendant du contexte dans lequel elles apparaissent (c'est-à-dire de l'ensemble des théorèmes qui ont été prouvés précédemment) et cela complique sensiblement les activités de maintenance.

Parmi les différents prouveurs que nous avons étudié, nous avons choisi le système Coq pour mener nos expérimentations. Ce système, que nous présentons en détail au chapitre 2, est basé sur une théorie des types, le Calcul des Constructions Inductives. Il possède des objets preuve, et permet la manipulation de variables existentielles. On y indique à quel sous-but s'applique une tactique en utilisant l'index de ce sous-but. Les hypothèses sont référencées par un nom et le système possède des procédures de décision automatique.

Le choix de ce système parmi les nombreux systèmes existants est justifié par le fait qu'il permet d'allier les avantages de la théorie des types d'ordre supérieur (en particulier l'existence d'objets preuve sous forme de λ-terme) et des procédures de preuve à la LCF. Si de nombreuses analyses pertinentes peuvent être faites sur les objets preuve, l'apport des tactiques est également indéniable en ce qu'il reflète la démarche de preuve tout en permet-

tant d'intégrer des composantes de preuves automatiques. Ce point de vue hybride fournit des contraintes nouvelles pour certains problèmes (comme le problème du retour-arrière logique ou la modification de théorie).

Les outils et prototypes présentés dans ce travail ont donc été partiellement réalisés dans le système Coq et son interface utilisateur CtCoq. Mais comme nous essayons de le montrer, les méthodes utilisées peuvent souvent être appliquées à d'autres systèmes.

1.3.3 Présentation de quelques systèmes de preuves

Nous présentons maintenant rapidement les autres systèmes cités dans ce document ainsi que quelques-uns de leurs environnements de développement (des exemples de sessions complètes sont donnés en annexe A).

Des travaux de comparaison entre différents outils d'aide à la preuve ont par ailleurs été menés. Citons sans prétendre être exhaustif, le travail de Zammit [148] pour Coq et HOL, de Griffioen et Huisman [57] pour PVS et Isabelle/HOL. Dans [66] F. Kammüller a, par ailleurs, comparé IMPS, PVS et Isabelle en s'intéressant surtout à leur traitement de la modularité. Enfin, [2] compare HOL et Alf.

La plupart des systèmes que nous décrivons ci-dessous sont des héritiers de LCF même s' ils n'en ont pas tous gardé le style. Ils implémentent les notions de tactique et de combinateur de tactiques ou tacticielle. On termine ce chapitre avec quelques autres démonstrateurs qui, bien que d'une filiation différente, comportent des caractéristiques intéressantes ou ont connu un certain succès industriel.

HOL

On peut considérer que l'héritier le plus direct de LCF est HOL [55], ne serait-ce que parce que l'un des concepteurs, M. Gordon, était aussi un des principaux acteurs du projet LCF. Le système est basé sur la logique classique d'ordre supérieur. La première version, HOL88 [55] est écrite en Lisp (une partie du code vient directement de LCF). Un nouvelle version, HOL90 [126] est une réécriture en SML [87, 101]. C'est aussi SML qui sert de méta-langage pour écrire les tactiques. K. Slind a récemment introduit une nouvelle version, HOL98, en Moscow ML.

Une preuve peut se faire par chaînage avant mais se fait généralement par chaînage arrière. Comme dans LCF, les tactiques sont des fonctions ML qui opèrent sur un sous-but donné. Il n'y a pas de variables existentielles et les différents sous-buts engendrés par une tactique sont donc indépendants. Enfin pour choisir quel sous-but attaquer, il faut mettre le but sur lequel on veut travailler au sommet de la pile des sous-buts ouverts. Cela se fait par rotation des sous-buts dans la pile grâce a la fonction `rotate_proofs`. L'exemple 1.1 montre une preuve en HOL98 (le ! se lit $\forall$)

Exemple 1.1

```
                g‘! m n. m < n ==> !p. m < n + p‘;;
Ligne 1         e(REPEAT GEN_TAC);
Ligne 2         e(STRIP_TAC);                (* introduction *)
Ligne 3         e(Induct_on ‘p‘);            (* induction sur p *)
Ligne 4         r 1;                         (* change l’ordre des sous-buts *)
               (* étape d’induction *)
Ligne 5         e(IMP_RES_TAC LESS_SUC );
Ligne 6         e(ASM_REWRITE_TAC [ADD_CLAUSES]);
                (* cas de base *)
Ligne 7         e(ASM_REWRITE_TAC [ADD_CLAUSES]);
```

Les lignes 1 et 2 permettent d'introduire les hypothèses dans le contexte local. La ligne 3 permet le raisonnement par récurrence sur l'entier `p`*. Elle génère donc deux sous-buts correspondant respectivement au cas de base (*`p=0`*) et à l'étape d'induction où l'on suppose la propriété vrai pour* `k` *et où l' on essaie de la montrer pour* `k+1`*. La ligne* `4` *inverse l'ordre de ses sous-but dans la piles des résultats à prouver. On commence donc par prouver le cas d'induction par les lignes 5 et 6 puis le cas de base avec la ligne 7.* $\diamondsuit$

Il n'existe pas d'objet preuve en HOL mais, Wong [145] a implémenté (en modifiant HOL88) la notion de preuve totalement expansée qui permet de sauvegarder une preuve comme une combinaison des huit règles de base (ou d'un sur-ensemble de ces huit règles) de la logique d'ordre supérieur. Les preuves peuvent alors être vérifiées par tout autre vérificateur de preuve, implémentant cette logique. Wong a par ailleurs développé un tel vérificateur [146]. Les problèmes de maintenance de preuve en HOL ont été discutés par Curzon dans [31, 30].

Il existe de nombreux environnements pour HOL. Windley a conçu un environnement Emacs [144] dans lequel Curzon implémente la notion de "virtual theories" [32] discutée au chapitre 6. Du coté des interfaces graphiques, Schubert et Biggs [121] proposent une interface sous X, qui permet de représenter graphiquement les arbres de preuve, Syme a développé un environnement complet tkHol [132] basé sur TclTk [91]. Enfin, un environnement structuré basé sur Centaur, CHOL, dû à Laurent Théry [134] qui a été le véritable point de départ des expériences de Bertot, Théry et Kahn [135] sur l'adaptation des concepts, outils et méthodes des environnements de programmation aux environnements de preuve. Il permet par exemple la visualisation des dépendances entre théories. La conception de l'environnement CtCoq dans lequel nous avons travaillé a été très influencée par CHOL.

Isabelle

Isabelle [97, 102, 98, 99, 100] est un démonstrateur de théorèmes génériques utilisant ML comme méta-langage. De nouvelles logiques sont introduites en spécifiant leur syntaxe et leurs règles d'inférence. Les procédures de preuve sont exprimées par des tactiques et des tacticielles. De nombreuses logiques du premier ordre (FOL, ZF, LCF etc.) ou d'ordre supérieure avec en particulier HOL y sont implémentées.

Le système est basé sur un λ-calcul simplement typé et utilise l'unification d'ordre supérieur et la résolution.

On y dispose de quatre tactiques de base et de tacticielles à la LCF. Mais les tactiques d'Isabelle diffèrent de celles de LCF, HOL, Nuprl ou Coq en ce sens qu'elles opèrent sur l'état global de la preuve et non sur un sous-but donné. Même lorsque l'on écrit quelque chose comme:
`by (resolve_tac thm 3);`
3 est un argument de la tactique, qui elle, s'applique à l'état global, en se sens que les substitutions issues de l'unification de la conclusion de `thm` avec le 3^{e} sous-but sont appliquées à tous les sous-buts. L'exemple 1.2 montre une preuve élémentaire en Isabelle.

Exemple 1.2

```
Goal "(P & Q) | R --> (P | R)";
by (resolve_tac [impI] 1);
by (eresolve_tac [disjE] 1);
by (dresolve_tac [conjunct1] 1);
by (resolve_tac [disjI1] 1);
by (resolve_tac [disjI2] 2);
by (REPEAT (assume_tac 1));
```

Isabelle procède par unification d'ordre supérieur, la première ligne par exemple unifie le but courant avec le théorème `impI` *d' énoncé:* `"(?P ==> ?Q) ==> ?P --> ?Q"`. ♢

On peut heureusement abréger les noms de tactique comme le montre la preuve de la commutativité de la conjonction de l'exemple 1.3.

Exemple 1.3

```
goal HOL.thy
"(A & B)-->(B & A)";
br impI 1;
br conjI 1;
be conjE 1;
ba 1;
be conjE 1;
ba 1;
qed "and_comms";
```

Du côté des interfaces, on peut citer les expériences de Bertot, Thery et Kahn [135] utilisant Centaur et l'interface XIsabelle [36] de K. A. Eastaughffe que nous présentons au chapitre 3.

Lego

Le système Lego [77, 76, 106] implémente différents systèmes de types (LF, CC, GCC, UTT) et surtout ECC [75]. Ecrit en SML, c'est le plus proche de Coq, d'abord puisque ECC et CCI sont très proches mais aussi dans son mode de fonctionnement. Ainsi la plupart des choses que nous dirons sur Coq restent valables en Lego. Notons néanmoins quelques différences qui influencent la conception et la mise en œuvre des outils que nous décrivons dans le reste de cette thèse.

Lego dispose de variables existentielles qui servent à représenter des sous-buts ouverts. Elles peuvent être utilisées pour retarder la preuve d'un résultat. Elles peuvent servir d'argument implicite, lors de l'application d'un terme fonctionnel, représentant un type qui sera synthétisé par le système. Elles peuvent être introduites dans un but par la tactique `Intros`. Si ce but génère à son tour plusieurs sous-buts, ces derniers partagent la variable existentielle et ne sont donc plus réellement indépendants.

D'autre part quand une tactique a généré plusieurs sous-buts, il est moins facile que dans Coq de décider quels sous-buts on désire attaquer. On dispose de la commande `Next` pour changer l'ordre des sous-buts restant à prouver mais elle restreint les permutations du but courant à l'ensemble de ses frères.

Enfin si la plupart des tactiques travaillent sur le but courant (le premier de la liste des sous-buts ouverts), il existe des tactiques qui travaillent sur l'ensemble des sous-buts ouverts. Par exemple la commande `Immed` résout tous les sous-buts solvables par des résultats du contexte courant.

L'exemple 1.4 montre ces différents points (`<a:A>` se lit $\forall a : A$).

Exemple 1.4

```
Logic;
[A:Type][f,g:A->Prop];
Goal (<a:A>(and (f a)(g a)))-><b:A>(and (g b)(f b));
Intros s;
Intros #; (* introduction d'une variable existentielle *)
Next +1;  (* permutation des buts *)
andE s.2;
andI;
Refine H1;
Immed;
```

L'introduction d'une variable existentielle de type `A` *crée un second sous-but correspondant à la preuve que* `A` *n'est pas vide.* ♢

Du côté des interfaces, mentionnons une expérimentation [138] basée sur Centaur (avec notamment la construction interactive de l'arbre de preuve en utilisant le serveur de graphe [68, 74] qui fonctionne avec Centaur).

D. Aspinall, H. Goguen, T. Kleymann et D. Sequeira ont développé sous Emacs un environnement générique, Proof General [8] utilisable pour Coq, Isabelle et Lego. Leur environnement reprend de nombreuses idées et outils comme la preuve par sélection [16], d'abord développés dans le cadre d'un environnement structuré comme CtCoq en les adaptant au cadre moins structuré fourni par Emacs.

Nuprl

Ecrit en Lisp, le système Nuprl [26, 25] est basé sur la théorie des types de Martin-Löf [84, 83]. Il possède comme Coq une représentation des objets preuves sous forme de λ-terme et dispose d'un mécanisme d'extraction. Il s'accompagne d'un environnement de développement multi-fenêtres et s'appuie sur des possibilités d'édition inspirées du *Cornell Program Synthesizer* [116].

Alf

Le système Alf [81, 80] est aussi basé sur la théorie des types de Martin-Löf avec substitutions explicites. La version que nous avons manipulée est celle décrite dans la thèse de Lena Magnusson. Elle se présente sous la forme d'un environnement d'édition de preuves structurées qui se caractérise par le fait que l'on manipule (édite) directement le λ-terme. Les deux opérations de base sur le terme sont l'insertion et l'effacement. C'est à partir d'elles que sont développées toutes les autres manipulations ainsi qu'un mécanisme de "undo local" dont nous reparlerons au chapitre 3. Le système se compose d'un moteur de preuve écrit en ML et d'une interface d'édition multi-fenêtres.

PVS

L'assistant de preuve PVS [123] est basé sur une logique classique d'ordre supérieur avec fonctions, ensembles, enregistrements, n-uplets, sous-typage, types dépendants, et théories paramétrées. On y construit des scripts structurés mais il n'y a pas d'objet preuve. Il permet en outre de construire des théories paramétrées. Le système fonctionne avec une interface sous Emacs, et utilise TclTk [91] pour les représentations graphiques.

Du fait de l'utilisation de sous-types dépendant de prédicats, la vérification du typage des définitions de fonction entraîne des obligations de preuves. Les preuves proprement dites se font comme dans les autres systèmes à l'aide du langage de tactiques. Une fois terminée la preuve peut être sauvée dans un fichier qui pourra être automatiquement rappelé et rejoué lors d'une prochaine session. Comme dans HOL ou Lego, lorsqu'une tactique génère plusieurs sous-buts, on peut passer de l'un à l'autre en les permutant dans la pile au moyen de la commande `postpone`. Enfin, une fois un théorème prouvé, il est possible de demander de quels autres théorèmes et de quelles définitions il dépend (mais la documentation ne mentionne pas comment ces dépendances sont calculées). Enfin le système utilise Latex pour permettre un affichage plus agréable de ses théories.

1.3.4 Quelque autres systèmes

Les démonstrateurs conçus par Amy Felty dans son travail de thèse [45, 46] qui implémentent la logique intuitioniste du premier ordre dans une version mixte du calcul des séquents et de la déduction naturelle ont aussi influencé nos travaux. Implementés en Lambda-Prolog ces travaux ont servi de base à ses expérimentations sur la manipulation de preuves, les problèmes de retour-arrière, de correction et de réutilisation de preuve [50, 49] que nous abordons au chapitre 3. Ils ont aussi servi de support aux travaux de Penny Anderson sur les transformations de preuve [4, 5].

Parmi les autres démonstrateurs cités dans ce travail, mentionnons: IMPS [39, 40, 42, 43, 38] qui est une version non constructive de la théorie des types autorisant la définition de fonctions partielles et de sous-types. Le système autorise le développement de théories parametrées au travers de la notion de "little theories" [41]. Ce système dispose d'une interface assez conséquente construite sous Emacs.

Le "Karlsruhe Interactive Verifier" [112, 113] suit aussi la tradition de LCF mais se caractérise par un haut degré d'automatisation: la sélection des tactiques se fait le plus souvent par heuristique. Lorsque toutes les heuristiques sélectionnées échouent, l'intervention de l'utilisateur est requise dans le choix des tactiques ou de nouvelles heuristiques, ainsi que dans la décision de revenir en arrière ou d'ajouter des lemmes intermédiaires. Le système comporte par ailleurs une interface graphique et une organisation hiérarchique des projets que l'on peut visualiser en utilisant le logiciel de dessin de graphe daVinci [52] que nous avons aussi utilisé.

Mentionnons aussi le Système Jape [20] développé par Richard Bornat et Bernard Sufrin à des fins pédagogiques. Il permet de manipuler différentes logiques en déduction naturelle. L'interface présente de nombreuses caratéristiques comme la construction guidée par des menus ou la manipulation de preuves à la souris. La visualisation des preuves par chaînage arrière en utilisant un système de boites qui permet d'en visualiser et d'en comprendre la structure est également intéressante.

Il nous faut aussi citer Omega [61] et Ergo [137] dont nous mentionnons les interfaces. Enfin terminons cette rapide exposition des prouveurs généralistes en citant B [1] et LP [58, 82] qui ont connu ou connaissent un certain succès industriel.

1.4 Remarques générales

La variété des applications abordées dans cette thèse s'appuie sur quelques lignes directrices.

La notion de dépendance y joue un rôle central. On s'intéresse d'abord aux dépendances entre différentes parties d'une preuve, puis entre différentes preuves et entre différentes théories. La structure permettant de modéliser les premières est celle d'arbre de preuve que nous introduirons formellement au chapitre 3. Pour modéliser les secondes nous utilisons

au chapitre 6 une notion de graphe de dépendance représenté par un graphe orienté sans circuit.

Pour les systèmes dans lesquels un objet preuve est maintenu, un autre problème fondamental est la dualité script de preuve/objet preuve. Le script reflète la façon dont l'utilisateur construit ses preuves. Les objets preuve sont utilisés pour la représentation interne des preuves (et éventuellement à des fin d'extraction).

Quelle que soit la manière dont le résultat prouvé est conservé par le système, ce qui nous intéresse en priorité, c'est de pouvoir le reproduire (on dit alors que l'on rejoue la preuve). Les manipulations et transformations de preuve sont faites le plus possible sur le script et non sur une représentation interne de la preuve, même si cette dernière doit être manipulée par nos outils pour garder la cohérence entre le script et l'état du démonstrateur. Néanmoins, pour certaines applications, il est impossible de ne travailler que sur le script et nous devons utiliser le terme de preuve.

Remarquons qu'il est toujours possible de régénérer un script à partir de ce terme de preuve. Ce script sera alors formé des règles de base de la logique sous-jacente. Ces règles de base sont les règles de typage du terme de preuve et il existe un isomorphisme entre l'arbre de dérivation constitué de ces règles et le terme de preuve. Mais il est beaucoup moins coûteux de conserver le terme plutôt que l'arbre de typage.

Dans les systèmes qui ne sont pas basés sur une théorie typée, et où il n'y a donc pas de terme de preuve, pour des raisons de taille on ne conserve généralement pas l'arbre de dérivation constitué des règles primitives, le résultat prouvé est simplement ajouté dans la base de faits du système.

La majorité de nos outils manipule les scripts et, chaque fois que nous sommes amenés à transformer un script, nous cherchons à rester le plus proche possible du script originalement écrit par l'utilisateur.

Chapitre 2

Présentation détaillée de Coq et de l'environnement CtCoq

2.1 Présentation de Coq

L'assistant à la démonstration **Coq** est développé depuis 1984 à l'INRIA, 1989 à l'ENS Lyon et 1997 à l'université Paris-Sud. Il a évolué d'un simple vérificateur de type pour un noyau purement fonctionnel à un environnement complet de développement intégrant une compilation séparée, des notations extensibles, des tactiques sophistiquées. Nous donnons maintenant une présentation intuitive de ce système.

C'est un système basé sur une théorie des types d'ordre supérieur qui privilégie la notion de définition inductive *le Calcul des Constructions Inductives* [96, 142]. Le langage de spécification ainsi défini combine la programmation fonctionnelle, les déclarations de prédicats à la Prolog et un calcul des prédicats d'ordre supérieur.

2.1.1 Terme de preuve versus script de preuve.

Une formule (i.e. une spécification) est représentée par un type, une preuve de cette formule (ie. le programme qui la réalise) est un λ-terme de ce type.

Par exemple, en Coq, on exprimera l'associativité à gauche de l'addition par le théorème suivant:

```
Lemma plus_assoc_l : (n,m,p:nat)((plus n (plus m p))=(plus (plus n m) p)).
```

et une preuve de ce théorème sera le λ-terme suivant:

```
[n,m,p:nat]
 (nat_ind [n0:nat](plus n0 (plus m p))=(plus (plus n0 m) p)
   (refl_equal nat (plus m p))
   [n0:nat]
    [H:(plus n0 (plus m p))=(plus (plus n0 m) p)]
     (eq_ind_r nat (plus (plus n0 m) p)
       [n1:nat](S n1)=(S (plus (plus n0 m) p))
       (refl_equal nat (S (plus (plus n0 m) p))) (plus n0 (plus m p)) H)
   n)
```

La notation `(n,m,p:nat)` doit être lue $\forall n,m,p.nat$ et `[ ]` est la notation pour la λ abstraction. `nat_ind` est le théorème de récurrence sur les entiers naturels dont énoncé est: `(P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)`
Ainsi, le terme de preuve précédent représente une preuve par récurrence sur n.

Prouver une formule consiste donc à exhiber un terme dont elle est le type. Et vérifier qu'un terme est bien la preuve d'un énoncé donné consiste juste à en vérifier le typage.

Comme on le constate en regardant le terme précédent, une preuve sous forme de λ-terme est assez peu lisible pour le commun des mortels. Un axe de recherche en développement [27] consiste à essayer de traduire ce λ-terme en langage pseudo-naturel.

L'utilisateur n'aura donc généralement pas à manipuler ces termes qui restent la représentation interne. D'un point de vue opérationnel, il construira sa preuve par applications successives de "commandes" du "langage de preuve" du système. On dit que la construction de la preuve est dirigée par les buts, c'est-à-dire qu'elle est construite par raffinements successifs d'un but initial à l'aide de ce qu'on appelle des *tactiques*[1].

Ces tactiques sont des règles d'inférence du *Calcul des Constructions Inductives*, appliquées de bas en haut, des règles moins primitives ou des programmes fonctionnels. Elles peuvent être composées entre elles, pour former de nouvelles tactiques, par ce que l'on appelle des opérateurs de tactiques ou tacticielles.

Par exemple, la preuve du lemme précédent peut être construite en appliquant successivement les tactiques de la Figure 2.2.

1. La figure 2.1 page 23 décrit les tactiques les plus utilisées dans ce document.

```
L1      Intros n m p.
L2      Elim n.
-         (* cas de base *)
L3      Simpl.
L4      Apply refl_equal.
-         (* cas de récurence *)
L5      Intros.
L6      Simpl.
L7      Rewrite ->H.
L8      Apply refl_equal.
```

FIG. 2.2 – *Un script de preuve de l'associativité à gauche de l'addition*

De façon interne le système manipule un λ-terme avec des trous; à chaque trou correspond un sous-terme qu'il faut construire.

On peut observer ci-après, l'état interne du système après l'application des tactiques: `(Intros n m p)` et `(Elim n)`:

```
   < terme de preuve > =

   [n,m,p:nat]
  (nat_ind [n0:nat](plus n0 (plus m p))=(plus (plus n0 m) p)
   (?2067 n m p) (?2068 n m p) n)
```

Les nombres précédés d'un point d'interrogation sont des variables existentielles (ou méta-variables). Elles représentent les sous-termes de preuve restant à construire. A chacune de ces variables existentielles est associée un sous-but restant à prouver (on parlera aussi de sous-but ouvert). Ici on a:

- **?2067** associée à
  ```
  nat->(m,p:nat)(plus O (plus m p))=(plus (plus O m) p);
  ```
- **?2068** associée à
  ```
  nat ->(m,p,n:nat)
   (plus n (plus m p))=(plus (plus n m) p)
   ->(plus (S n) (plus m p))=(plus (plus (S n) m) p)]
  ```

La preuve incomplète peut donc se représenter comme un couple dont le premier membre est le terme de preuve avec ses méta-variables et le deuxième la liste des variables existentielles et des sous-buts qui leurs sont associés (`nat_ind` est le théorème de récurence sur les entiers naturels).

Chaque sous-terme à fournir correspond à un sous-but à prouver par application des tactiques. Lorsqu'il n'y a plus de sous-but, c'est que dans la représentation interne le λ-terme n'a plus de trou, et la preuve est donc complète. Tout cela est transparent à l'utilisateur et pour lui, la construction de la preuve consiste donc à transformer récursivement des sous-buts en utilisant les tactiques à sa disposition.

Dans notre exemple, l'utilisateur est parti du but

```
(n,m,p:nat)(plus n (plus m p))=(plus (plus n m) p).
```

et l'application des tactiques `(Intros n m p)` et `(Elim n)`, a engendré deux sous-buts:

```
(plus O (plus m p))=(plus (plus O m) p)
```

et

```
(n:nat)(plus n (plus m p))=(plus (plus n m) p)
   ->(plus (S n) (plus m p))=(plus (plus (S n) m) p)
```

FIG. 2.3 – *Une étape de construction de la preuve*

La démonstration (complète ou non) apparaît comme une suite de tactiques que nous appellerons *script de preuve*. Chaque tactique est précédée d'un numéro qui correspond à l'indice du sous-but auquel elle s'applique dans la liste des sous-buts ouverts (c'est 1 par défaut).

Remarquons qu'aucune variable existentielle n'apparaît dans le script ou dans les sous-buts. On pourrait en déduire qu'elles n'existent qu'au niveau de la représentation interne mais nous verrons au chapitre 3 qu'elles peuvent aussi être utilisées par l'utilisateur notamment pour lui permettre de retarder l'instanciation d'un témoin lors de l'élimination d'un quantificateur existentiel.

2.1.2 Structure du script.

Nous avons donné figure 2.2 le script "dans l'ordre". Dans ce script en effet, à chaque étape, c'est le premier but ouvert que propose le système qui est attaqué. Mais on peut construire la preuve en attaquant les sous-buts dans un ordre quelconque. On précise alors le sous-but sur lequel on veut travailler grâce à son indice dans la liste des sous-buts ouverts. On précède la tactique utilisée par cet indice [2] comme cela est illustré sur la figure 2.4

```
L1    1:Intros n m p.
L2    1:Elim n.
L3    2:Intros.
L4    1:Simpl.
L5    2:Simpl
L6    2:Rewrite ->H.
L7    1:Apply refl_equal.
L8    1:Apply refl_equal.
```

FIG. 2.4 – *Autre script de preuve de l'associativité à gauche de l'addition*

2. l'indice 1 est facultatif

Notons que, bien que ce script soit linéaire, il cache une structure arborescente, chaque nœud est une tactique, et les branches qui en sont issues représentent les sous-buts générés.

Ainsi les scripts de preuve que nous venons de montrer (figure 2.2 et 2.4) correspondent tous deux à l'arbre de la figure 2.5.

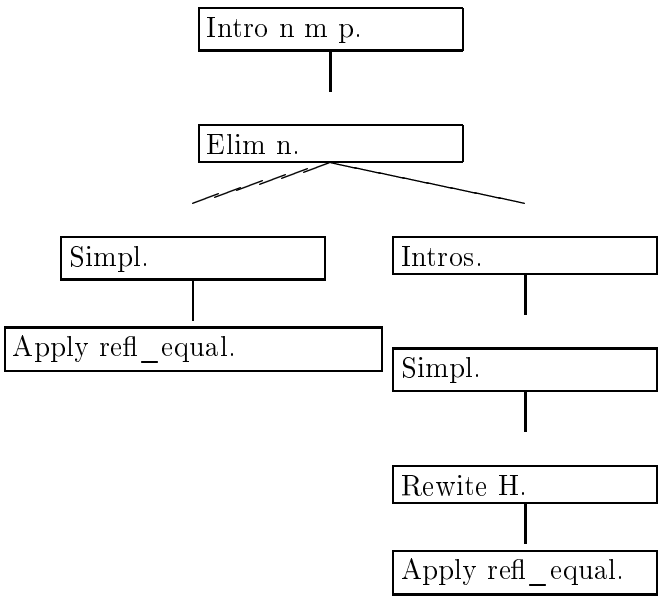


FIG. 2.5 – *L'arbre de preuve de l'associativité à gauche de l'addition*

Cette structure arborescente est d'ailleurs également maintenue, de façon interne, par le système Coq. En fait la structure maintenue par Coq contient toutes les informations sur la preuve en cours. Pour des raisons de coût elle n'est pas conservée une fois la preuve terminée. Nous verrons que pour certaines applications il peut néanmoins être utile de la sauvegarder.

Les tacticielles permettent de regrouper en une seule tactique plusieurs étapes de preuve; ainsi le script suivant construit la même preuve que les précédents mais il est plus compact, certains nœuds de l'arbre de preuve ont été regroupés.

```
L1    Intros n m p;Elim n.
L2    1:Simpl;Apply refl_equal.
L3    1:Intros;Simpl.
L4    1:Rewrite ->H;Apply refl_equal.
```

FIG. 2.6 – *Version plus compacte du script de preuve de l'associativité à gauche de l'addition*

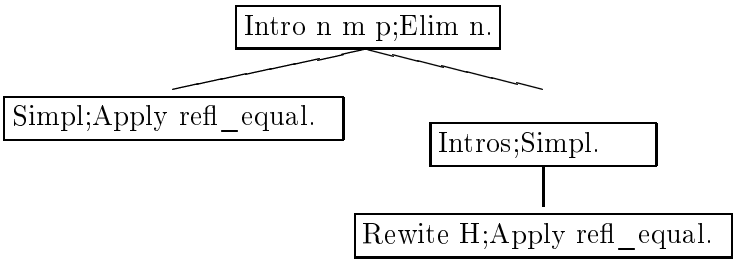


FIG. 2.7 – *L'arbre de preuve compacté*

Au cours du développement d'une preuve, la stratégie de démonstration se reflète donc dans la construction d'un arbre de preuve dont les nœuds sont les tactiques que l'on applique. Pour aider à s'orienter dans la structure de la preuve que l'on construit, nous utilisons la forme arborescente sous-jacente à la preuve pour fournir d'abord des outils de navigation dans l'arbre puis des outils permettant de modifier une partie donnée à l'intérieur d'une preuve sans affecter les parties indépendantes.

2.1.3 Présentation de l'environnement de preuve CtCoq

Nous présentons ici l'interface CtCoq dans laquelle certains de nos outils ont été implémentés. CtCoq[12] fournit un environnement de travail pour le système de preuve Coq. Cet environnement a été développé en utilisant une méthode générale pour la construction d'interfaces homme-machine pour les systèmes de preuve [17, 135].

Point de vue de l'utilisateur

L'environnement Ctcoq supporte le langage complet de Coq. Il a en outre les caractéristiques suivantes:

- *une interface graphique*: les formules et les commandes sont affichées en utilisant des polices de caractères et des couleurs multiples.
- des mécanismes d'*édition et de présentation structurées*: l'environnement fournit les moyens d'éditer structurellement formules et commandes, des sous-expressions entières peuvent être réduites en un seul geste avec la souris et de nouvelles notations peuvent être ajoutées aisément.
- une édition dirigée par des menus: l'environnement fournit des menus pour construire de nouvelles commandes et formules, ou pour les transformer. Ces menus sont facilement extensibles.
- la *preuve par sélection*: l'environnement utilise la structure des formules logiques pour aider systématiquement l'utilisateur à guider la preuve par de simples sélections à la souris[15].

Dans CtCoq, on dispose d'outils de gestion du script. Ils permettent de conserver une version "propre" du script des commandes qui ont été envoyées avec succès au démonstrateur pendant une session de travail. Ce script pourra être rejoué lors d'une nouvelle session et conduira au même résultat final sans provoquer d'erreur.

La figure 2.8, page 24, montre l'environnement de travail complet. La fenêtre principale (au centre) est composée de trois sous-fenêtres.

1° la fenêtre de *Commande* dans laquelle l'utilisateur édite et visualise un script
2° la fenêtre *d'Etat* dans laquelle sont affichés les buts restant à prouver
3° la fenêtre de *Contexte* dans laquelle sont affichés les résultats d'une recherche (commande `Search`) dans la base de données du démonstrateur.

De plus, entre les fenêtres de *Commande* et *d'Etat*, une ligne de boutons contrôle les interactions entre l'environnement et le démonstrateur. Le bouton `Do it` envoie la commande contenue dans la sélection courante au démonstrateur, `Discard` envoie l'ordre de défaire la dernière commande jouée (c'est la commande `Undo` de Coq) et `Abort` envoie l'ordre d'abandonner la preuve en cours.

La fenêtre de *Commande* contient différentes régions qui peuvent être distinguées par la couleur de fond:

1° la *zone finale* qui contient les commandes qui ont déjà été exécutées par le système de preuve (en haut et plus foncé sur la figure);
2° *zone tampon* qui contient la commande qui est en cours d'exécution dans le démonstrateur;
3° *zone d'attente* qui contient les commandes qui attendent d'être envoyées au système de preuve;
4° *zone normale* qui contient les commandes qui sont en cours d'édition mais n'ont pas encore été mises dans la zone d'attente.

Quand l'utilisateur demande l'envoi d'une commande au système de preuve (bouton `Do it`), elle est stockée dans la *zone tampon* jusqu'à réception d'une réponse du démonstrateur (s' il y a déjà une commande elle est stockée momentanément dans la *zone d'attente*). Si la réponse est un message d'erreur, la commande envoyée repasse dans la *zone normale* ainsi que toutes celles qui sont dans la *zone d'attente*, sinon elle passe dans la *zone finale* et la première commande de la *zone d'attente* est envoyée au système et passe dans la *zone tampon*.

On est ainsi assuré que les commandes sont exécutées dans l'ordre où elle sont envoyées.

De plus l'utilisateur ne peut pas écrire directement dans la *zone finale* de sorte que seule les commandes qui ont été réellement exécutées par le système de preuve y sont stockées.

Le seul moyen de modifier cette zone est d'utiliser les commandes *Discard*, *Abort* ou *Reset*. Les deux premières agissent sur une preuve en cours alors que la dernière agit sur le

contexte global, supprimant toutes les définitions introduites après une définition donnée (et par effet de bord abandonnant la preuve en cours).

Sur la figure 2.8 on distingue en outre en haut, à droite une fenêtre de menu pour l'édition structurée, à gauche une fenêtre lisp, et en bas à gauche une fenêtre Centaur contenant du code PPML (spécification d'affichage)[62] en cours d'édition.

Point de vue du concepteur

Toute l'interface est basée sur l'environnement Centaur. Coq et Centaur sont deux processus séparés, il y a par ailleurs un troisième processus Coq qui a été spécialisé pour l'analyse syntaxique. Tous ces processus communiquent en utilisant des sockets TCP, avec des protocoles différents selon la direction. Les données venant de Coq sont transférées comme une structure d'arbre selon un protocole développé par Anne Marie Déry et Laurence Rideau [35]. Du coté de l'interface on ne travaille plus que sur l'arbre de syntaxe abstraite. Le formalisme utilisé pour décrire la syntaxe abstraite est Metal[63]. On dispose d'un langage pour manipuler les arbres de syntaxe, le VTP (Virtual Tree Processor) qui est une bibliothèque de fonctions Lisp. Nous l'utilisons abondamment pour construire nos outils de manipulation de script. La syntaxe abstraite peut être décompilée via plusieurs moteurs d'affichage. Ces différents moteurs sont écrit en utilisant un langage de boîtes PPML[62]. L'ensemble du système est écrit en Lisp. Des développements récents montrent que ce travail s'adapte à une implémentation en Java.

Intros	Cette tactique introduit la partie gauche d'une implication, ou les variables liées d'une quantification universelle dans la liste des hypothèses locales.
Elim n et Induction n	Ces tactiques "cassent" le type inductif n et donnent les cas de la preuve par induction sur n.
Simpl	Cette tactique réduit un terme par application des règles de $\beta - \iota$-réduction et de δ-expansion.
Apply t	Cette tactique essaie d'unifier le but courant avec la conclusion du type du terme t. Si cela réussit, elle renvoie les sous-buts corresondants à l'application de l'unificateur aux prémisses du type de t .
Rewrite -> h et Rewrite <- h	Ces tactiques réécrivent le but courant en appliquant la règle de réécriture correspondant à `h` (la flèche donne le sens d'application de la règle).
Exact	Cette tactique permet de fournir explicitement un terme de preuve. `Exact p` résout un but `G` si `G` est convertible avec le type de `p`.
Assumption	Cette tactique essaie de résoudre le but courant en appliquant une des hypothèses du contexte local.
Auto et Trivial	Ces tactiques implémentent les procédures de décisions automatiques à la Prolog. Grossièrement elles essayent d'appliquer récursivement un ensemble de théorèmes précédemment marqués par la commande `Hint`. Elles diffèrent par la profondeur de la recherche.
Cut U	Cette tactique transforme le but courant `G` en deux sous-buts `U->G` et `U`.

FIG. 2.1 – *Descriptif des tactiques les plus utilisées dans ce document*

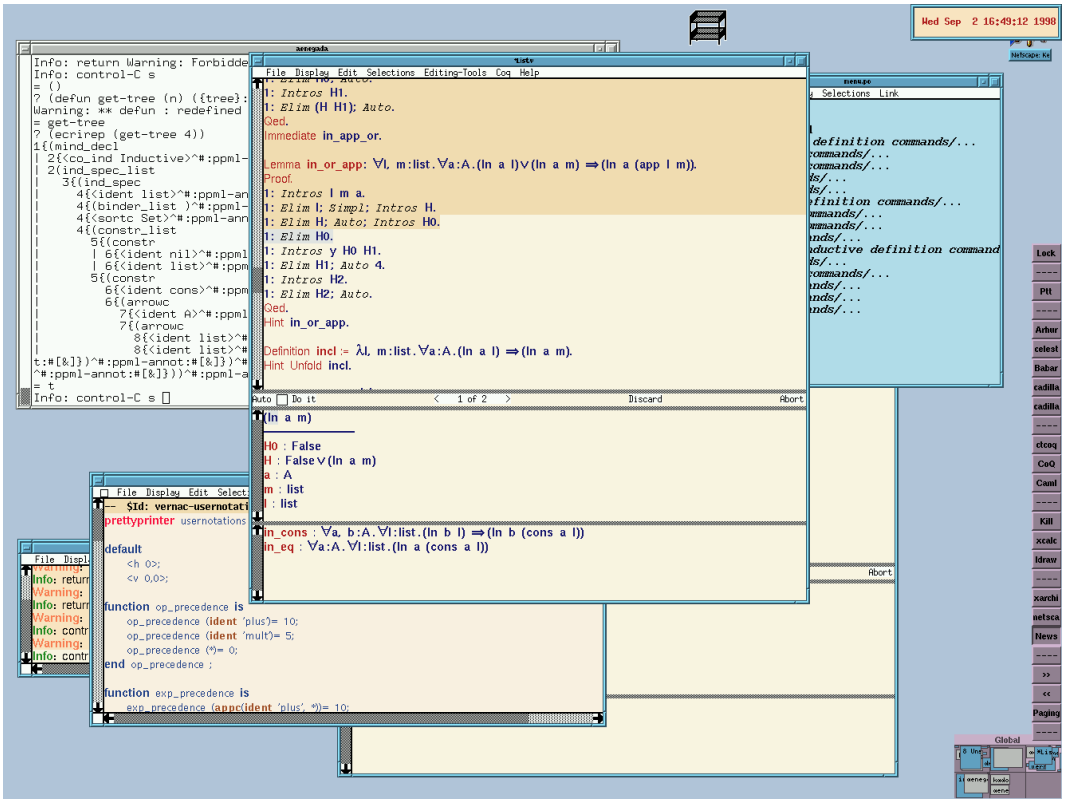


FIG. 2.8 – *Environnement de travail avec CtCoq*

Première partie

Etude locale des théorèmes

Chapitre 3

Compréhension et modification de preuves.

Introduction

La représentation sous forme d'arbre des scripts de preuves permet de bien comprendre les contraintes de base qu'il faut respecter lorsque l'on veut modifier l'ordre des commandes dans un script. Chaque commande s'applique sur exactement un but pour produire un nombre arbitraire de sous-buts et c'est précisément cet aspect qui induit la structure d'arbre. Il n'est pas possible d'appliquer une commande sur un but avant que ce but n'ait été engendré par une commande précédente. Cette contrainte est exprimée de façon naturelle par l'ordre préfixe sur les chemins dans les arbres.

Un script est une représentation en séquence des commandes d'une preuve respectant les contraintes de précédence. Cette mise en séquence est représentée par une fonction de rang qui associe chaque commande à sa position (un entier naturel) dans la séquence.

Un arbre de preuve représente une preuve partielle, au sens où il peut rester des buts à prouver. Dans le mode d'interaction traditionnel de Coq, les buts sont ordonnés dans l'ordre lexicographique pour former une liste et l'on fait référence à un but par son indice dans cette liste plutôt que de donner son chemin dans l'arbre.

Ainsi la notion d'arbre et la notion de chemin disparaissent complètement du mode d'interaction usuel fourni par Coq: les buts comme les commandes sont référencés par leur position dans des listes. Le point clé de notre travail a été de comprendre comment la structure d'arbre permet la mise en œuvre d'outils plus efficaces. Néanmoins, pour faciliter l'intégration de nos outils avec des utilisations conventionnelles de Coq nous maintenons en permanence des correspondances avec la présentation usuelle des scripts de preuve sous forme de séquences de commandes.

Toutefois, ce modèle d'arbre reste une approximation. En général, il n'est pas vrai qu'une tactique n'agit que sur un seul but, comme nous le verrons dans le § 3.5 qui traite

des variables existentielles. Néanmoins, nous verrons que des considérations pragmatiques nous permettent de nous reposer sur la validité de notre modèle *presque partout*.

3.1 Un modèle d'arbre de preuve

Le moyen le plus naturel pour représenter une position dans un arbre est la succession des indices qui décrit le chemin suivi depuis la racine de l'arbre, comme dans une nomenclature à la Dewey.

Un chemin de longueur n est un élement de N_+^n. L'ensemble des chemins sur N_+ est :

$$N_+^* = \cup_{n\in N} N_+^n$$

.

ϵ représente le chemin de longueur nulle (l'unique élément de N_+^0). Par abus de langage on identifiera les entiers positifs et les chemins de longueur 1. La concaténation de deux chemins c_1 et c_2 sera notée $c_1.c_2$.

On définit sur les chemins la relation d'ordre partielle, l'ordre *prefixe* $\prec_p$ par

$$c_1 \prec_p c_2 \leftrightarrow \exists c\ c_2 = c_1.c$$

Nous utiliserons aussi un ordre total sur les chemins, l' ordre *lexicographique* noté $<_l$ et défini par:

1. $\forall c \in N_+^*,\ \epsilon <_l c$
2. $\forall c,c_1 \in N_+^*,\forall i,j \in N_+,\ i < j \Longrightarrow i.c <_l j.c_1$
3. $\forall c,c_1 \in N_+^*,\forall i \in N_+,\ c <_l c_1 \Longrightarrow i.c <_l i.c_1$

On a par exemple:

$$\epsilon <_l 1 <_l 1.1 <_l 1.1.1 <_l \ldots <_l 1.1.2 <_l 1.1.2.1 <_l \ldots <_l 1.2 <_l \ldots <_l 1.3 <_l \ldots <_l 2\ldots$$

$$\epsilon \prec_p 1 \prec_p 1.1 \prec_p 1.1.1 \prec_p 1.1.1.2\ldots$$

mais $(1;1)$ et $(2;1)$ ne sont pas comparables pour $\prec_p$.

Un arbre $\mathcal{T}$ est un ensemble de chemins (donc un élement de $\mathcal{P}(N_+^*)$) vérifiant les propriétés suivantes :

1. $\forall c,c' \in N_+^*\ c \prec_p c' \wedge c' \in \mathcal{T} \Rightarrow c \in \mathcal{T}$
2. $\forall c,c' \in N_+^*\ c.i \in \mathcal{T} \wedge 0 < j < i \Rightarrow c.j \in \mathcal{T}$
3. $\forall c \in \mathcal{T},\exists i \in N,\forall n,\ n > i \rightarrow c.n \notin \mathcal{T}$

Si $\mathcal{T}$ est un arbre, les chemins qui le composent s'appellent les nœuds de l'arbre. Si c est un nœud tel que $c = c_1.i$ on dit que c est le i^{eme} fils de c_1 et que c_1 est le père de c. Un nœud qui n'a pas de fils s'appelle "une feuille".

Un arbre étant donné, on définit sur les nœuds un prédicat de clôture $PC_{\mathcal{T}}$ qui permettra de distinguer deux types de nœuds, les nœuds dit ouverts et les nœuds que l'on qualifiera de clos. $PC_{\mathcal{T}}(c)$ signifie que le nœud c est un nœud clos. En outre, les nœuds ouverts sont des feuilles, ce qui s'exprime par la propriété suivante:

$$\forall \mathcal{T} \in \mathcal{P}(N_+^*), \forall c \in \mathcal{T}, \neg PC_{\mathcal{T}}(c) \Rightarrow \forall i \in N_+ \; c.i \notin \mathcal{T}$$

Pour tout arbre $\mathcal{T}$ on note $\mathcal{T}'$ l'ensemble $\{x \in \mathcal{T} | PC\mathcal{T}(x)\}$ des nœuds clos.

Un nœud ouvert est un nœud auquel aucune tactique n'est associée (c'est- à-dire que le but qui lui est associé n'a pas été attaqué). A tout nœud clos on peut associer un but et la tactique que l'on a appliquée sur ce but. Le nombre de fils d'un nœud clos est le nombre de sous-buts produits par l'application de cette tactique.

On note $\mathcal{TACTIQUE}$ et $\mathcal{BUT}$ respectivement l'ensemble des tactiques et l'ensemble de buts. On se donne les deux opérations suivantes:

- $tactique : \mathcal{T}' \rightarrow \mathcal{TACTIQUE}$
- $but : \mathcal{T} \rightarrow \mathcal{BUT}$

Elles permettent d'associer une tactique à un nœud clos et un but à un nœud quelconque. Lorsque qu'un nœud n est tel que $n \neq \epsilon$ on dit que $but(n)$ est un sous-but, si de plus n est ouvert, on dit que le sous-but est ouvert.

Un script de preuve est un ordonnancement des tactiques d'un arbre de preuve, nous modélisons cela avec la fonction `rg` suivante:

$$rg : \mathcal{T}' \rightarrow [1..Card(\mathcal{T}')]$$

qui vérifie les deux propriétés suivantes:

1. rg est bijective
2. $c \prec_p c' \Rightarrow rg(c) < rg(c')$

La condition 2 met donc en correspondance l'ordre "temporel" (ou chronologique) et l'ordre "logique" entre les commandes.

On notera $<_r$ la relation définie par :

$$c_1 <_r c_2 \Leftrightarrow rg(c_1) < rg(c_2)$$

A partir d'un arbre $\mathcal{T}$ de $\mathcal{P}(N_+^*)$ et rg on peut définir les opérations suivantes[1] :

1. $nfils : \mathcal{T} \rightarrow N$ est définie par :
   $nfils(c) = max\{j | c.j \in \mathcal{T}\}$

2. $index : \mathcal{T}' \rightarrow N$ est définie par :
   $index(b) =$
   $Card\{c.i \in \mathcal{T} | rg(c) < rg(b) \wedge c.i <_l b\} - Card\{c \in \mathcal{T}' | rg(c) < rg(b) \wedge c <_l b\} + 2\}$

Expliquons la formule donnée pour calculer l'index associé à un chemin donné. L'index représente la position d'un nœud, dans la liste des nœuds ouverts, au moment ou la tactique qui lui est associé est jouée. Il s'obtient en ajoutant un au nombre de nœuds ouverts inférieurs, pour l'ordre lexicographique au nœud considéré. Soit `b` le noeud considéré, dans la formule 2 ci-dessus, le premier ensemble dont on prend le cardinal représente tous les nœuds dont les parents ont été joué avant `b` et qui lui sont inférieur pour l'ordre lexicographique. Le second représente tous ceux qui ont été joué avant `b` et qui lui sont inférieur pour ce même ordre. Comme le premier ensemble ne contient pas la racine (qui n'as pas de parents), la différence des cardinaux est inférieure de un au nombre de sous-buts ouverts inférieurs à `b` pour l'ordre lexicographique et par conséquent pour obtenir l'index de `b`, il faut ajouter deux à cette différence.

Pour un arbre de preuve donné, l'index dépend exclusivement du rang, toute modification du rang entraînera donc une modification de l'index.

Définition: 3.1 *On appelle invaraint $rangIndex$ cette relation entre le rang et l'index.*

L'ordre $\prec_p$ permet de caractériser les relations logiques entre les différents sous-buts associés aux nœuds. Si $n_1 \prec_p n_2$ on dira aussi que n_2 dépend de n_1. L'ordre $<_l$ est utile pour ordonner un ensemble de feuilles associées à des sous-buts ouverts.

La fonction rg contient l'historique de l'arbre $\mathcal{T}$ auquel elle est associée, c'est-à-dire qu'elle permet de retrouver par quels autres arbres on est passé dans la construction de $\mathcal{T}$. La fonction $nfils$ permet de connaître le nombre de sous-buts qui ont été générés par l'application d'une tactique. La fonction $index$ indique quelle était, au moment où il a été joué, la position d'un sous-but dans la liste des sous-buts ouverts ordonnée par la relation $<_l$.

Lorsqu'on travaille sur des scripts par édition textuelle, on ne dispose que de la séquence des commandes envoyées au système de preuve et des indices. Ceci correspond aux fonctions suivantes:

- $index' = index \circ rg^{-1} : 1..card(T') \rightarrow N$
- $tactique' = tactique \circ rg^{-1} : 1..card(T') \rightarrow \mathcal{TACTIQUE}$

1. on pourra se reporter à l'exemple 2.4 du chapitre précédent pour avoir une intuition de la notion d'index

On parlera de la tactique de rang i pour $tactique'(i)$.

Dans l'environnement CtCoq où nous avons travaillé, la fonction rg^{-1} est donnée. (c'est-à-dire qu'elle fait partie des informations que le prouveur fournit à l'environnement).

Une autre possibilité consiste à ne fournir qu'une information sur le nombre de fils générés à chaque étape. Pour cela on introduit une fonction: $nfils' : 1..card(T') \rightarrow N$ qui est suffisante pour reconstruire la structure de l'arbre de preuve.

On commence alors par définir par récurrence la fonction
$cheminOuvert : 1..card(T') \rightarrow \mathcal{T}\ list$ définie par:

- $cheminOuvert(1) = (\epsilon)$
- $cheminOuvert(p+1) = (cheminOuvert(p)[1],...,cheminOuvert(p)[index'(p)-1],$
  $cheminOuvert(p)[index'(p)].1,...cheminOuvert(p)[index'(p)].nfils'(p),$
  $cheminOuvert(p)[index'(p)+1]...cheminOuvert(p)[longueur(cheminOuvert(p))])$

$CheminOuvert(i)$ représente la liste des chemins des sous-buts qui étaient ouverts au moment où l'on a joué la tactique de rang i.

On peut alors associer à chaque ligne du script de preuve un chemin de l'arbre de preuve par la fonction: $chemin' : 1..card(T') \rightarrow \mathcal{T}$ définie par :

$$chemin'(p) = cheminOuvert(p)[index'(p)]$$

A ce stade, nous avons une bijection entre le script et l'arbre de preuve. Nous allons utiliser la structure d'arbre que nous venons introduire pour définir des transformations du script. Mais avant cela, nous proposons différents outils qui fournissent à l'utilisateur une meilleure compréhension de la structure de ses preuves.

3.2 Visualisation de l'arbre et navigation dans le script

Comme nous l'avons mentionné au début de ce chapitre, (page 27), les notions d'arbre et de chemin n'apparaissent pas dans le mode d'interaction usuel de Coq. L'utilisateur ne manipule qu'un script de preuve, (l'arbre de preuve est seulement une structure interne). Il est donc nécessaire de fournir des outils pour permettre à l'utilisateur de retrouver, d'analyser et de comprendre la structure arborescente de sa preuve.

Pour cela, on a principalement deux possibilités. La première consiste à afficher l'arbre de preuve à l'aide d'un programme de dessin d'arbres ou de graphes. La seconde consiste à mettre en valeur la structure dans la représentation textuelle (linéaire) du script. Ces deux méthodes connaissent de nombreuses variantes et optimisations.

Les outils de navigation dans la représentation textuelle que nous décrivons au § 3.2 font maintenant partie de la distribution standard de CtCoq. Les autres solutions n'ont fait l'objet que de prototypes.

Visualisation graphique de l'arbre de preuve

Nous avons mené quelques expériences de visualisation de l'arbre de preuve à l'aide de logiciels de dessin de graphes[2].

Remarquons que pour la visualisation des arbres, la plupart des logiciels de dessin de graphe conviennent mais dans la seconde partie de cette thèse nous aurons aussi à visualiser des graphes orientés. Bien que presque tous ces logiciels utilisent l'algorithme de Sugiyama [130, 129], les résultats des différents systèmes dans la visualisation de gros graphes sont très inégaux (certains ne supportent pas plus de quelques dizaines de nœuds, d'autres permettent de visualiser un graphe de plusieurs centaines de nœuds).

Après une première expérience dans le système Centaur en utilisant le serveur de graphe standard de Centaur [68, 74] et des essais sur différents systèmes, nous avons choisi d'utiliser **daVinci** [52, 51], un logiciel de visualisation de graphes développé à l'université de Brême.

Nous avons interfacé directement Coq (donc en pratique ML) avec daVinci, la communication entre les deux se faisant par pipe. Une description complète de cette interface ML/daVinci est donnée en annexe.

La première question qui se pose est de savoir ce que l'on désire visualiser. On peut décider de ne faire apparaître que les nœuds clos. On donne alors à chaque nœud une étiquette contenant le texte de la tactique. On dit alors que l'on visualise l'arbre de tactique.

La figure 3.1 montre l'arbre de tactique de la preuve incomplète d'une propriété de la β-réduction:
`Lemma red_abs : (M,M':lambda)(red M M') -> (red (Abs M) (Abs M')).`

Dans le cas des preuves incomplètes, la visualisation des seuls nœuds clos présente cependant l'inconvénient majeur de ne pas être stable dans le temps. Ainsi, si une tactique *tac* a généré trois sous-buts et qu'à l'étape `t`, seuls les deux derniers ont été attaqués par des tactiques tac_1 et tac_2 (qui sont donc visualisées), l'utilisateur ne sait pas que les deux nœuds qu'il voit ne sont pas les seuls fils de *tac* et encore moins que ce ne sont pas les deux premiers fils de *tac*. Si à l'étape suivante (`t+1`) c'est le premier sous-but produit par *tac* qui est attaqué, la représentation change et un nouveau fils est inséré à gauche des deux autres nœuds. Ainsi si on visualise figure 3.2 l'arbre de tactique de la preuve complète du lemme `red_abs` on constate qu'une branche a été insérée pour résoudre le second sous-but généré par la tactique `Induction.`

Pour éviter ce genre de problème on peut représenter en plus les nœuds ouverts. Ainsi, à chaque étape de preuve, seul le statut (clos ou ouvert) change, mais en aucun cas son arité. Pour une preuve complète ces deux représentations sont évidemment isomorphes.

Mais quand on manipule de grosses preuves, l'idéal est de visualiser l'arbre de preuve décrit dans la section 3.1. Plutôt que d'étiqueter les nœuds avec le texte des tactiques ou

2. Le lecteur interessé par le dessin de graphe pourra se reporter à l'URL suivante http://www.cs.uni-sb.de/RW/users/sander/html/gstools.html

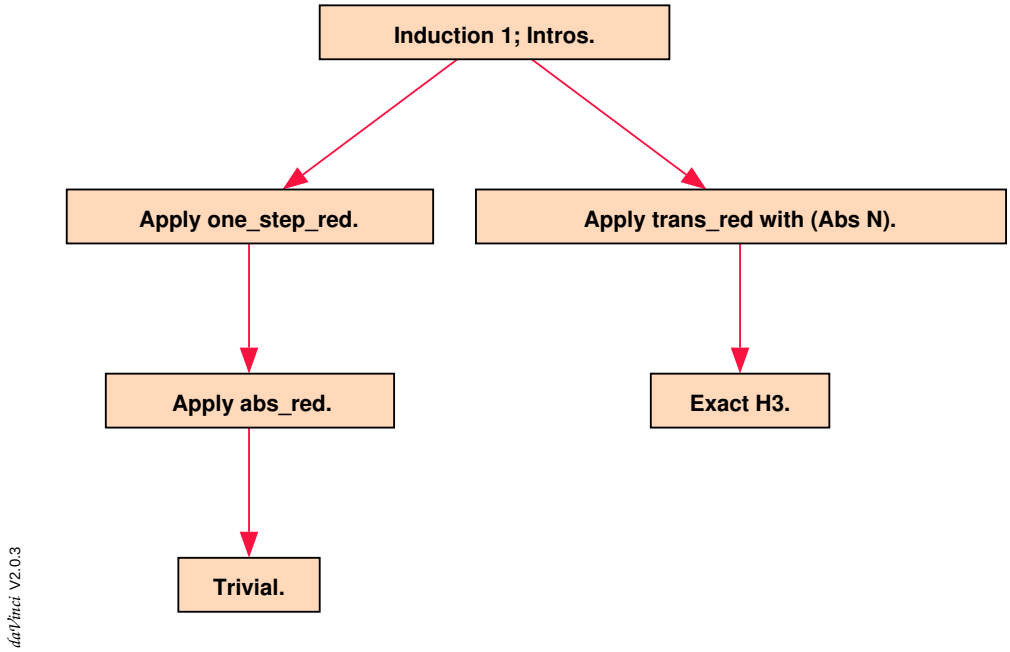


FIG. 3.1 – *Représentation graphique de l'arbre de tactique d'une preuve incomplète*

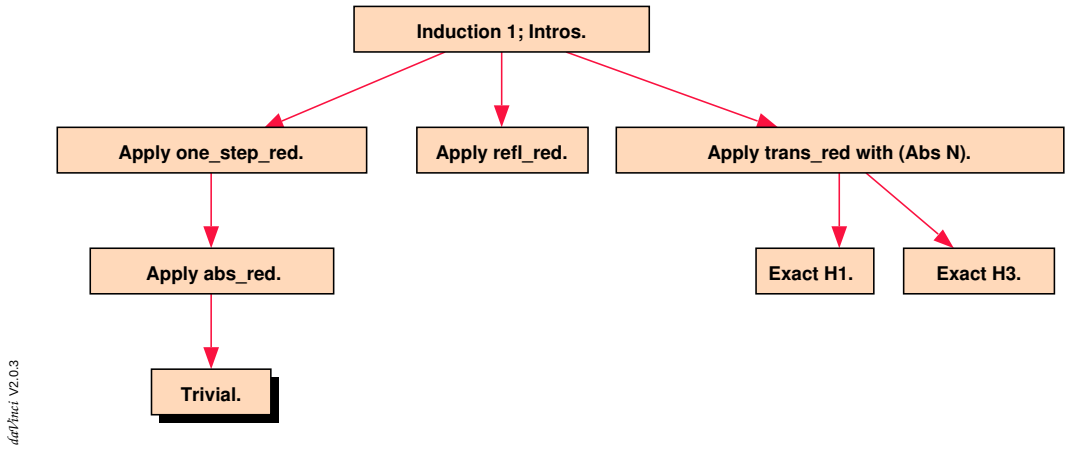


FIG. 3.2 – *Représentation graphique de l'arbre de tactique d'une preuve complète*

des buts qui peut être très volumineux, on différencie simplement les nœuds ouverts des nœuds clos par la couleur ou la forme des nœuds. Il est possible de sélectionner un nœud pour visualiser le but associé et si le nœud est clos la tactique associée.

Dans un environnement de preuve permettant la visualisation de l'arbre, il est également important que la sélection sur le graphique d'un nœud auquel est associée une tactique entraîne la sélection de la tactique correspondante dans la fenêtre d'édition. Cela devra être mis en œuvre lors de l'intégration dans l'environnement CtCoq du prototype que nous avons réalisé.

Des outils de visualisation des arbres de preuve existent pour d'autres systèmes. Dans l'environnement Centaur une première expérience de visualisation des arbres de preuve en utilisant le serveur de graphe [68, 74] avait été esquissée pour Lego par Pierre Valentin [138]. Les nœuds ne portaient pas de d'étiquette mais pouvaient être mis en correspondance avec les tactiques qui leur étaient associées dans la fenêtre d'édition.

L'interface standard (Emacs) de PVS [123, 117, 93, 92] utilise TCL/TK [91] pour visualiser les arbres de preuve. L'affichage d'une preuve (commande `M-X x-show-proof`) produit un arbre dont les nœuds sont visualisés par le symbole $\vdash$. Cliquer sur l'un des nœuds affiche le séquent qui lui est associé.

Les différentes interfaces graphiques d'HOL, de Kiv ou d'Isabelle [94, 36] permettent aussi de visualiser l'arbre de preuve.

Plusieurs interfaces utilisent daVinci. Citons celles de Kiv, du démonstrateur de théorèmes Ergo [137], ou les expériences autour de HOL90 qui furent menées par Ralf Reetz [111]. Ce dernier a implémenté une interface entre daVinci et SML/NJ 0.93 [6] (malheureusement pas maintenue lors du passage aux versions suivantes de SML) et il l'utilise pour visualiser les arbres de preuve mais aussi les dépendances entre théories ou entre objets d'une théorie comme nous le faisons dans la seconde partie de cette thèse.

Pour les systèmes plus éloignés de la tradition LCF, on peut aussi citer l'interface graphique distribuée d'Omega [124].

Tous ces systèmes proposent une visualisation complète de l'arbre de preuve. C'est-à-dire qu'ils représentent tous les nœuds de cet arbre. Comme nous allons le voir cela est assez peu utilisable sur de grosses preuves.

Problèmes de croissance. Le problème majeur de la visualisation complète de l'arbre de preuve est que cet arbre croit très rapidement.

On peut assez facilement réduire la croissance verticale en regroupant "les enfants uniques" dans le même nœud que leurs parents, la représentation graphique ne sert alors qu'à mettre en lumière les embranchements de l'arbre de preuve. On dit alors que l'on représente l'arbre de choix. Cette opération de regroupement de nœuds peut n'être qu'un artifice de visualisation ou être réellement effectuée sur la preuve, on parle alors de contraction de preuve. Nous développons cette notion ainsi que sa réciproque, l'expansion de preuve au chapitre suivant.

Quelques statistiques sur les scripts de preuve de la bibliothèque Coq nous ont permis d'estimer que moins de 10% des nœuds sont des nœuds de choix. Le regroupement des tactiques ne générant qu'un seul sous-but réduit donc réellement la croissance verticale de l'arbre de preuve.

Par contre il est beaucoup moins évident de limiter la croissance horizontale de l'arbre. Cette limitation apparait clairement, dans les preuves où l'on fait du raisonnement par cas ou par induction sur des objets définis inductivement. Par exemple lorsque l'on fait des preuves sur la sémantique des langages de programmation où les grammaires sont définies par des types inductifs ayant de nombreux constructeurs, certaines tactiques engendrent un grand nombre de sous-buts.

Visualisation textuelle

La visualisation de la structure sur le script de preuve pose aussi des problèmes que nous allons décrire maintenant.

Indentation. Intuitivement pour représenter la structure d'arbre sur le texte correspondant, on peut indenter chaque ligne du script en fonction de sa profondeur dans l'arbre de preuve. En pratique les choses ne sont pas si simples.

Problème du peigne. Comme nous l'avons mentionné dans notre présentation, l'utilisateur ne construit pas forcément son script linéairement. En variant régulièrement les sous-buts qu'il attaque il peut obtenir un script comme celui de la figure 3.3

```
L1  Intros n m p.
L2  Elim n.
L3  2:Intros.
L4  1:Simpl.
L5  2:Simpl
L6  2:Rewrite ->H.
L7  1:Apply refl_equal.
L8  1:Apply refl_equal.
```

FIG. 3.3 – *Un script de preuve de l'associativité à gauche de l'addition*

Une simple indentation des lignes de ce script en fonction de leur position dans l'arbre de preuve conduit à une structure en forme de peigne comme celle de la figure 3.4 ou il est difficile d'identifier les relations entre les différentes tactiques.

Donc, dès que le script de preuve ne correspond pas à un parcours de l'arbre de preuve en profondeur d'abord et de la gauche vers la droite, l'indentation interactive du script devient sans intérêt si l'on ne modifie pas au préalable la structure de ce script. Mais modifier la structure du script à la volée ne semble pas non plus une solution viable. D'une part cela modifie constamment la vision que l'utilisateur a de son script en supprimant

```
L1      Intros n m p.
L2          Elim n.
L3              2:Intros.
L4                  2:Simpl
L5              1:Simpl.
L6                      2:Rewrite ->H.
L7                  1:Apply refl_equal.
L8                          1:Apply refl_equal.
```

FIG. 3.4 – *Un script indenté "en peigne"*

toute trace de la stratégie suivie, d'autre part cela oblige à de nombreux parcours du script pour insérer chaque nouvelle commande en bonne position. Mais surtout cela n'est pas toujours possible comme nous le verrons aux §3.5 à cause des contraintes liées à la présence de variables existentielles.

Enfin l'indentation conduit au même genre de problème de croissance que dans le cas des représentations graphiques, qui se manifeste par un décalage trop fort vers la droite.

Nous fournissons néanmoins la possibilité, **une fois la preuve finie** de réorganiser le script. Ce que l'on a appelé "le problème du peigne" disparaît et le script peut alors être indenté. Comme dans le cas de la visualisation graphique, nous en limitons la croissance horizontale tout en mettant en évidence les embranchements de l'arbre de preuve.

L'idée est que dès qu'une tactique génère plusieurs fils, ceux-ci sont décalés, et dès qu'une sous-preuve est complète on saute une ligne; ainsi seuls les embranchements sont mis en évidence et la taille du script ne croît pas trop vite en largeur. Les scripts des figure 2.2 et 3.3 se réecrivent tous les deux comme illustré sur la figure 3.5.

```
L1      Intros n m p.
L2      Elim n.
L3          Simpl.
L4          Apply refl_equal.

L5          Intros.
L6          Simpl.
L7          Rewrite ->H.
L8          Apply refl_equal.
```

FIG. 3.5 – *Script "restructuré"*

Il faut noter que ce réordonnancement d'un script de preuve n'est valide que dans un cadre sans contrainte, c'est-à-dire si l'utilisateur n'a pas manipulé de variables existentielles dans la preuve initiale.

Nos autres outils d'aide à la compréhension de la preuve, ne modifient pas la structure du script en cours de développement mais l'instrumentent pour permettre d'y simuler la navigation dans l'arbre de preuve.

Navigation

Exemple de navigation dans un script de preuve Sur notre exemple figure 3.3, on peut sélectionner la tactique `Apply refl_equal` à la ligne `7` et retrouver quelle tactique a généré le but qui est abordé par la tactique sélectionnée (c'est-à-dire, retrouver son **père** dans l'arbre de tactiques). Dans notre cas, cela montrera la tactique `Simpl` de la ligne `4`. On peut aussi vouloir savoir quels sont les **frères** de cette dernière tactique, la fonctionnalité adéquate nous montrera la tactique `Intros` de la ligne `3`. On pourra demander à voir son premier **fils** ce qui nous donnera la tactique `Simpl` de la ligne `5`. On peut enfin demander à voir **toute la descendance**, ce qui pour la ligne `5` nous montrera les lignes `5,6` et `8`.

Dans l'environnement CtCoq cette navigation s'effectue par de simples combinaisons de touches qui correspondent aux différentes actions possibles: retrouver le père, les enfants, toute une descendance, les frères etc ...Le résultat est visualisé par un changement de couleur des lignes répondant à la question. Sur la figure 3.6 suivante on a demandé toute la descendance de la ligne 4, les lignes 4 et 7 sont donc passées en gris.

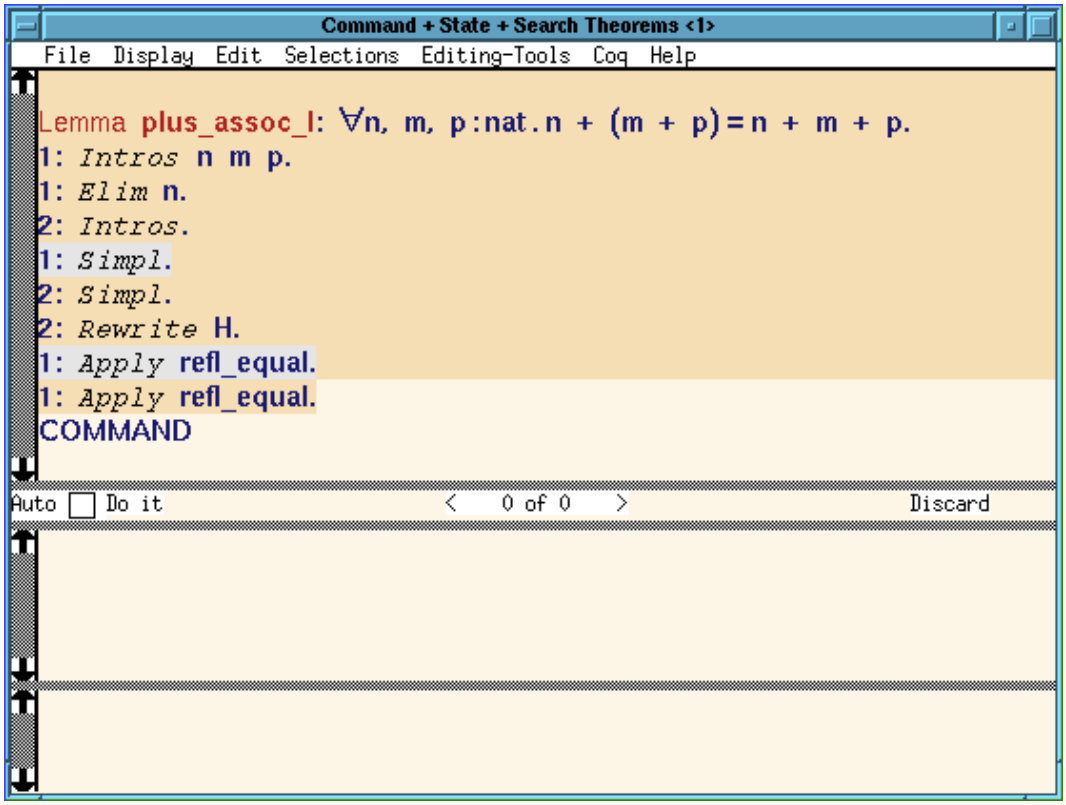


FIG. 3.6 – *Exemple de navigation dans un script*

Enfin dès qu'une tactique du script est sélectionnée, il est possible de demander l'affichage du sous-but qu'elle a attaqué. Ainsi sur la figure 3.7 on a demandé à voir quel sous-but avait été attaqué par la tactique `Simpl` de la ligne 4.

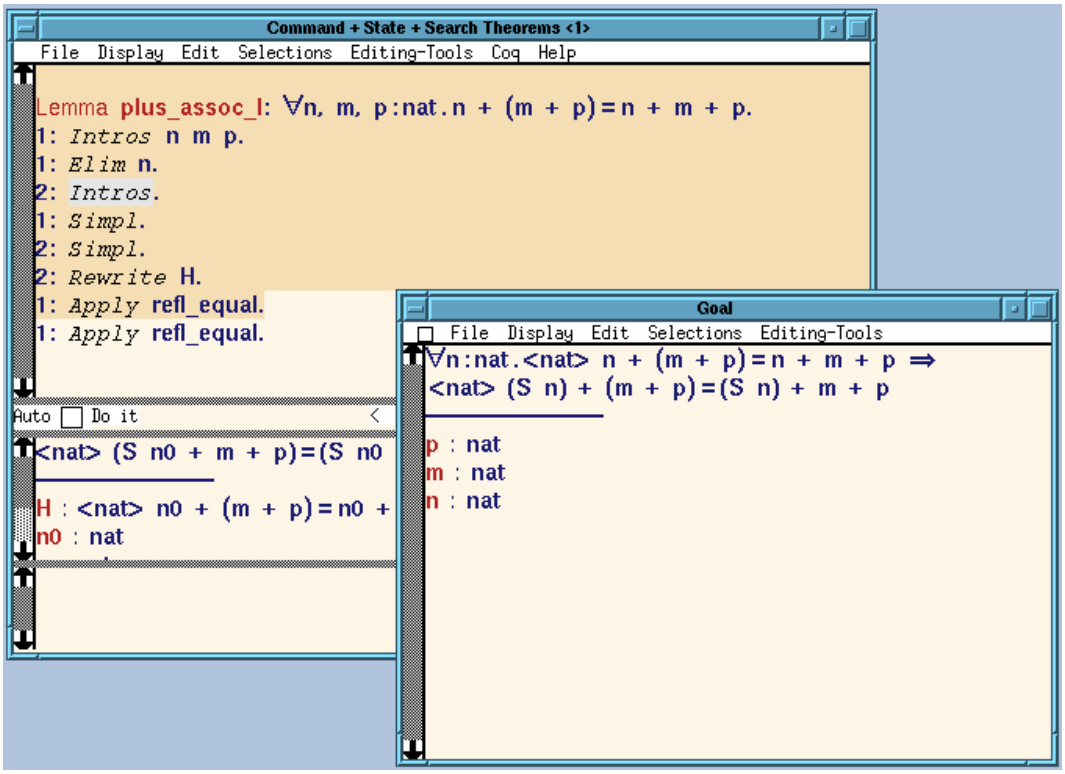


FIG. 3.7 – *Liaison tactique/sous-but*

La conjugaison de ces deux possibilités améliore réellement la compréhension d'une démonstration en cours.

3.2.1 La relation sous-but/tactique

Dès que l'on a une interface graphique, donc en particulier en CtCoq, on dispose d'une fenêtre d'édition dans laquelle sont entrées les commandes et d'une fenêtre d'"état" dans laquelle s'affiche le résultat de ces commandes.

Il faut pouvoir sélectionner un but donné dans la fenêtre d'état et retrouver dans la fenêtre d'édition la tactique qui l'a généré.

Dans l'exemple de la figure 3.8, il y a deux sous-buts, on a sélectionné le second sous-but dans la fenêtre d'état(fenêtre inférieure) et la sélection de la fenêtre d'édition (fenêtre supérieure) a été déplacée sur la tactique `Split` qui a generé ce second sous-but.

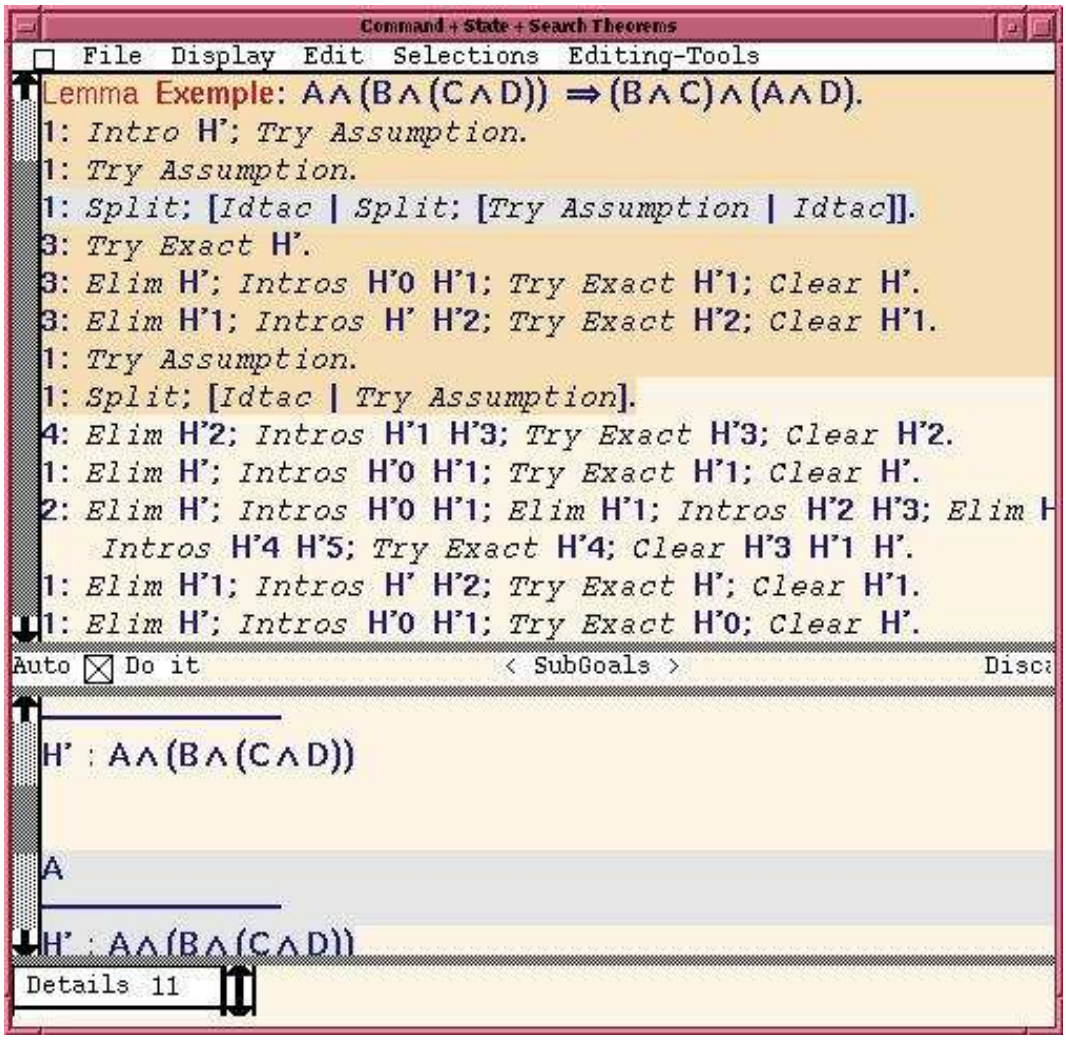


FIG. 3.8 – *Relation entre les fenêtres d'édition et d'état*

3.3 Mécanisme de retour-arrière

Comme celui de la plupart des prouveurs, le système de retour en arrière (commande `Undo`) de Coq n'utilise que les dépendances historiques entre les tactiques (une tactique t_1 dépend historiquement d'une tactique t_2 si elle a été jouée postérieurement à t_2 c'est-à-dire si $t_2 <_r t_1$). Le script est considéré comme une pile de commandes et, pour effacer une tactique donnée il faut tout dépiler jusqu'à la tactique à effacer, y compris les parties de script correspondant à des sous-arbres de preuve logiquement indépendants de cette tactique (c'est-à-dire qui ne sont pas comparables pour l'ordre prefixe $\prec_p$).

Ainsi si l'on désire effacer la tactique `Simpl` à la ligne 4 du script de la figure 3.3 on depilera 4 fois l'état courant (`Undo 4`) et on ne gardera que le script de la (figure 3.9):

L1	Intros n m p.
L2	Elim n.
L3	2:Intros.

FIG. 3.9 – *Après un undo classique*

Toute une partie de la preuve, pourtant sans lien logique avec la partie que l'on désirait effacer, a été perdue. De plus maintenir une pile des états a un coût élevé. Par conséquent le nombre d'états stockés, et donc le nombre de "Undo" possibles est borné. Cela n'est guère satisfaisant quand on manipule de grosses preuves.

Nous avons implémenté un mécanisme de retour-arrière qui permet d'effacer n'importe quelle partie de la preuve en tenant compte des dépendances logiques entre les sous-preuves et donc de ne pas modifier les parties du script qui s'appliquent à des sous-buts indépendants. De plus nous maintenons la cohérence de la partie "effacée" qui pourra donc être récupérée et rejouée immédiatement. Nous appelons "Undo-Redo" la commande implémentant ce mécanisme.

3.3.1 La commande "Undo-Redo"

Après application de la commande "Undo-Redo" sur la ligne 4 du script de la figure 3.3 on obtient le script de la figure 3.10.

L1	Intros n m p.
L2	Elim n.
L3	2:Intros.
L4	2:Simpl
L5	2:Rewrite ->H.
L6	2:Apply refl_equal.
L7	1:Simpl.
L8	1:Apply refl_equal.

FIG. 3.10 – *Un script après application de la commande "Undo-Redo"*

Le script reste cohérent. Dans la partie du script qui est conservée, tous les indices indiquant à quel sous-but s'applique la tactique ont été remis à jour. La partie effacée (c'est à dire l'ensemble des dépendances de la tactique sélectionnée) est réécrite, de façon distincte, à la fin de la partie du script qui est conservée; elle peut a priori être modifiée mais les indices sont néanmoins remis à jour pour permettre à l'utilisateur qui le voudrait de rejouer immédiatement cette partie de preuve.

Dans l'interface CtCoq, la partie conservée reste dans la "zone finale" qui ne peut être modifiée tandis que la partie effacée a été déplacée en tête de la "zone normale" où elle peut être modifiée.

On définit maintenant dans notre modèle l'opération qui est à la base du retour arrière logique. Du point de vue de la gestion d'un script, le *retour arrière logique* consiste simplement à déplacer à la fin du script l' ensemble des tactiques associées à une branche de l'arbre de preuve. Dans notre modèle cela se traduit donc par une modification de la fonction de rang.

La fonction undo sur un nœud c peut donc se définir comme suit:

$$undo((\mathcal{T},P,rg),c) = (\mathcal{T},P,rg')$$

La fonction rg' founit un nouvelle ordonnancements des commandes.

Pour $\mathcal{T}$,P,et c fixés $rg'\mathcal{T}' \rightarrow [1..Card(\mathcal{T}')]$ est définie par:
$rg' = \lambda c_1.if\ c \prec_p c_1$

$$then\ Card(\mathcal{T}') - Card(\{c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) > rg(c_1)\}$$

$$else\ rg(c_1) - Card(c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) < rg(c_1)\}$$

Cette fonction est toujours une fonction de rang, c'est-à-dire qu'elle vérifie les propriétés 1 et 2 de la page 29 (la preuve qu'elle est bijective se fait par un raisonnement par l'absurde).

La fonction *index* est implicitement modifiée par le changement du rang. Comme nous l'expliquons au paragraphe suivant, dans l'implémentation du processus de retour-arrière, les index devront être physiquement modifiés dans le script.

3.3.2 Problèmes de cohérence

Le problème du retour-arrière logique est la conservation de la cohérence du script. En effet si l'on se contente d'effacer du script l'ensemble des tactiques correspondant à la branche de l'arbre de preuve que l'on désire couper, le script restant peut devenir incohérent et ne pourra alors pas être rejoué lors d'une prochaine session de preuve.

La figure 3.11 donne une exemple de script incohérent. Les tactiques des lignes 3, 4, 5 et 6 ne s'appliquent plus au bons sous-buts.

Pour garder la cohérence, une fois les tactiques deplacées (c'est-à-dire une fois les rangs modifiés) il faut éventuellement modifier les numéros indiquant à quel sous-but s'applique chaque tactique, c'est-à-dire rétablir l'invariant $rangIndex$ de la definition 3.1 page 30.

En fait, on doit d'un coté, garder la cohérence du script restant pour qu'il puisse être rejoué dans une prochaine session, et de l'autre, restituer la cohérence de la partie effacée afin de pouvoir éventuellement la rejouer dans la session courante. Ce principe de Undo-Redo est à rapprocher du modèle US&R de Vitter [139]. Dans son modèle de Undo-Redo pour éditeur de texte, tout ce qui est défait est stocké dans une pile et il est possible de le rejouer en sautant éventuellement certains des états stockés.

Nous détaillons maintenant le fonctionnement du retour arrière logique.

L1	1:Intros n m p.
L2	1:Elim n.
L3	1:Intros.
L4	1:Simpl
L5	1:Rewrite ->H.
L6	1:Apply refl_equal.
	———————————
L7	1:Simpl.
L8	1:Apply refl_equal.

FIG. 3.11 – *Un script devenu incohérent*

3.3.3 Fonctionnement du retour arrière logique.

On récupère la ligne de script sélectionnée puis on calcule sa liste de dépendances en utilisant l'ordre partiel $\prec_p$ (t_1 dépend de t_2 si $chemin(t_1) \prec_p chemin(t_2)$). Ensuite, on va déplacer toutes les tactiques de cette liste à la fin du script que l'on a déjà joué. Puis on parcourt le script ainsi obtenu pour modifier chaque index de manière à rétablir l'invariant $rangIndex$.

En pratique l'implantation de la mise à jour des index n'utilise pas les fonctions $index$ et $index'$ qui, basées sur la notion ensembliste de cardinal, auraient un coût trop élevé.

Les mises à jours sur le script étant achevées, on modifie la structure interne manipulée par le démonstrateur, pour mettre le système et le script en conformité. Notons que dans un système où il ne serait pas possible de modifier simplement l'état de la structure interne du démonstrateur on a toujours la possibilité de défaire la preuve complète et rejouer la partie du script que l'on veut conserver. Ces deux opérations sont bien sûr faites de manière transparente à l'utilisateur. Le coût à payer est alors un temps d'attente qui est proportionel à la partie du script à rejouer.

La figure 3.12 montre l'utilisation du Undo-Redo dans l'environnement CtCoq. Les tactiques effacées ont été déplacées à la fin du script, dans la *zone d'attente*, et sont affichées dans une couleur particulière pour les différencier des tactiques qui attendent d'être envoyées au système de preuve. Tous les index ont été mis à jour, elles peuvent donc être rejouées après d'éventuelles modifications. De même, les index des tactiques contenues dans la zone finale ont été modifiés pour rétablir la cohérence. Cette partie du script pourra donc être rejouée lors d'une nouvelle session.

3.4 Aperçu de l'implémentation

3.4.1 Instrumentation du script

On utilise une structure de données arborescente pour représenter la plupart de nos données. Ainsi les scripts, buts, formules logiques ou chemins dans l'arbre de preuve, ont

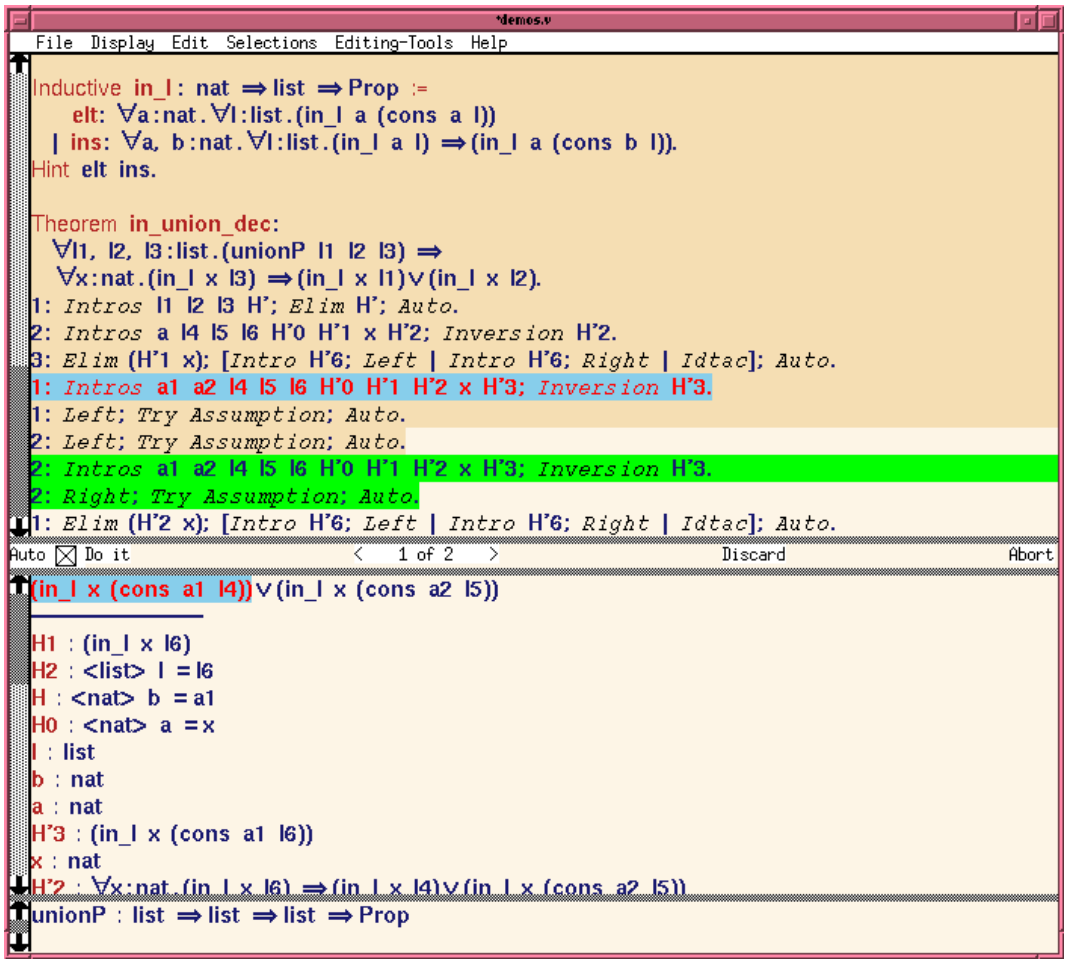


FIG. 3.12 – *L'environement CtCoq après un Undo-Redo*

tous une structure arborescente. Le manipulateur d'arbres (Virtual Tree Processor [64]) que nous utilisons fournit une structure d'arbre doublement chaînée, de telle sorte que l'accès au père d'un terme donné se fait en temps constant. Les nœuds peuvent porter des annotations qui peuvent être utilisées par le système mais ne sont ni utilisables ni même visibles par l'utilisateur.

Pour assurer la correspondance entre la représentation linéaire du script de preuve et l'arbre de preuve, chaque ligne de script exécutée est annotée par le chemin auquel elle correspond dans l'arbre de preuve (l'annotation est une liste d'entiers), par le nombre de but(s) qu'elle a généré, et par un nom de théorème qui indique à quelle preuve se rattache cette ligne de script. Dans ces conditions, certaines opérations, comme rechercher la tactique associée à un chemin donné, ont une complexité temporelle en $O(n)$ (où n est le nombre de lignes du script). Pour remédier à cela, nous utilisons une table de hachage pour associer à un chemin dans l'arbre de preuve la tactique correspondante.

Un script contient généralement plusieurs preuves. Nous avons aussi une table de hachage pour chacune des preuves du script. Ces tables sont elles-mêmes conservées dans une autre table utilisant comme clef les noms de théorèmes. Cette table globale est également accrochée en annotation du script complet.

3.4.2 Interaction avec le prouveur

La structure d'arbre que nous maintenons dans l'environnement est un miroir simplifié d'une structure similaire maintenue par le système de preuve. Chaque but reçu du système est annoté par son chemin. Pour être en mesure d'effacer une branche donnée, nous avons ajouté au système Coq une commande pour supprimer la sous-preuve d'un sous-but référencé par son chemin dans l'arbre de preuve.

D'autre part l'interface utilisateur maintient la liste des sous-buts ouverts. Lorsqu'on applique une tactique, son préfixe numérique (1 par défaut) permet de savoir quel sous-but elle concerne. Le système Coq répond à l'envoi d'une tactique en renvoyant seulement les nouveaux sous-buts. Les chemins correspondant à ces sous-buts, sont insérés dans la liste de chemins, et l'ancien chemin est utilisé pour annoter la commande que l'on vient d'exécuter. On ajoute aussi les annotations correspondant au nombre de buts générés, (c'est le nombre de sous-buts reçus) et au nom du théorème. Ce mécanisme est implémenté dans CtCoq indépendamment de mon travail.

3.4.3 Gestion de la mémoire

Les tables de hachage ont un coût mémoire qui doit être pris en compte. Quand la commande `Reset` est utilisée, cela peut conduire à effacer de nombreuses preuves de l'environnement. Nous interceptons donc l'événement correspondant à l'envoi d'un `Reset` de manière à supprimer de la table globale toutes les tables correspondant à des preuves effacées.

De plus, comme nous l'avons dit, l'interface CtCoq permet de travailler sur plusieurs preuves en même temps, en utilisant plusieurs fenêtres d'édition, chacune contenant son propre script. Cette possibilité interfère très peu avec notre mécanisme d'effacement. Néanmoins, une interface multi-fenêtre doit au moins fournir la possibilité d'ouvrir et de fermer chacune des fenêtres séparément. Quand on ferme une fenêtre, il est très important de récupérer la mémoire correspondant aux données qu'elle contenait. Ainsi l'espace mémoire correspondant aux différentes tables de hachage doit être récupéré. Notre choix d'attacher la table globale comme une annotation sur la racine du script rend cela complètement transparent: quand une fenêtre est fermée, le script qu'elle contient est effacé et par conséquent les annotations qu'il porte disparaissent.

3.5 Arbre de preuve et variables existentielles

Les outils que nous venons de décrire font implicitement l'hypothèse que les différentes branches de l'arbre de preuve sont indépendantes entre elles. Nous allons voir maintenant qu'en présence de variables existentielles cela n'est pas toujours le cas. Nous montrons que cela complique le mécanisme de retour-arrière logique et nous proposons des solutions. Mais voyons d'abord succinctement l'intérêt des variables existentielles. Les exemples sont donnés en Coq. Le § 3.5.2 page 47 rappelle brièvement comment on manipule ces variables dans ce système.

3.5.1 Pourquoi utiliser des variables existentielles ?

Une utilisation classique est liée à l'introduction de quantificateurs. Lors de la résolution de $\exists x P(x)$, pour introduire le quantificateur existentiel, on doit donner un témoin de la propriété `P`. C'est-à-dire que l'on doit connaître la solution avant de résoudre le problème. L'utilisation de meta-variables permet de retarder l'instanciation du témoin. Par exemple pour résoudre $\exists x \ .x + 2 = 5$ sans variable existentielle on instancie x par 3 et on prouve que $3 + 2 = 5$. Avec les variables existentielles, on introduit une variable X et on essaie de prouver $X + 2 = 5$, et par réduction, on obtient $X = 3$ qui nous permet d'instancier X.

Nous reprenons cet exemple en Coq pour illustrer la manipulation des variables existentielles:

```
Lemma equation : (Ex [n:nat]  ((plus n (2))=(5))).
```

On élimine le quantificateur existentiel en introduisant une variable existentielle:

```
equation < EApply ex_intro.
1 subgoal

  ============================
   (plus ?77 (2))=(5)
```

Pour l'instant la seule contrainte sur la variable existentielle `?77` est qu'elle doit être de type `nat`. Elle peut etre visualisé par la commande `Show Existentials.`

```
equation < Show Existentials.
Existential 1 = ?77 : [[]  |- nat]
```

On continue la démonstration par quelques étapes de réécriture[3]

```
equation < Do 2 Rewrite <-plus_n_Sm.
1 subgoal

  ============================
   (S (S (plus ?77 (0))))=(5)

equation < Do 2 Rewrite <- plus_n_O.
1 subgoal

  ============================
   (S (S ?77))=(5)

equation < Do 2 Apply eq_S.
1 subgoal

    ============================
   ?77=(3)
```

On instancie maintenant la variable existentielle ?77 par 3. Elle n'est pas pour autant remplacée par cette valeur dans le but restant mais une contrainte à été introduite: ?77 ne pourra être unifiée qu'avec un terme convertible avec la valeur introduite (ici 3).

```
equation < Instantiate 1 (3).
1 subgoal

    ============================
   ?77=(3)
```

3. Les théorèmes `plus_n_Sm` et `plus_n_O` correspondent respectivement aux égalités `(n,m:nat)(S (plus n m))=(plus n (S m))` et `(n:nat)n=(plus n (0))`, `eq_S` exprime la propriété `(n,m:nat)n=m->(S n)=(S m)` .

On unifie ?77 et la valeur introduite par la contrainte. (remarquons que `NormEvars` se contente de faire un `Change` du terme dans lequel la valeur de la contrainte a été substituée à la variable existentielle. (ici `Change (3)=(3)`)

```
equation < NormEvars.
1 subgoal

  H : (n,m:nat)(S (plus n m))=(plus n (S m))
  H0 : (n:nat)n=(plus n (0))
  ============================
   (3)=(3)

equation < Apply refl_equal.
Subtree proved!
```

Plus généralement on introduira des variables existentielles chaque fois que l'on veut retarder (ou contourner) l'obligation de fournir un témoin.

3.5.2 Manipulation des variables existentielles en Coq

Les tactiques permettant d'introduire et de manipuler des variables existentielles en Coq sont les suivantes:

- `EApply` (généralisation de `Apply` pouvant introduire des variables existentielles).
- `EAuto` (généralisation de `Auto` travaillant sur des sous-buts contenant des variables existentielles).
- `EExact` (généralisation de `Exact` travaillant sur des sous-buts contenant des variables existentielles).
- `Instantiate n term` (essaie d'instancier la n-ième variable existentielle avec `term`).
- `NormEvars` (remplace les variables contraintes par leurs valeurs).

Dans la section suivante on va s'intéresser à l'influence des variables existentielles sur l'indépendance des branches de l'arbre de preuve et à l'interaction entre le mécanisme de retour-arrière logique décrit précédemment et la présence de variables existentielles. On verra que pour fonctionner en leur présence le mécanisme de retour-arrière doit être modifié.

3.5.3 Influence des variables existentielles sur l'indépendance des branches de l'arbre de preuve

En présence de variables existentielles les différentes branches de l'arbre de preuve ne sont plus forcément indépendantes comme l'illustre l'exemple qui suit.

```
Lemma lemme1 : (a,b,c,d:nat)(a=b)->(b=c)->(a=d)->a=c.
```

FIG. 3.13 – *Le lemme1*

Considerons le lemme suivant:

On applique la tactique : `Intros;EApply trans_equal`[4]. Cela produit deux sous-buts, et introduit une variable existentielle dénotée par "?". On peut représenter l'état de la preuve par l'arbre suivant:

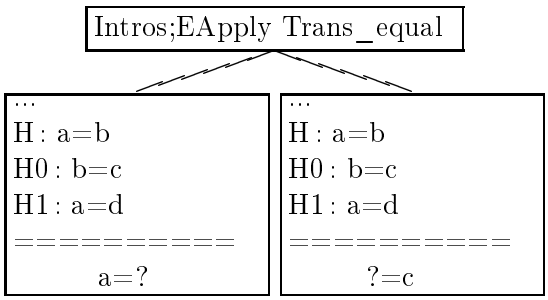


FIG. 3.14 – *L'arbre de preuve après application du théorème de transitivité de l'égalité*

A ce stade la manière dont on résout un sous-but peut introduire des restrictions sur la façon de résoudre l'autre sous-but. On ne peut plus prouver les résultats séparément et fournir les preuves ainsi obtenues à une fonction de validation pour obtenir une preuve du but initial.

Comme nous l'avons expliqué au chapitre 1, l'utilisateur peut choisir quel sous-but il veut attaquer en priorité. Dans cet exemple s'il prouve le premier sous-but (celui de la branche de gauche) en appliquant `EExact H1`, cela introduit la *contrainte* `?=d`, sur la variable existentielle. Bien que cette contrainte ne soit pas visible (à moins d'appliquer `NormEvars`), il est alors impossible de prouver le second sous-but en appliquant la tactique `EExact H0`. En effet cela contredirait la contrainte existant sur la variable existentielle. L'utilisateur peut néanmoins terminer la preuve par la tactique suivante `Rewrite <- H0;Rewrite <- H;Symmetry;EExact H1` qui, elle, ne contrarie pas la contrainte précédemment introduite.

On voit sur cet exemple qu'en Coq, c'est le système de preuve qui gère les contraintes introduites et empêche l'utilisateur d'introduire des incohérences.

D'autre part certaines tactiques peuvent se comporter de façon différente en fonction des contraintes présentes au moment de leur application. Cela se produit essentiellement avec les procédures de décisions automatiques ou les tactiques utilisant les opérateurs `Try` ou `Orelse`.

4. ou trans_equal définie la transitivité de la relation d'égalité dont l'énoncé est donné par: `(A:Set;x,y,z:A)x=y->y=z->x=z`

3.5.4 Interaction avec le retour-arrière

Reprenons la preuve du lemme précédent dans l'état représenté sur la figure 3.14.

L'utilisateur résout le second sous-but (celui de la branche de droite) avec la tactique `2: EExact H0`. Cela introduit la contrainte `?=b`.

Maintenant il s'attaque au sous-but restant avec la tactique `Try EExact H1.` Mais du fait de la contrainte précédemment introduite, il n'est pas possible d'unifier `d` avec la variable existentielle, donc `Exact H1` échoue et `Try EExact H1.` ne fait rien. L'utilisateur termine sa preuve par la tactique `EExact H.` qui unifie la variable existentielle avec `b`.

Il a donc joué le script suivant:

```
Intros;EApply trans_equal.
2:EExact H0.
Try EExact H1.
EExact H.
```

correspondant à l'abre ci dessous:

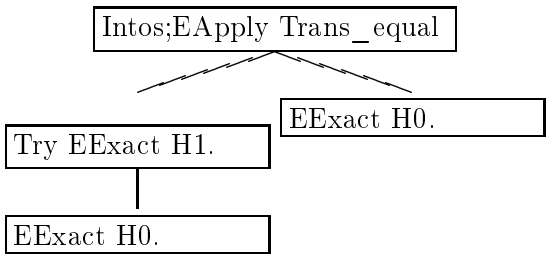


FIG. 3.15 – *L'arbre de preuve du lemme 1*

Il décide alors de défaire la branche de droite. Le mécanisme de retour arrière tel que nous l'avons décrit précédemment génère donc le script suivant [5]

```
Intros;EApply trans_equal.
Try EExact H1.
EExact H.
--------------
EExact H0.
```

A ce stade, il peut effectivement rejouer `EExact H0.` et terminer sa preuve car le mécanisme de retour arrière n'a pas changé les contraintes. Mais le script obtenu n'est pas valide. Il ne pourra être rejoué ultérieurement. En effet si l'on rejoue ce script, lors d'une

5. La partie de script au dessus des pointillés et celle qui est conservée et qui devrait correspondre à l'état du système. La partie du dessous est celle qui a été effacée et qui pourra éventuellement être rejouée

autre session de travail, lorsque la tactique `Try EExact H1.` est appliquée, il n'y pas de contraintes sur la variable existentielle introduite par `Intros;EApply trans_equal`. La tactique `EExact` peut donc introduire la contrainte `?=d` et unifier la variable avec `d`. Donc `Try EExact H1.` réussit et résout le premier sous-but en introduisant la contrainte `?=d`. Du coup `EExact H.` est obsolète. En pratique le système essaye de l'appliquer au sous-but restant et échoue. De même `EExact H0.` échoue car la variable ne peut plus être unifiée avec `b`.

On est confronté à des problèmes similaires avec les procédures de décision automatique. Ainsi on peut prouver le lemme précédent par le script suivant,

```
Intros;EApply trans_equal.
2:EAuto.  (* introduit la contrainte ?= b *)
EAuto.    (* Applique H1 *)
```

Le premier `EAuto` résout le second sous-but en utilisant l'hypothese `H0`. Cela introduit donc la contrainte `?= b`. et le second `EAuto` résout le premier sous-but en utilisant l'hypothèse `H1` qui est compatible avec la contrainte.

Si maintenant on efface la seconde branche on obtient le script suivant.

```
Intros;EApply trans_equal.
EAuto. (* introduit la contrainte ?=d *)
------
EAuto.
```

On peut rejouer le `EAuto` effacé car les contraintes n'ont pas été modifiées par notre mécanisme de retour-arrière mais là encore, le script obtenu n'est plus valide.

En effet si l'on rejoue ce script lors d'une nouvelle session, le premier `EAuto.` est appliqué au premier sous-but qu'il résout en utilisant l'hypothèse `H1` (qui précède l' hypothèse `H` dans la liste des hypothèse utilisées par `EAuto`). Cela introduit une contrainte `?=d`. Le second `EAuto` ne peut alors plus utiliser l'hypothèse `H0` car cela contredirait la contrainte. Il n'arrive donc pas à résoudre le sous-but dont la preuve nécessite maintenant des étapes de réécriture qu'`EAuto` ne sait pas faire. L'utilisateur se retrouve donc avec une preuve incomplète et un sous-but ouvert:

```
  d : nat
  H : a=b
  H0 : b=c
  H1 : a=d
  ============================
   ?112=c
```

Ces exemples montrent qu'en présence de variables existentielles, l'ordre dans lequel sont prouvés les différents sous-buts n'est pas indifférent. Et tout mécanisme qui modifie cet ordre (en particulier le retour-arrière logique) est donc susceptible d'invalider le script de preuve.

Pour résoudre ce problème et être capable de défaire une partie de preuve en présence de variables existentielles, tout en maintenant la validité du script, on doit étendre notre notion de dépendance. On ajoute aux dépendances logiques issues de l'arbre de preuve, un certain degré de dépendance historique (chronologique) entre les tactiques qui attaquent des sous-buts contenant des méta-variables. Il nous faut par ailleurs fournir le moyen de désinstancier une variable existentielle.

3.6 Solution du problème

3.6.1 Principe de base

L'idée est de maintenir les informations qui indiquent à quel endroit dans le script les variables existentielles ont été introduites et instanciées.

Ensuite on a principalement deux démarches possibles:

1. La première que nous qualifierons de conservatrice est utile si l'on désire défaire une branche sans revenir sur les contraintes qu'elle a introduites. Elle consiste à procéder au retour arrière comme dans le cas "sans variable existentielle" puis à transformer le script restant et à modifier l'état du système pour en rétablir la cohérence vis à vis de ces variables. Trois situations sont possibles:
   (a) Si la branche effacée ne travaille pas sur des variables existentielles contraintes on n'a rien à faire.
   (b) Si elle ne partage pas ses variables avec d'autres branches de l'arbre de preuve, on n'a qu'à supprimer les contraintes sur les variables existentielles.
   (c) Sinon il va falloir ajouter dans le script une tactique qui force l'instanciation des variables existentielles dans les autres branches où elles ont été utilisées. Le plus simple est donc d'insérer à la place des tactiques effacées qui effectuaient des instanciations une instruction `Instantiate` qui maintient simplement l'instanciation. De la sorte si l'on rejoue ce script lors d'une prochaine session les contraintes seront introduites au même endroit.
2. La seconde démarche que l'on pourrait qualifier de destructive sera utile lorsque l'on désire non seulement effacer une branche mais également revenir sur les contraintes qu'elle a introduites (la situation est encore plus complexe si on désire revenir sur ces contraintes en essayant de conserver néanmoins au maximum la branche où elles ont été introduites).
   La première démarche qui est une solution ad hoc entraîne une modification de l'arbre de preuve puisque l'on rajoute un nœud à chaque fois que l'on insère la tactique `Instanciate` dans le script. Nous ne la décrirons donc pas dans notre modèle d'arbre. Mais nous étendons maintenant ce modèle pour y décrire les dépendances introduites par les variables existentielles et la mise en œuvre de la seconde proposition de retour arrière logique.

3.6.2 Extension du modèle d'arbre de preuve

Soit $\mathcal{E}xiste$ l'ensemble des variables existentielles.

On se donne deux nouvelles opérations sur les arbres:

- $existe : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{E}xiste)$
- $instanciee : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{E}xiste)$

Soit c un nœud de l'arbre de preuve, $existe(c)$ représente l'ensemble des variables existentielles présentes dans le sous-but $but(c)$ et $Instanciee(c)$ l'ensemble des variables existentielles qui sont instanciées lors de l'application de $tactique(c)$.

En l'absence de variables existentielles les dépendances entre les différents sous-buts étaient totalement décrites par la relation $\prec_p$.

On introduit maintenant une notion réduite de dépendance historique (on utilisera parfois le terme dépendance existentielle) entre des sous-buts partageant une variable existentielle. Pour tous nœuds c et c_1 d'un arbre $\mathcal{T}$, on notera $(depExist\ c\ c_1)$ la relation définie par:

1. $c <_r c_1$
2. $\neg(c \prec_p c_1)$ (pour la rendre disjointe de la relation $\prec_p$)
3. $Instanciee(c) \cap existe(c_1) \neq \phi$ ($tactique(c)$ a instancié au moins une des variables existentielles contenues dans le sous-but $but(c_1)$)

On note

$$DepExist(c) = \{c_1 \in S, (depExist^+\ c\ c_1)\}$$

l'ensemble des "dépendances existentielles" du nœud c.

Enfin on introduit une notion générale de dépendance immédiate qui englobe les deux notions de dépendance.

On note $(depAll\ c\ c_1) \Longleftrightarrow (c \prec_p c_1) \vee (depExist\ c\ c_1)$

On définit une relation générale de dépendance en introduisant la relation Dep comme la fermeture transitive de $depAll$.

$$(Dep\ c\ c_1) \Longleftrightarrow (depAll^+\ c\ c_1)$$

On peut enfin définir l'ensemble des dépendances d'un nœud i par

$$Dep(c) = \{c_1 \in \mathcal{T}', (Dep\ c\ c_1)\}$$

En l'absence de variables existentielles $Dep(c)$ se confond évidement avec $\{c_1 \in \mathcal{T}', (c \prec_p c_1)\}$.

Utilisation pour le retour arrière. Compte tenu de ces définitions, au niveau du script un retour arrière local sur la tactique i associé au nœud c nous conduit à déplacer à la fin du script, i et toutes les tactiques associées aux nœuds de $Dep(c)$.

La fonction *undo* sur un nœud c se définit donc toujours par

$$undo((\mathcal{T},P,rg),c) = (\mathcal{T},P,rg')$$

mais où rg'est maintenant défini par:

$$rg' = \lambda c_1.if\ c_1 \in Dep(c) \cup \{c\}$$
$$then\ Card(\mathcal{T}') - Card(\{c_i \in \mathcal{T}' | c_i \in Dep(c) \cup \{c\} \wedge rg(c_i) > rg(c_1)\}$$
$$else\ rg(c_1) - Card(c_i \in \mathcal{T}' | c_i \in Dep(c) \cup \{c\} \wedge rg(c_i) < rg(c_1)\}$$

Ce mécanisme de retour arrière correspond à la démarche destructive précitée. Il est sûr mais il n'est pas optimal. Nous montrons maintenant comment en améliorer l'efficacité.

Séparer le retour arrière et la "désinstanciation" des variables existentielles Il peut en effet arriver que l'on veuille effacer une tactique ayant instancié une variable existentielle sans pour autant défaire cette instanciation.

Cela nous amène à séparer retour arrière local et "désinstanciation" (on retrouve la démarche conservatrice).

Ainsi si l'on veut effacer la tactique i d'un script S sans défaire les instanciations qu'elle a effectuées, on commence par effacer les dépendances logiques de i. Mais on doit aussi introduire dans le script une commande pour forcer les instanciations de variable afin que les mêmes instanciations se produisent si l'on rejoue le script dans une session ultérieure.

Pour cela on recherche parmi les tactiques historiquement liées à i, celle qui est jouée la première.

soit

$$j = min_{<_r}(DepExist\ j\ i)$$

et on la précède d'une commande forçant les instanciations. En Coq, ce sera par exemple `Instantiate`.

Exemple: Soit le lemme suivant :

```
Lemma lemme2 : (Ex [x:nat]
 (( x + 4)=7∧ (x ≥ 4)∨(x ≥ 2))
      ∧ (x ≤ 7)∧ (x + 2)=5).
```

l'utilisateur applique le script ci-dessous :

```
EApply ex_intro.
Split.
Left.
Split.
Do 4 Rewrite <- plus_n_Sm;Do 4 Rewrite <- plus_n_O;EAuto .
2:NormEvars;Split;Simpl.
2:EAuto.
2:Trivial.
NormEvars;Simpl.(* impossible d'aller plus loin *)
```

Qui correspond à l'arbre de preuve de la figure 3.16.

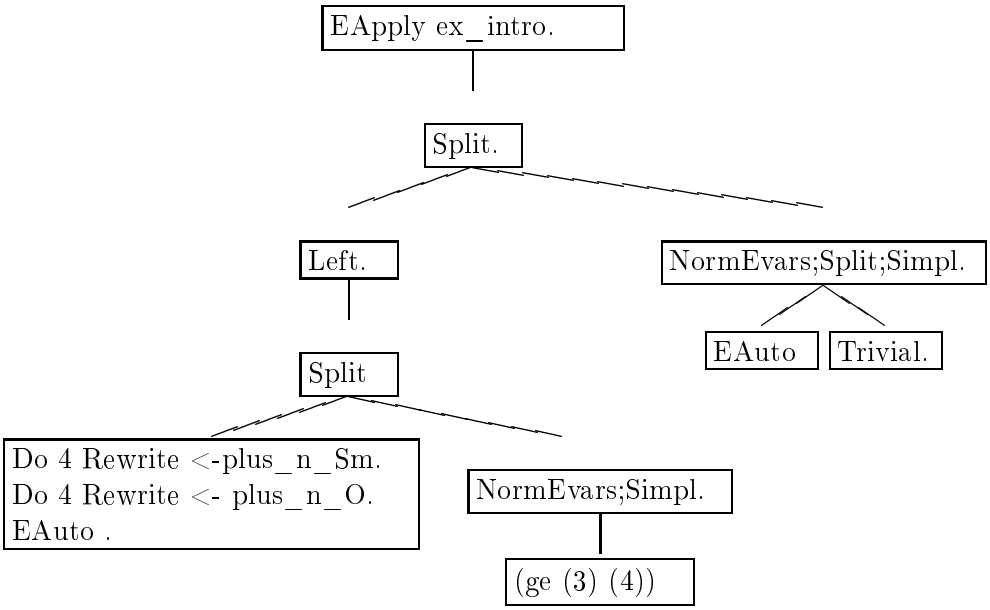


FIG. 3.16 – *L'arbre de preuve du lemme 2*

A ce stade l'utilisateur constate qu'il est impossible de finir la preuve car on n'a pas choisi le bon membre dans la disjonction `((plus x (4))=(7)/\(ge x (4))\/(ge x (2)))` . Il coupe donc la branche issue de `Left`.

Avec le retour-arrière logique que nous venons de proposer, les deux branches issues du `Split` doivent être effacées car la seconde dépend historiquement de la première (elle contient une variable existentielle instanciée dans la première.). Le script conservé est donc minimal:

```
EApply ex_intro.
Split.
```

```
----------------------
Left.
Split.
Do 4 Rewrite <- plus_n_Sm;Do 4 Rewrite <- plus_n_O;EAuto .
2:NormEvars;Split;Simpl.
2:EAuto.
2:Trivial.
NormEvars;Simpl.(* impossible d'aller plus loin *)
```

L'utilisateur a perdu tout le travail fait sur la branche de droite.

Avec le retour-arrière logique de la section 3.3 qui ne prend pas en compte les variables existentielles, il obtient le script suivant dont il voudra modifier la partie effacée.

```
EApply ex_intro.
Split.
2:NormEvars;Split;Simpl.
2:EAuto.
2:Trivial.
----------------------
Left.
Split.
Do 4 Rewrite <- plus_n_Sm;Do 4 Rewrite <- plus_n_O;EAuto .
NormEvars
```

Mais ce script n'est plus valide. En effet la tactique `Normevars` y est appliquée avant toute introduction de contrainte. De plus, même si on la supprimait, l'application du `EAuto` de la ligne suivante aux sous-buts `(le? (7))` introduirait une nouvelle contrainte `?=7` rendant impossible la preuve du sous-but `(plus x (2))=(5))`.

Pour garder l'instanciation faite précédemment et maintenir la validité du script, le système insère à la place de la tactique `Left` (correspondant à la racine de la branche effacée) une tactique qui se contente de faire la même instanciation de la variable existentielle que celle qui était faite précédemment, on obtient le script suivant.

```
EApply ex_intro.
Split.
Instantiate 1 (3). (* ici on force l'instanciation *)
2:NormEvars;Split;Simpl.
2:EAuto.
2:Trivial.
----------------
...
```

Ici le travail fait pour prouver la branche de droite est conservé et l'utilisateur n'a plus qu'a modifier la partie effacée et à terminer sa preuve. Le script complet est donc le suivant:

```
EApply ex_intro.
Split.
Instantiate 1 (3). (* ici on force l'instanciation *)
2:NormEvars;Split;Simpl.
2:EAuto.
2:Trivial.
Right;EAuto .
```

Optimisation de la désinstanciation. On peut aussi essayer d'améliorer le mécanisme de désinstanciation. L'idée est que certains sous-buts peuvent contenir une variable existentielle dont l'instanciation n'influe par sur la preuve. De tels sous-buts n'ont pas à être reconsidérés lorsque l'on désinstancie cette variable.

Exemple: Soit le lemme suivant:

```
Lemma lemme3 : (Ex [x:nat] (ge x (3))/\(minus x x)=(0)/\(plus x (2))=(7)).
```

On applique le script ci-dessous :

```
EApply ex_intro.
Repeat Split.
```

qui conduit à trois sous-buts partageant la même variable existentielle.

On prouve le premier par `EAuto` qui, utilisant le fait que `(n:nat)(ge x x)`, introduit une contrainte `?=3`. On trouve le second en appliquant le lemme `minus_n_n`[6]. Enfin on utilise `NormEvars` pour faire apparaître les contraintes dans le dernier sous-but.

```
EAuto. (* introduit une contrainte ?=3 *)
Symmetry;Apply minus_n_n.
NormEvars.(* ici on découvre que l'on doit changer l'instanciation de la
    variable existentielle mais on n'a pas à modifier le sous-but numéro 2 *)
```

6. dont l'énoncé est `(n:nat)(0=(minus n n))`

L'arbre de preuve correspondant est le suivant :.

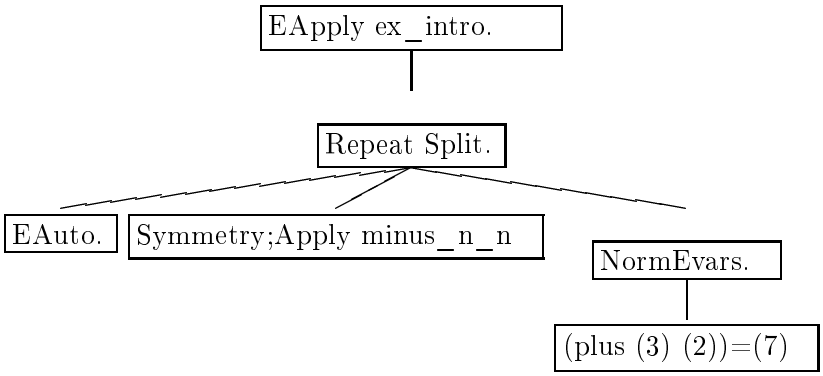


FIG. 3.17 – *L'arbre de preuve du lemme 3*

Mais la contrainte introduite sur la variable existentielle par `EAuto.` empêche l'utilisateur de terminer sa preuve car `(plus (3) (2))=(7)` n'est évidemment pas démontrable. Il doit donc faire un retour-arrière et modifier la contrainte. D'un autre côté la modification de cette contrainte n'influence pas la preuve du second sous-but issu du `Split`: `(minus ? ?)=(0)`. En effet ce résultat est indépendant de la valeur de `?`. On doit donc pouvoir le conserver et obtenir le script.

```
EApply ex_intro.
Repeat Split.
2:Symmetry;Apply minus_n_n.
-------------------
...........
```

L'utilisateur peut alors modifier la partie effacée et terminer sa preuve. Le script final est

```
EApply ex_intro.
Repeat Split.
2:Symmetry;Apply minus_n_n.

2:Do 2 Rewrite <- plus_n_Sm;Do 2 Rewrite <- plus_n_O;EAuto .
NormEvars;Simpl;EAuto.
```

Dans un système comme Coq, seules certaines tactiques sont susceptibles de manipuler des variables existentielles.

Lorsqu'une sous-preuve n'utilise pas ces tactiques on peut affirmer que son comportement est indépendant de l'instanciation des variables existentielles qu'elle contient. C'est le cas de la preuve du second sous-but dans l'exemple précédent.

Nous pouvons donc affiner notre relation de dépendance historique pour ne pas introduire de dépendances historiques vers de telles tactiques.

Soit SE l'ensemble de tactiques pouvant manipuler des variables existentielles. On introduit une variante de la relation $depExist$ (que nous appelons $depExistOpt$) de façon à ne pas ajouter de dépendance historique vers des tactiques n'appartenant pas à SE.

On définit donc

$$(depExistOpt\ c\ c_1) \Longleftrightarrow (depExist\ c\ c_1) \wedge tactique(c_1) \in SE$$

puis

$$(depAllOpt\ c\ c_1) \Longleftrightarrow (c\ \prec_p c_1) \vee (depExistOpt\ c\ c_1)$$

$$(DepOpt\ c\ c_1) \Longleftrightarrow (depAllOpt^+\ c\ c_1)$$

L'ensemble des dépendances d'un nœud c devient

$$DepOpt(c) = \{c_1 \in \mathcal{T}', (DepOpt\ c\ c_1)\}$$

Enfin dans les fonctionnalités de retour arrière nous remplaçons Dep par $DepOpt$.

3.7 Idée de l'implémentation

Côté Coq nous avons ajouté le code Ocaml permettant de savoir à chaque étape quelles sont les variables existentielles qui sont introduites et quelles sont celles qui sont contraintes. Nous avons aussi ajouté le code nécessaire pour supprimer une contrainte sur une variable existentielle. Nous avons étendu le protocole de communication de l'interface pour transmettre les informations sur les variables existentielles.

Du côté de l'interface, chaque ligne de script doit être annotée par la liste des variables qu'elle introduit et par celle des variables qu'elle contraint.

On utilise ces annotations pour calculer les dépendances d'une ligne de script telle qu'elles sont données par la relation Dep du paragraphe 3.6.2. Le retour arrière logique s'implémente alors comme dans le cas sans variable existentielle. On déplace tout ce qui dépend du nœud effacé, de la *zone finale* vers le début de la *zone normale*. On parcourt le script pour mettre à jour les index. Enfin on met à jour l'état du système de preuve.

L'optimisation du paragraphe 3.6.2 pose le problème de la synchronisation des chemins dans l'interface avec les chemins dans le système de preuve. En effet du côté de l'interface on va insérer la tactique d'instanciation quelque part dans le script, cela va donc décaler le chemin des tactiques de rang supérieur. Pour le reste on fait un retour-arrière logique

comme s'il n'y avait pas de variable existentielle (donc en utilisant les dépendances issues de la relation $\prec_p$ décrites au paragraphe 3.3). La tactique `Instantiate` n'apparaît donc pas dans la structure d'arbre maintenue par le système. Deux solutions sont possibles: maintenir une table de correspondance ou insérer la tactique `Instanciate` à la suite de son père en utilisant la tacticielle `then`, ce qui évite de décaler les chemins. L'implémentation des autres optimisations n'a pas encore été étudiée.

3.7.1 Conclusion sur les variables existentielles

En pratique, l'utilisation des variables existentielles dans un système comme Coq pose de nombreux autres problèmes. Leur présence en même temps que les types dépendants complique sensiblement le mécanisme de l'unification. Leur emploi semble encore peu répandu: une rapide analyse des "contributions" de la distribution standard révèle que `EAuto` apparaît 2 fois et `EApply` 13 fois sur un total de 118106 lignes de script. Nous avons cependant étendu le mécanisme de retour-arrière logique que nous avions initialement conçu pour qu'il gère correctement les contraintes liées aux variables existentielles. Même si la solution retenue n'est pas optimale, les problèmes liés aux variables existentielles ne constituent pas un obstacle à l'utilisation de nos outils dans l'état actuel du système Coq.

3.8 Autres études du mécanisme de retour-arrière

Le problème du retour-arrière a déjà été étudié par Lena Magnusson [79] dans le prouveur Alf. Dans ce système on manipule directement le terme de preuve et non un script. Les deux opérations de base sur le terme sont l'insertion et l'effacement. C'est à partir d'elles que sont développées toutes les autres manipulations ainsi que le mécanisme de retour-arrière logique. Dans le cas de Alf, le problème de tactique dont le comportement change en fonction des contraintes n'apparaît donc pas

Amy Felty et Douglas Howe [47, 48] ont aussi étudié le problème dans le cadre du prouveur en λ-Prolog développé par A. Felty. Pour mettre en œuvre le retour-arrière logique, ils réalisent un retour-arrière historique puis rejouent automatiquement et de façon transparente à l'utilisateur les parties de preuve de script correspondant à des branches de l'arbre de preuve indépendantes du nœud effacé. Ils proposent aussi de modifier incrémentalement le but de la preuve en cours et de réutiliser toutes les parties de preuve qui ne sont pas affectées par la modification faite.

Eastaughffe et Ozols [37] s'intéressent à la manipulation des arbres de preuve pour le système Isabelle. Ce système, rappelons-le, est basé sur l'unification d'ordre supérieur. Ils proposent une modification de l'algorithme d'unification permettant de garder trace de l'introduction des contraintes afin de les utiliser pour minimiser l'ensemble des résultats a défaire lors d'un retour-arrière logique.

Citons aussi les mécanismes de révision de preuve décrits par Andrew Stevens [128] basés sur le marquage des sous-termes dans les objets preuves.

Enfin on peut citer les mécanismes de "Proof Reuse" développés par Wolfgang Reif et Kurt Stenzel [114, 115] pour le système KIV dont le but est de faciliter la récupération de preuves en cas d'erreurs dans les spécifications et qui est à ce titre plus proche des outils d'aide à la modification proposés aux chapitres 8 et 9 de cette thèse.

Nous allons maintenant présenter des outils d'expansion et de contraction de l'arbre de preuve avant d'aborder des mécanismes plus fins de manipulation de preuve et notamment la lemmification et la factorisation de preuve.

Chapitre 4

Contraction et expansion de l'arbre de tactiques

4.1 Introduction

Jusqu'à présent nous avons considéré qu'une étape de preuve était associée à une ligne de script. En fait dans la plupart des démonstrateurs, une ligne de script peut être soit une tactique de base, soit l'application d'un opérateur de composition à un ensemble de tactiques pour former une nouvelle tactique. On parle alors de tactique composée. Pour de nombreuses opérations telles que la sécurisation du nommage des hypothèses, nécessaire pour garantir le bon fonctionnement des outils de *lemmification* et de *factorisation* que nous proposons au chapitre 5, la détection de *code mort*, ou la localisation des erreurs utilisée dans les outils d'aide à la modification de la seconde partie de cette thèse, une tactique composée est une représentation trop grossière d'une étape de preuve. On veut être capable d'analyser le comportement de chacune des tactiques de base passées en argument aux opérateurs de composition.

Dans ce chapitre nous allons nous intéresser à deux opérateurs particuliers, la composition séquentielle *then* (souvent représentée par un point virgule), et la composition parallèle *parallel*. L'application de l'opérateur *parallel* ne produit pas une tactique exécutable seule. Cet opérateur n'a de sens que combiné à l'opérateur *then* (on notera souvent $t;[t_1|...|t_n]$ pour $(then\ t\ (parallel\ t_1...t_n))$ mais on appelle parfois *thenl* l'opérateur issu de la composition de *then* et *parallel*).

La sémantique de ces opérations est la suivante:

– $t_1;t_2$ joue la tactique t_1 puis joue la tactique t_2 sur chacun des sous-buts produits par t_1.

– $t_0; [t_1|...|t_n]$ joue la tactique t_0 puis, si cela produit n fils, chaque t_i est appliquée au i^{eme} fils. Si le nombre d'arguments de *parallel* ne correspond pas au nombre de fils de t_0, la tactique échoue.

Pour manipuler les opérateurs *then* et *parallel* il est important de bien comprendre leurs priorités. Même lorsqu'elles sont clairement spécifiées dans les spécifications du système, un exemple simple peut permettre de les vérifier. Prenons un exemple dans le système Coq:
Exemple 4.1

Étant donnée le lemme suivant:

```
Lemma priorites : a->b->c->d->(a/\b)/\(c/\d).
Intros.
```

laquelle des tactiques suivantes peut être appliquée?

```
Split;Split;[Assumption|Assumption|Assumption|Assumption].
Split;Split;[Assumption|Assumption].
```

On constate que seule la première est acceptée; pour la deuxième le système nous annonce que le nombre de paramètres de parallel n'est pas bon. Cela permet de conclure que **l'opérateur then associe à gauche**. ♢

Remarquons qu'en l'absence de variables existentielles et de composition avec *parallel* l'opérateur *then* le fait que *then* associe à droite ou à gauche n'a pas d'importance. . Ces priorités étant établies, on écrira $(then\ t_1\ t_2\ t_3)$ pour $(then\ (then\ t_1\ t_2)\ t_3)$.

Dans la suite nous appellerons *tactique simple* toute ligne de script qui n'est pas construite en utilisant les opérateurs de compositions *then* et *parallel*. Pour les autres nous parlerons de *tactique composée*.

Ces opérateurs permettent de définir deux opérations sur l'arbre de preuve: la contraction et l'expansion. La contraction consiste à regrouper, à l'aide des opérateurs de composition, un ensemble de tactiques correspondant à une branche de l'arbre de tactique, en une seule tactique. L'expansion est la réciproque, elle consiste à découper une tactique composée en tactiques simples.

Ces deux opérations peuvent évidemment être appliquées à l'ensemble d'un script de preuve pour obtenir une preuve composée d'une unique tactique dans le cas de la contraction ou un script uniquement composé de tactiques simples dans le cas de l'expansion.

La figure 4.1 page 63 donne un exemple d'application de l'expansion sur un théorème d'arithmétique sur Z de la bibliothèque standard de Coq[1].

Nous présentons maintenant les procédures de contraction et d'expansion ainsi que quelques uns des outils qu'elles permettent de mettre en œuvre.

1. theories/ZARITHM/fast_integer.v

```
Lemma ZLII:
 (x,y:positive) (compare x y INFERIEUR) = INFERIEUR ->
                (compare x y EGAL) = INFERIEUR ∨ x = y.

(Induction x;Induction y;Simpl;Auto;Try Discriminate);
  (Intros z H1 H2; Elim (H z H2);Auto; Intros E;Rewrite E;Auto).
Save.
```

On n'a qu'un seul nœud mais on peut le décomposer pour obtenir le script correspondant à un parcours de l'arbre suivant:

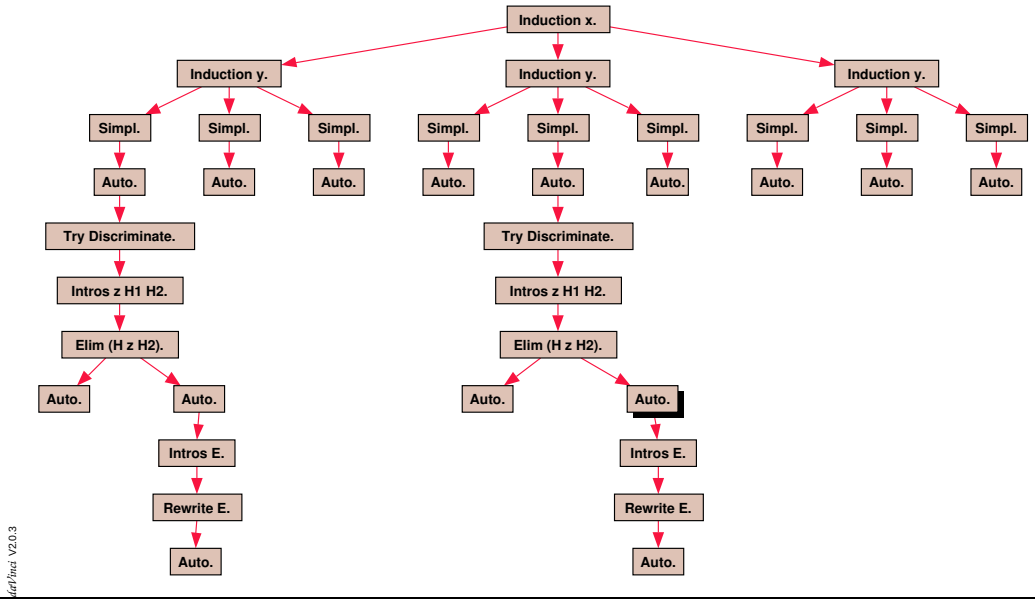


FIG. 4.1 – *Décomposition d'un nœud de l'arbre*

4.2 Contraction

La contraction est une opération assez simple à mettre en œuvre, dès lors que l'on connaît la structure de l'arbre de preuve.

4.2.1 Principe de fonctionnement

Un nœud étant sélectionné, on utilise les outils du chapitre précédent pour identifier ses fils. On donne ci-dessous les algorithmes de contraction. Les fonctions $fils : \mathcal{T} \rightarrow \mathcal{T}\ list$ et $tactique : \mathcal{T} \rightarrow \mathcal{TACTIQUE}$ donnent respectivement la liste des chemins des fils (ordonnée par $<_l$) et le texte de la tactique d'un nœud donné. Les fonctions *then* et *parallel* construisent les tactiques composées avec leurs arguments respectifs. Enfin $[m_1 \dots m_n]$ représente une liste contenant les n éléments $m_1 \dots m_n$.

ContracteTactique (n)
Entrée: n:$\mathcal{T}$ racine du sous-arbre à contracter.
Sortie: t:$\mathcal{TACTIQUE}$ tactique équivalente au sous-arbre de tactiques de racine n.
Début
Si (fils n)=[]
Alors **Retourner** (tactique n)
Sinon **Si** (fils n)=[$m\_1 \dots\ m\_n$]
Alors **Retourner** (then (tactique n) (parallel
(ContracteTactique m_1)...
(ContracteTactique m_n)))

FinSi
FinSi
Fin

Pour contracter tout un script il suffit donc de se positionner à la racine de l'arbre de tactique, d'appliquer `Contracte Tactique` et de substituer le résultat au script initial.

4.2.2 Variantes et réflexions

Tel que nous l'avons décrit jusqu'à présent, la contraction ne s'applique qu'à un sous-arbre complet. Il peut être intéressant de fournir un mécanisme de contraction partielle qui permet de ne contracter que jusqu'à une profondeur donnée.

Pour limiter en profondeur le mécanisme de contraction, il suffit d'ajouter un compteur dans la fonction précédente, de décrémenter ce compteur à chaque apple récursif et d'arrêter la récursion quand ce compteur est nul:

```
ContracteTactiqueProfondeur (n k)
Entrée: n:T  racine du sous-arbre à contracter.
            k:integer  profondeur maximale de contraction.
Sortie: t:TACTIQUE tactique équivalente au sous-arbre de tactiques de racine n.
Début
Si  k=0 Alors  Retourner (tactique n)
Sinon Si (fils n)=[]
       Alors  Retourner (tactique n)
       Sinon  Si (fils n)=[m_1... m_n]
            Alors  Retourner (then (tactique n)
                                   (parallel
                                     (ContracteTactiqueProfondeur m_1 k-1)...
                                     (ContracteTactiqueProfondeur m_n k-1)))
              FinSi
       FinSi
FinSi
Fin
```

D'autre part, notre algorithme contracte en conservant toute l'information de structure, c'est pour bien exprimer cela que l'on met un opérateur `parallel` même quand il n'y a qu'un fils. Mais il n'utilise pas le pouvoir de factorisation offert par la tacticielle *then*.

La figure 4.2 montre le résultat de la contraction du script expansé correspondant à l'arbre de la figure 4.1. On aimerait parfois pouvoir partager une tactique si elle est vraiment appliquée à tous les buts. On aura ainsi un script compacté comme celui de la figure 4.1.

Pour cela, on peut dans le troisième `Si` de l'algorithme `ContracteTactique`, analyser les tactiques s'appliquant aux sous-buts issus de la tactique traitée. Si la même tactique est appliquée sur tous ces sous-buts, on peut supprimer l'opérateur `parallel`. La figure 4.3 page suivante donne l'algorithme ainsi modifié.

```
Lemma ZLII:
 (x,y:positive) (compare x y INFERIEUR) = INFERIEUR ->
                (compare x y EGAL) = INFERIEUR \/ x = y.
(Induction x;
 ([ Induction y;
    ([ Simpl;
       ([ Auto with arith;
          ([ Try Discriminate;
             ([ Intros z H1 H2;
                ([ Elim (H z H2);
                   ([ Auto with arith
                    | Auto with arith;
                      ([ Intros E;
                         ([ Rewrite E; ([ Auto with arith ]) ]) ]) ]) ]) ]) ]) ]) ]
)
     | Simpl; ([ Auto with arith ])
     | Simpl; ([ Auto ]) ])
  | Induction y;
    ([ Simpl; ([ Auto with arith ])
     | Simpl;
       ([ Auto with arith;
          ([ Try Discriminate;
             ([ Intros z H1 H2;
                ([ Elim (H z H2);
                   ([ Auto with arith
                    | Auto with arith;
                      ([ Intros E;
                         ([ Rewrite E; ([ Auto with arith ]) ]) ]) ]) ]) ]) ]) ]
)
     | Simpl; ([ Auto with arith ]) ])
  | Induction y;
    ([ Simpl; ([ Auto with arith ])
     | Simpl; ([ Auto with arith ])
     | Simpl; ([ Auto with arith ]) ]) ]).
```

FIG. 4.2 – *Résultat de la contraction du script expansé correspondant à l'arbre de la figure 4.1.*

```
ContracteFactorise (n)
Entrée: n:T racine du sous-arbre à contracter.
Sortie: t:TACTIQUE tactique équivalente au sous-arbre de tactiques de racine n.
Variables locales: AddParallel:boolen
 Début
Si (fils n)=[]
  Alors Retourner (tactique n)
  Sinon Si (fils n)=[m]
    Alors Retourner (then (tactique n)(ContracteFactorise m))
    Sinon Si (fils n)=[m_1 ... m_n]
       Alors AddParallel:=Faux
             Pour i = 2 à n Début
             Si (tactique m_i) ≠(tactique m_1)
                 Alors AddParallele:=Vrai;sortir de la boucle
             FinSi
             Fin
             Si AddParallel
               Alors Retourner (then (tactique n)
                                     (parallel
                                         (ContracteFactorise m_1)...
                                         (ContracteFactorise m_n)))
                Sinon Retourner (then (tactique n)
                                         (ContracteFactorise m_1))
             FinSi
          FinSi
        FinSi
FinSi
Fin
```

FIG. 4.3 – *Algorithme de contraction avec factorisation*

Mais lors d'une telle contraction, on perd de l'information sur les tactiques contractées. En effet, au départ, chaque tactique correspond à une ligne de script et est annotée par le nombre de buts qu'elle a générés. Après une contraction avec factorisation, on ne dispose plus de cette information que sur la tactique obtenue. Par exemple après contraction avec factorisation de l'arbre figure 4.1 on ne sait plus que la tactique `Induction x` génère trois fils.

Il peut être intéressant de trouver un compromis entre la contraction sans factorisation donné par l'algorithme `ContracteTactique` et la contraction avec factorisation totale fournie par l'algorithme `ContracteFactorise`. L'idée est de regrouper les nœuds sans frère. Il suffit de parcourir l'arbre en appliquant à chaque nœud une fonction de contraction qui ne contracte pas les nœuds qui ont plusieurs fils.

Ainsi, compacté le script correspondant à l'arbre de la figure 4.1 devient

```
Induction x.
Induction y.
```

```
Simpl;Auto;Try Discriminate;Intros z H1 H2;Elim (H z H2)
Auto.
Auto;Intros;Rewrite E;Auto.
Simpl;Auto.
Simpl;Auto.
...
```

Enfin, il nous faut signaler que la contraction peut modifier l'ordre d'évaluation des tactiques et que par conséquent elle n'est sûre que dans un monde sans variables existentielles.

4.3 Expansion

L'expansion est plus compliquée à mettre en œuvre car pour découper une tactique composée à partir de l'opérateur *then*, il faut savoir combien de sous-buts sont générés par chacune des tactiques simples qui la compose.

4.3.1 Principe de fonctionnement

Généralement deux approches sont possibles:

1. Travailler sur la structure d'arbre (Dans Coq le `Prooftree`)
2. Travailler indépendamment de cette structure du coté de l'interface.

La première solution est plus économique et plus simple à mettre en œuvre. Mais, elle n'est possible que si le démonstrateur maintient de façon interne une structure d'arbre suffisamment informative ce qui est le cas du système Coq.

La seconde solution a l'avantage d'être complètement générique. Elle peut en effet s'adapter à n'importe quel système de preuve capable de lui fournir les informations lui permettant de reconstruire l'arbre de preuve (cf. chapitre 3). De plus, pour travailler directement sur la structure interne maintenue par le système, il faut que le système ait été capable de la construire, c'est-à-dire que la tactique ait réussi. Cela rend impossible l'expansion d'une tactique composée qui échoue. Ce qui est pourtant utile justement pour comprendre les raisons de l'échec et localiser les tactiques qui en sont responsables.

Nous présentons maintenant succinctement la mise en œuvre de la seconde solution. La mise en œuvre pratique dans l'environnement CtCoq est rendue beaucoup plus complexe par l'architecture distribuée du système et les problème de synchronisation qu'elle introduit.

On donne ci-dessous l'algorithme d'expansion d'une ligne de script commençant par un opérateur `then`. Comme nous l'avons dit en introduction, le résultat de l'application de *parallel* à un ensemble de tactiques ne produit pas une tactique exécutable, on introduit donc le type $PTactiques$ des pseudo-tactiques. $PTactiques$ est la clôture de l'ensemble des tactiques par l'opérateur *parallel* (il représente soit une tactique soit un objet de type

(*parallel* $t_1 \dots t_n$) . Le premier argument d'un opérateur *then* est toujours une tactique. Les suivants sont des pseudo-tactiques, c'est-à-dire soit des tactiques soit le résultat de l'application de l'opérateur *parallel* à un ensemble de tactiques. Notons t_1,$t_2 \dots t_n$ la liste d'arguments de l'opérateur *then* que l'on désire expanser.

On commence par jouer le premier argument qui produit n sous-buts auxquels on va devoir appliquer les tactiques restantes. Deux cas sont alors possibles:

1° si l'argument t_2 est une tactique, c'est que tous les sous-buts vont subir le même traitement. Nous allons donc devoir appliquer *then* $t_2 \dots t_n$ à chacun des sous-buts créés par t_1.
Pour cela, t_1 ayant crée n sous-buts, on fait $n-1$ copies de la suite $t_2 \dots t_n$. Ces copies sont stockées dans une pile $P1$. Puis nous faisons un appel récursif pour expanser la première branche (la liste de tactiques $t_2 \dots t_n$ qui s'applique au premier sous-but généré par t_1). Lorsque l'expansion de la première branche est finie, on passe à la seconde branche en dépilant la suite de tactiques correspondantes.

2° si l'argument t_2 est de type *parallel* $t_{2_1} \dots t_{2_k}$ c'est que chaque sous-but va subir un traitement spécifique. On fait k copies de $t_3 \dots t_n$ puis on ajoute $t_{2_1} \dots t_{2_k}$ respectivement en tête de chacune des copies. Ces copies sont stockées dans une pile $P2$. On stoppe alors l'expansion sur la branche en cours, on incrémente l'index en fonction du nombre de sous-buts produits par t_1 et on passe à l'expansion de la branche suivante (en la dépilant de $P1$).

Quand la pile $P1$ est vide, on traite le cas des tactiques issues de l'expansion des opérateurs *parallel* en les dépilant de $P2$. Lorsque $P2$ est vide, l'expansion est terminée. La procédure `ExpanseThenListe` donnée figure 4.4 est le corps de l'algorithme d'expansion. Un exemple d'utilisation est donné en annexe.

Elle est utilisée par la procédure `ExpanseLigne` ci dessous pour expanser une ligne de script.

ExpanseLigne (n)
  Entrée: n : *integer* `rang de la tactique expansée`
  Variables locales: `tactique:`$\mathcal{TACTIQUE}$
  `index:`*integer*
  `pile-d-arguments:`(*PTactique pile*)
  `p1:`(*PTactique pile*)
  `p2:`(*PTactique pile*)
  `nbuts:`*integer*
Début
`tactique:=`*tactique*`'(n)`
`index:=`*index*`'(n)`
Si `tactique est de la forme` *then* `t_1... t_n`
  Alors `pile-d-arguments:=CreePile; p1:=CreePile; p2:=CreePile`
    `Pour i = n à 1 Empiler(t_i,pile-d-arguments)`
    `ExpanseThenListe(pile-d-arguments,n,index,p1,p2,index)`
FinSi
Fin

```
ExpanseThenListe (pile-d-arguments rang index p1 p2 k)
  Entrée: pile-d-arguments:PTactiques pile  liste des arguments de l'opérateur then.
          rang:integer                      rang de la tactique expansée.
          index:integer                     index de la tactique expansée.
          p1:(PTactique pile)               pile de tactiques en attentes.
          p2:(PTactique pile)               pile des parallel en attentes.
          k:integer                         sauvegarde de l'index initial
  Variables locales: tactique:TACTIQUE
                     nbuts:integer
Début
tactique:=Dépiler(pile-d-arguments)
Inserer-dans-le-script(tactique,rang,index)
jouer(tactique)
nbuts:=nbut(tactique)
Si Est-vide(pile-d-arguments)
   Alors
     Si Est-vide(p1)
        Alors
           Si Est-vide(p2)
              Alors sortir
              Sinon p1:=Clone(p2) ;Vider(p2);index:=k
           FinSi
     FinSi
     pile-d-arguments:=Dépiler(p1)
     ExpanseThenListe(pile-d-arguments,(rang+1),(index+nbuts),p1,p2,k)
   Sinon premier:=Sommet(pile-d-arguments)
     Si premier est de la forme (parallel t_1... t_nbuts)
        Alors
          Si Est-vide(p2)
             Alors Dépiler(pile-d-arguments)
                   Pour i=nbuts a 1
                      Empiler(Empiler(t_i,Clone(pile-d-arguments)),p2)
          FinSi
          index:=index+nbuts
          Si Est-vide(p1)
             Alors p1:=Clone(p2) ; Vider(p2);index:=k
          FinSi
        pile-d-arguments:=Dépiler(p1)
        Sinon
          Si nbuts>0
             Alors Pour i=nbuts a 1 Empiler(Clone(pile-d-arguments),p1)
          FinSi
          Si Est-vide(p1)
             Alors
               Si Est-vide(p2)
                  Alors sortir
                  Sinon p1:=Clone(p2) ;Vider(p2);index:=k
               FinSi
          FinSi
          pile-d-arguments:=Dépiler(p1)
     FinSi
     ExpanseThenListe(pile-d-arguments,(rang+1),index,p1,p2,k)
FinSi
Fin
```

FIG. 4.4 – *Corps de l'algorithme d'expansion*

Enfin, pour expanser le script complet, on expanse en séquence chacune des lignes.

Nous présentons maintenant trois applications de l'expansion de preuve qui seront utilisées dans la suite de ce travail.

4.3.2 La sécurisation du nommage des hypothèses

Nous discutons dans le chapitre 5 les problèmes liés à la façon de faire référence aux hypothèses locales. Pour l'instant disons simplement que lorsque des hypothèses sont introduites dans le contexte local un nom peut leur être attribué soit explicitement par l'utilisateur (c'est le cas en Coq lorsque l'on applique `Intros n m`), soit automatiquement par le système (c'est le cas si on utilise simplement `Intros`). Mais l'algorithme utilisé par le système pour nommer une hypothèse dépend du contexte où s'effectue la preuve. Nous voulons être capable de transformer le script de sorte que toutes les instructions introduisant des hypothèses locales les nomment explicitement de manière à rendre les noms introduits indépendants du contexte où est jouée la preuve.

Dans une tactique composée de la forme $tac_1; Intros; tac_2$, si tac_1 génère plusieurs sous-buts, le nombre d'hypothèses introduites par $Intros$ sur chacun de ces sous-buts peut varier. Nous ne pouvons donc en aucun cas nous contenter de transformer notre tactique en $tac_1; Intros\ n_1...n_2; tac_2$. Nous commençons par expanser la tactique. Ensuite, pour chaque `Intros.` joué, nous regardons quels sont les noms d'hypothèses qui ont été introduits par le système et nous modifions le script pour les faire figurer explicitement dans le `Intros`.

Exemple 4.2

Ainsi, si l'on considère le lemme suivant.

```
Lemma nommage :(A,B,C:Prop)A->(B->C->A)/\(B->A).
Intros;Split;Intros;Assumption.
```

On fait une expansion, puis on analyse la base d'hypothèses avant et après avoir joué le premier Intros. On constate qu'il introduit `A`,`B`, `C` *et* `H`. *Il doit donc être transformé dans le script en* `Intros A B C H`. *On continue l'analyse,* `Split` *génère deux sous-buts.* `Intros` *appliqué au premier sous-but introduit deux hypothèses* `H1` *et* `H0` *tandis qu'appliqué au second il n'introduit qu'une hypothèse* `H0` *(qui n'est bien sûr pas forcément la même que l'hypothèse* `H0` *introduite dans le premier). Chacune de ces tactiques doit donc être modifiée dans le script, respectivement remplacée par* `Intros H0 H1` *et* `Intros H0`. *Enfin, on re-contracte le script en utilisant* `ContracteFactorise`. *On obtient:*

```
Intros A B C H;Split;[Intros H0 H1|Intros H0];Assumption.
```

Remarquons que `Intros A B C H;Split;Intros H1 H0;Assumption` *échouerait avec le message* `Error: INTRO used non-product` *car dans le second cas on n'a pas d'hypothèse* `H1` *pouvant être introduite.* ♢

4.3.3 La détection de code mort

Dans cette section on appelle code mort des tactiques qui ne font rien, c'est-à-dire qui ne modifient ni l'ensemble des buts courants ni les contraintes sur les variables existentielles. Nous verrons au paragraphe 7.3 du chapitre 7 une autre forme de code mort.

En Coq, les tactiques susceptibles de ne pas modifier l'état du système de preuve sont par exemple `Auto`, `Try Tactique` ...

Lors qu'elle est isolée, détecter qu'une telle tactique ne fait rien est trivial, il suffit de comparer l'état du système avant de la jouer avec l'état du système après l'avoir joué (mais attention, il ne suffit pas de comparer les arbres de preuve, il faut aussi tenir compte des éventuelles contraintes sur des variables existentielles). Lorsque que l'on détecte une telle tactique, on peut décider de ne pas l'incorporer au script conservé dans la fenêtre d'édition ou simplement la marquer pour pouvoir la supprimer ultérieurement (elle peut en effet être utilisée par la suite si l'on veut contracter un script).

Mais lorsqu'une telle tactique est utilisée dans une tactique composée (formée à partir des opérateurs de composition), qui elle, modifie l'état du système, les choses se compliquent. L'idée est alors d'expanser la tactique composée pour se ramener au cas des tactiques simples, puis de supprimer le code mort avant de recompacter les tactiques simples restantes.

Exemple 4.3

Soit le lemme suivant:

```
Lemma plus_sym : (n,m:nat)((plus n m)=(plus m n)).
Intros n m ;Auto with arith; Try Apply SuperTheorem;Elim n ;
   Simpl ; Auto with arith;Intros y H ;Auto with arith;
     Elim (plus_n_Sm m y) ; Auto with arith.
```

Une expansion suivie d'une analyse des tactiques simples produites puis d'une contraction du script montre que ce script peut être simplifié en:

```
Intros n m ;Elim n ; Simpl ; [Auto with arith;Intros y H |Intros y H;
Elim (plus_n_Sm m y) ; Auto with arith].
```

Nous n'avons pas poussé plus loin le travail sur le code mort qui outre l'accroissement de la lisibilité du script peut améliorer les performances lorsque l'on rejoue le code ultérieurement. Les procédures de décision automatique peuvent en effet consommer beaucoup de temps pour finalement laisser un but inchangé.

Il pourrait aussi être intéressant, par exemple, de détecter le cas où la tactique `Symmetry` est appliquée un nombre pair de fois ou les cas où des règles de réécriture sont appliquées de façon cyclique. Enfin signalons que l'optimisation du code, et notamment l'élimination des inférences inutiles dans HOL, a été abordé en détail par Richard Boulton dans sa thèse [22].

4.3.4 La localisation d'erreurs

Le problème avec un script de preuve sous forme contractée, comme celui de la figure 4.1, est la correction des erreurs en cas d'échec. En effet, soit la tactique repasse complètement, soit elle échoue, mais il est impossible de n'en conserver qu'une partie. Lorsqu'un script échoue, il est important de pouvoir déterminer le plus précisément possible la partie du script qui pose problème. Lorsqu'une tactique composée échoue, nous voulons donc être capable de comprendre lequel de ses composants est responsable de l'échec. Il faut donc pouvoir expanser une tactique composée même si elle ne peut être rejouée, afin de conserver toutes les parties de la preuve qui restent valides. Pour cela, nous modifions donc notre algorithme d'expansion pour qu'il prenne en compte, lors de l'exécution, l'échec d'une (ou plusieurs) des tactiques simples composant la commande expansée.

Quand une tactique simple échoue, on recrée la tactique composée représentant la branche qui en est issue, on la déplace à la fin du script, on incrémente l'index (car on a un but ouvert de plus) et on continue le traitement. La procédure `ExpanseVerif` de la figure 4.5, donne l'algorithme modifié.

A la fin de l'expansion toutes les parties ayant posé problème sont donc réécrites en fin de script où elles peuvent être modifiées pour être rejouées.

Dans la seconde partie de cette thèse nous étudions les problèmes liés à la modification de preuves et de théories. Au cours de telles modifications, on est amené à vérifier, en essayant de les rejouer, un certain nombre de résultats dépendants des résultats initialement modifiés (le calcul des dépendances et une méthode proposant à l'utilisateur un ordre de propagation des modifications sont étudiées aux chapitre 6 et 8). Si les modifications sur les résultats initiaux ne sont pas trop importantes, on peut espérer pouvoir, au moins partiellement, rejouer les preuves des théorèmes qui en dépendent.

D'autre part comme nous le montrons au chapitre 9, il peut arriver qu'après de tels changements, le script d'une preuve apparemment indépendante des résultats modifiés ne puisse plus être rejoué. Nous verrons que cela est dû au changement de comportement des procédures de décision automatique et que dans la majorité des cas (mais pas dans tous !) le fait qu'une telle procédure résolve plus de choses rend une partie du script inutile. Comme nous le monterons dans chapitre 9, l'expansion du script sera une première étape essentielle à la détection et à l'élimination des parties de script devenues inutiles.

4.4 Conclusion

La composition des fonctions de contraction et d'expansion n'est pas l'identité mais elle admet un point fixe. En fait, les fonctions $Contracte \circ Expanse$ et $Expanse \circ Contracte$ sont idempotentes.

La contraction se réalise facilement du côté de l'interface, sans avoir besoin de dialoguer avec le système de preuve.

```
ExpanseVerif (pile-d-arguments rang index p1 p2 k)
  Entrée: pile-d-arguments:PTactiques pile  liste des arguments de l'opérateur then.
          rang:integer                      rang de la tactique expansée.
          index:integer                     index de la tactique expansée.
          p1:(PTactique pile) pile          pile de tactiques en attentes.
          p2:(PTactique pile)               pile des parallel en attentes.
          k:integer                         sauvegarde de l'index initial
  Variables locales: tactique:TACTIQUE
                     nbuts:integer
Début
tactique:=Dépiler(pile-d-arguments)
Inserer-dans-le-script(tactique,rang,index)
jouer-la-dernière-tactique-inserée
Si échec
   Alors deplacer-après-le-script(then tactique Clone(pile-d-arguments))
         Vider(pile-d-arguments) ;index:=index+1
   Sinon nbuts:=nbut(tactique)
FinSi
Si Est-vide(pile-d-arguments)
   Alors
     Si Est-vide(p1)
        Alors
           Si Est-vide(p2)
              Alors sortir
              Sinon p1:=Clone(p2) ;Vider(p2);index:=k
           FinSi
     FinSi
     pile-d-arguments:=Dépiler(p1)
     ExpanseVerif(pile-d-arguments,(rang+1),(index+nbuts),p1,p2,k)
   Sinon premier:=Sommet(pile-d-arguments)
     Si premier est de la forme (parallel t_1... t_nbuts)
        Alors
          Si Est-vide(p2)
             Alors Dépiler(pile-d-arguments)
                   Pour i=nbuts a 1
                      Empiler(Empiler(t_i,Clone(pile-d-arguments)),p2)
          FinSi
          index:=index+nbuts
          Si Est-vide(p1)
             Alors p1:=Clone(p2) ; Vider(p2);index:=k
          FinSi
        pile-d-arguments:=Dépiler(p1)
        Sinon
          Si nbuts>0
             Alors Pour i=nbuts a 1 Empiler(Clone(pile-d-arguments),p1)
          FinSi
          Si Est-vide(p1)
             Alors
               Si Est-vide(p2)
                  Alors sortir
                  Sinon p1:=Clone(p2) ;Vider(p2);index:=k
               FinSi
          FinSi
          pile-d-arguments:=Dépiler(p1)
     FinSi
     ExpanseVerif(pile-d-arguments,(rang+1),index,p1,p2,k)
FinSi
Fin
```

FIG. 4.5 – *Corps de l'algorithme d'expansion gérant les erreurs*

L'expansion quant à elle, nécessite un dialogue constant avec le système. Lors de la mise en œuvre pratique de l'algorithme d'expansion, cela entraîne de nombreuses complications liées à des problèmes de synchronisation. Le développement en cours de la nouvelle version de CtCoq en java (appelée PCoq)[69] devrait grâce au multi-threading permettre de mieux gérer ces problèmes.

Enfin ces fonctions sont la pierre angulaire de la plupart des manipulations de script exposées dans la suite de ce document.

Chapitre 5

Lemmification et factorisation de lemmes

Pendant la construction d'une preuve, on peut être amené à prouver un sous-lemme ayant un intérêt propre. On peut l'énoncer et le prouver séparément (Ainsi l'interface CtCoq permet d'avoir plusieurs fenêtres de preuve pour prouver plusieurs résultats "en parallèle") mais cela nécessite de savoir a priori quelles hypothèses du contexte local vont être utiles à sa preuve et doivent être abstraites pour en construire l'énoncé exact. Le problème que nous cherchons à étudier est celui qui se pose à l'utilisateur lorsqu'il veut faire un lemme indépendant d'une sous-preuve effectuée dans le contexte local. Il lui faut alors repérer a posteriori les hypothèses de ce contexte qui ont été réellement utilisées, les abstraire pour construire l'énoncé du théorème et finalement extraire la partie de script correspondant à sa preuve. C'est ce processus que nous appelons lemmification.

5.1 Lemmification

Nous commençons par un exemple (un peu idyllique) puis nous détaillons le processus de lemmification. Nous abordons ensuite son implémentation dans Coq et CtCoq. Enfin nous discutons les faiblesses de notre approche.

Exemple 5.1

Supposons que l'addition soit définie comme suit:

```
Fixpoint plus [n:nat] : nat -> nat :=
   [m:nat]Cases n of
      O   => m
  | (S p) => (S (plus p m)) end.
```

L'utilisateur décide d'en prouver la commutativité: il écrit le script suivant (les commentaires rappellent le chemin dans l'arbre de preuve). Les noms des hypothèses introduites

sont explicitement indiqués pour des raisons que nous discutons au § 5.1.3. Cela est nécessaire au bon fonctionnement de nos outils.

```
Lemma comPlus : (p,q:nat)(plus p q)=(plus q p).
Induction p.           (*  ()                *)
Simpl.                 (* (1)               *)
Induction q.           (* (1;1)             *)(*(q:nat)q=(plus q O)*)
Trivial.               (* (1;1;1)           *)
Intros.                (* (1;1;2)           *)
Rewrite H.             (* (1;1;2;1)         *)
Trivial.               (* (1;1;2;1;1)       *)
Intros n H.            (* (2)               *)
Simpl.                 (* (2;1)             *)
Intro q.               (* (2;1;1)           *)
Rewrite (H q).         (* (2;1;1;1)         *)
Elim q.                (* (2;1;1;1;1)       *)(*(S (plus q n))=(plus q (S n))*)
(Simpl; Trivial).      (* (2;1;1;1;1;1)     *)

Intros n0 H0   .       (* (2;1;1;1;1;2)     *)
Simpl.                 (* (2;1;1;1;1;2;1)   *)
(Rewrite H0; Trivial). (* (2;1;1;1;1;2;1;1) *)
```

Il remarque alors que durant cette preuve il a prouvé des résultats intéressants par eux-mêmes et qui pourront être utiles dans d'autres preuves.

Il retient `(q:nat)q=(plus q O)` *et* `(S (plus q n))=(plus q (S n))` *et décide d'en faire des lemmes à part entière.*

Pour cela il peut utiliser notre fonction de lemmification. Cette fonction prend pour arguments un chemin dans l'arbre de tactique et un nom et génère le lemme correspondant au sous-but associé à ce chemin.

Ainsi dans cet exemple il pourra générer

```
# lemmify [1;1] "plus_n_O";;
Theorem  plus_n_O : (q:nat)q=(plus q O).
Intros .
Induction q.
Trivial.

Intros.
Rewrite H.
Trivial.
Save.
```

et

```
# lemmify [2;1;1;1;1] "plus_n_Sm";;
Theorem  plus_n_Sm : (n,q:nat)(S (plus q n))=(plus q (S n)).
Intros n q .
Elim q.
(Simpl; Trivial).

Intros n H0.
Simpl.
(Rewrite H0; Trivial).
Save.
```

On peut nettoyer le script de départ pour obtenir:

```
Lemma comPlus :(p,q:nat)(plus p q)=(plus q p).
Induction p.
Simpl.
Intros;Apply plus_n_O.
Intros n H.
Simpl.
Intro q.
Rewrite (H q).
Intros;Apply  plus_n_Sm.
Save.
```

♢

Dans l'environnement CtCoq la sélection du sous-but se fait par un simple clic à la souris et seul le nom du nouveau lemme est demandé à l'utilisateur. Les lemmes générés sont insérés automatiquement dans le script. Nous verrons néanmoins que pour que l'on puisse affirmer que le script reconstruit est une preuve valide du lemme généré il faudra le rejouer.

5.1.1 Principe de fonctionnement

Pour modéliser ces transformations nous enrichissons le modèle d'arbre de preuve du § 3.1. L'ensemble des buts étant noté $\mathcal{BUT}$, on considère un but comme un séquent $\Gamma, E \vdash g$. On appelle Γ la base locale d'hypothèses ou contexte local. E est appelé un environnement: c'est une liste de noms associés aux résultats précédemment prouvés. La notion d'environnement est étudiée dans la seconde partie de cette thèse.

L'environnement est fixé au début de la preuve et est donc le même pour tous les sous-buts de l'arbre de preuve (nous l'omettrons désormais dans notre représentation des

séquents). Le contexte local, quant à lui, évolue à chaque étape de la construction de la preuve.

Au commencement d'une preuve, Γ doit être vide. Donc dans un but associé à la racine d'un arbre de preuve, Γ est vide. Les règles d'introduction et d'élimination du $\forall$, permettent d'introduire ou de supprimer de nouvelles hypothèses dans Γ.

Les hypothèses introduites dans Γ peuvent être utilisées pour prouver g mais toutes ne sont pas forcément nécessaires. Si la preuve est complète (s'il n'y pas de but ouvert) on peut déterminer quelles hypothèses sont réellement utilisées dans la preuve.

On se donne une fonction $simplifier : \mathcal{BUT} \rightarrow \mathcal{BUT}$ qui associe à un but $\Gamma \vdash g$ le but $\Gamma' \vdash g$ tel que $\Gamma' = \{X \subseteq \Gamma | X \text{est utilisé dans la preuve de } g\}$. Γ' est donc le contexte minimal dans lequel la preuve de g reste valide. Cette notion de validité peut se définir à deux niveaux:

- Dans un système admettant un objet preuve comme Coq, cela peut être la validité du terme de preuve (i.e. le fait que dans le contexte Γ', son type est g).
- De façon plus générale on peut définir la validité d'une preuve par rapport au script, c'est-à-dire par le fait que si l'on rejoue le script avec le contexte local simplifié, il produira le même résultat qu'avec le contexte non simplifié.

Le processus de *lemmification* peut se décomposer de la manière suivante:

1. Sélectionner un nœud, et récupérer le sous-script de tactique (S) qui lui est associé.
2. Récupérer le but $B = \Gamma \vdash g$ associé au nœud que l'on a sélectionné.
3. Si la preuve est complète, simplifier B en $B' = \Gamma' \vdash g$
4. Fabriquer l'enoncé généralisé (c'est-à-dire un but dans lequel on a passé toutes les hypothèses de Γ' à droite du signe $\vdash$).
5. Compléter le script de preuve pour ajouter les commande `Intros` correspondant aux généralisations effectuées.

La première étape se fait à l'aide des outils du chapitre précédent, la seconde et la troisième sont de simples appel aux fonctions *but* et $simplifier$. La dernière étape consiste à rajouter en tête du script les commandes d'introduction qui nous ramènent dans l'état B'.

5.1.2 Idée de l'implémentation dans Coq et CtCoq

Les fonctionnalités du chapitre précédent fournissent le chemin, le sous-arbre et le but. **Le principe est le suivant:**

1° Retrouver dans le script de preuve `P`, la tactique `T` ayant attaqué le sous-but `SG`. Pour cela l'utilisateur utilisera les outils de navigation du chapitre précédent. Une fois la tactique sélectionnée par l'utilisateur le reste de la génération de lemme (donc toutes les étapes qui suivent) est automatique.

2° Retrouver la base `BH` d'hypothèses (contexte local) au moment de l'application de cette tactique `T`.

3° Retrouver dans le terme de preuve correspondant au script de preuve `P` le sous-terme de preuve `ST` correspondant au script de preuve SSG du sous-but `SG`.

4° Pour calculer les hypothèses réellement utilisées dans la preuve de `SG`, parcourir le terme de preuve `ST` et rechercher les identificateur libres dans ce terme; faire l'intersection de l'ensemble d'identificateurs obtenus avec l'ensemble des identificateurs référençant les hypothèses de `BH`. Soient $BU_1 : T_1 \ldots BU_m : T_m$ les hypothèses correspondant cette intersection.

5° Créer le Lemme `NL` et le script de preuve correspondant:

```
Lemma NL := (BU1:T1)...(BUm:Tm)(SG).
Intros BU1,...,BUm.   (réintroduction des hypothèses abstraites)
SSG
```

6° Pour prouver le lemme sans avoir à rejouer tout ce script on redonne au système le λ-terme `ST`.

```
Proof [BU1:T1]...[BUm:Tm]ST.
```

5.1.3 Faiblesse de notre approche

Problème de nommage des hypothèses

Dans [18] Paul E. Black et Phillip J. Windley étudient les problèmes liés à l'accès aux hypothèses en HOL. Ils présentent plusieurs façons de les référencer, par leur position dans la liste d'hypothèses, en les citant explicitement, en les nommant (ce qui se fait en Coq) ou en filtrant la liste d'hypothèses avec un motif donné. Ils affirment que le nommage explicite est la meilleure approche à long terme et la moins sensible aux modifications futures.

Dans Coq les hypothèses sont nommées lors de leur introduction dans le contexte local et l'accès à ces hypothèses se fait par référence au nom introduit. Ce nommage peut être fait explicitement par l'utilisateur, ou laissé au système en écrivant simplement `Intros`. Le comportement d'`Intros` pour le nommage des hypothèses est le suivant:

> Si le but courant est un produit dépendant `(x:T)U` et que `x` est un nom qui n'existe pas dans le contexte courant, alors `Intros` met `x:T` dans le contexte local sinon elle met `xn:T`, où n est un entier tel que `xn` soit un nom nouveau. [...] Si le but est un produit non dépendant `T->U` alors `Intros` ajoute au contexte local `Hn:T` (si `T` est de type `Prop` ou `Set`) ou `Xn:T` (si le type de `T` est `Type`) ou `n` est un entier tel que `Hn` et `Xn` soient des noms nouveaux.

Le problème de ce système de nommage (qui est également utilisé par les autres tactiques qui introduisent des hypothèses dans le contexte local comme `Split`, `Inversion` ou `Program`) est que les noms introduits sont dépendants du contexte.

Ainsi, si on essaye de rejouer le même script dans un contexte différent, les noms des hypothèses peuvent être différents et les tactiques qui les utilisent ne feront alors plus référence aux bonnes hypothèses. Le script ne sera plus valide.

Exemple 5.2

Ainsi si dans l'exemple 5.1 les hypothèses introduites n'étaient pas explicitement nommées, le second théorème isolé (`plus_n_Sm`*) serait:*

```
Theorem  plus_n_Sm : (n,q:nat)(S (plus q n))=(plus q (S n)).
Intros n q.
Elim q.
(Simpl; Trivial).

Intros.              (* on ne précise pas que l'hypothèse introduite doit *)
Simpl.               (* être nommé H0 *)
(Rewrite H0; Trivial).   (*Rewrite utilise une hypothèse qui n'existe plus*)
Save.
```

Il ne pourra être rejoué. Les trois premières lignes de script repassent sans problèmes et produisent le même résultat que précédemment. Après leur exécution, le contexte local ne contient que les hypothèses:

```
n : nat
q : nat
```

De sorte que la tactique `Intros` *de la quatrième ligne, nomme* `H` *l'hypothèse qu'elle nommait précédemment* `H0`*. Par suite la tactique* `Rewrite` *de la dernière ligne référence maintenant une hypothèse qui n'existe plus, ce qui provoque l'échec.* ♢

Il nous faut ici souligner les différences de style de preuve liées à l'utilisation de l'environnement CtCoq.

Différences dans les nommages des hypothèses introduites Quand on utilise Coq on a tendance à utiliser la tactique `Intros` sans argument. C'est alors le système de preuve qui nomme les hypothèses introduites. Mais ces noms n'apparaissent pas dans le script conservé par l'utilisateur. Ainsi s'il tente de rejouer son script dans un contexte légèrement différent, le système de preuve risque de nommer différemment les hypothèses introduites, de sorte que la suite du script deviendra incohérente.

Dans l'environnement CtCoq, il est bien sûr toujours possible d'utiliser `Intros` mais lorsque l'on fait les preuves en utilisant la machinerie de preuve par sélection [15], les noms des hypothèses à introduire sont calculés lors de la génération de la commande envoyée, de sorte que c'est une commande `Intros` $n_1 \ldots n_m$ qui est envoyée au système et stockée dans le script.

Remarquons que la différence entre les preuves obtenues avec un prouveur interactif utilisé en ligne de commande et le même prouveur utilisé avec une interface graphique

utilisateur est un problème très général. Il a été étudié dans un cadre générique par Merriam et Harrisson [86].

Pour essayer de rendre le nommage des hypothèses moins dépendant du contexte, on peut définir une discipline de conception et d'utilisation des tactiques introduisant des hypothèses dans le contexte local, mais, il est également intéressant de définir dans l'environnemt CtCoq les outils adéquats pour gérer les problèmes de nommage.

L'exemple 5.2 montre l'importance d'une procédure permettant de forcer le nommage explicite sur une preuve terminée. Une telle procédure utilisera les outils d'expansion de script du chapitre § 4.3. Nous avons décrit au chapitre 4 une procédure permettant de restaurer le nom des hypothèses introduites par la tactique `Intros`. Mais d'autres tactiques peuvent introduire des hypothèses dans le contexte local. La tactique `Split` est équivalente à `Intros;Apply ci`, où `ci` est le constructeur du type inductif correspondant au but courant. Le traitement fait sur `Split` peut donc être le même que celui fait pour `Intros` c'est-à-dire analyser le contexte local avant et après l'application de `Split` et remplacer dans le script `Split` par `Intros H1...Hi ...Hn;Split` où les H_i sont les noms des hypothèses introduites. On pourrait aussi prévoir d'autoriser la syntaxe `Split H1 ...Hn`. Le traitement des autres tactiques commençant par appliquer `Intros` est le même.

Les tactiques d'inversion posent plus de problèmes puisqu'elles font appel à `Intros` au milieu de leur exécution. On ne sait pas au départ combien d'hypothèses vont être introduites. L'idée est d'ajouter un argument à ces tactiques qui doit être un nom n'existant pas déjà dans l'environnement et qui sert alors de préfixe aux noms des hypothèses introduites. Cela n'as pas encore été testé, mais on montre le fonctionnement attendu sur un exemple fait à la main.

Exemple 5.3

```
Inductive Le : nat->nat->Set :=
  LeO: (n:nat)(Le O n)
| LeS: (n,m:nat)(Le n m) -> (Le (S n) (S m)).
Variable P:nat->nat->Prop.
Variable Q:(n,m:nat)(Le n m)->Prop.

Lemma ExInversion :(n,m:nat)(Le (S n) m)->(P n m).
ExInversion < Intros.
1 subgoal

  n : nat
  m : nat
  H : (Le (S n) m)
  ============================
   (P n m)


ExInversion < Inversion H mon_exemple.
1 subgoal
```

```
n : nat
m : nat
H : (Le (S n) m)
mon_exemple0 : nat
mon_exemple1 : nat
mon_exemple2 : n0=n
mon_exemple3 : (S m0)=m
mon_exemple4 : (Le n m0)
============================
 (P n (S m0))
```

Ainsi quelques soient les noms présents dans le contexte local, (pourvu qu'il ne contienne pas déjà l'identificateur `mon_exemple`), les noms d'hypothèses introduits par `inversion` sont les mêmes.

Autres problèmes

On rencontre ici aussi les deux problèmes récurrents de nos outils, les faiblesses dûes à l'utilisation des variables existentielles et des procédures de décision automatiques.

On peut essayer d'étendre le processus de lemmification à des sous-buts contenant des variables existentielles. Pour lemmifier un sous-but contenant des variables existentielles, la démarche est la même que dans le cas sans variable existentielle mais, il faut en plus réintroduire une quantification existentielle sur un objet du même type que la variable et remplacer toutes les instances de la variable dans le sous-but par cet objet. Par exemple si on veut lemmifier sur le sous-but `(S (S (plus?77 (0))))=(5)` on le transforme en `(Ex [x:nat](S (S (plus x (0))))=(5))` et on introduit en tête du script `EApply ex_intro.` Mais cela est possible que si la variable existentielle n'est pas contrainte par une tactique de rang inférieur à celle que l'on sélectionne pour la lemmification. Sinon on ne retrouvera pas forcément la même contrainte en rejouant le script issu de la lemmification.

Les problèmes lies à la sensibilité au contexte des procédures de décision automatique sont traités au chapitre 9. Disons simplement ici que dans de très rares cas où le script du sous-but lemmifié contient des opérateurs non monotones comme `Orelse` utilisés en liaison avec des procédures de décision automatique, il est possible que le script du résultat lemmifié ne puisse pas être rejoué.

5.2 Factorisation de lemme

Pendant la construction d'une preuve, il arrive que l'on rencontre plusieurs fois "le même sous-but". Il est assez rébarbatif et inefficace de faire deux fois la même sous-preuve. Nous aimerions disposer d'une commande de factorisation qui ferait une fois pour toute

la sous-preuve et l'utiliserait ensuite dans tous les sous-buts où cela est nécessaire. Nous détaillons cela sur un exemple trivial.

Exemple 5.4

On considère le lemme trivial ci-dessous:

```
Lemma factor : A\/A-> A/\A.
```

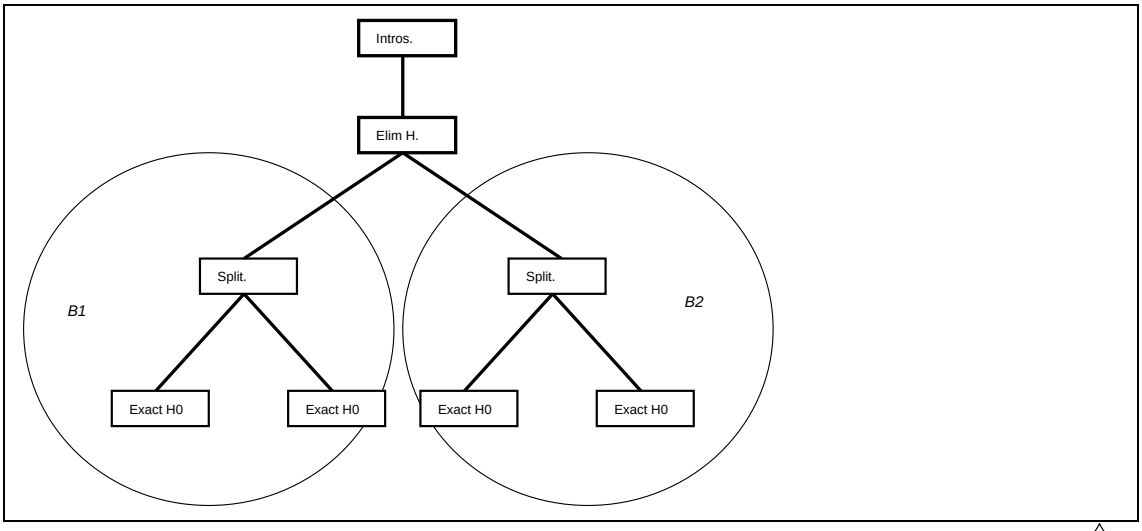
qui peut être prouvé par le script suivant

```
L1              Intros.
L2              Elim H.
L3              Split.
L4              Exact H0.

L5              Exact H0.

L6              Split.
L7              Exact H0.

L8              Exact H0.
```

Et qui correspond au λ-terme suivant:

```
factor < Show Proof.
LOC:
Subgoals
Proof: [H:A\/A]
        (or_ind A A A/\A [H0:A]<(A),(A)>{H0,H0} [H0:A]<(A),(A)>{H0,H0}
          H)
```

et à l'arbre de preuve suivant:



En fait les sous-buts résolus par les branches `B1` *et* `B2` *sont identiques et ont été générés par la tactique de la ligne 2:* `Elim H`. *Ils correspondent tous deux à:*

```
H : A\/A
============================
 A->A/\A
```

Le terme de preuve incomplet correspondant à cet état est:

```
factor < Show Proof.
LOC:
Subgoals
1092 -> A\/A->A->A/\A
1093 -> A\/A->A->A/\A
Proof: [H:A\/A](or_ind A A A/\A (?1092 H) (?1093 H) H)
```

Après résolution de la branche `B1`, *le terme de preuve est:*

```
factor < Show Proof.
LOC:
Subgoals
1162 -> A\/A->A->A/\A
Proof: [H:A\/A](or_ind A A A/\A [H0:A]<(A),(A)>{H0,H0} (?1162 H) H)
```

Le sous terme correspondant à cette branche est:

```
[H0:A]<(A),(A)>{H0,H0}
```

On constate qu'il est indépendant de H. On n'a donc pas besoin d'abstraire l'hypothèse `H`. *Le lemme à introduire est donc:*

```
Lemma intermédiaire :  A->A/\A.
```

On devrait pouvoir le prouver par le script suivant, correspondant à la branche `B1`.

```
L3           Split
L4           Exact H0

L5            Exact H0
```

Mais comme il n'y pas d'hypothèse `H` *dans l'environnement local au moment du* `Split`, *il introduit une hypothèse* `H` *au lieu de* `H0` *et par conséquent les lignes 4 et 5 échouent. Si on a pris soin de sécuriser le nommage des hypothèses on aura*

```
3: Intros H0;Split
4: Exact H0

5: Exact H0
```

Qui repasse sans problème.

En fait, pour ne pas perdre de temps à rejouer la preuve, on enverra au système, le sous terme de preuve isolé plus haut:

```
Coq < Lemma intermédiaire :  A->A/\A.
1 subgoal

  ============================
   A->A/\A

intermédiaire <  Proof [H0:A]<(A),(A)>{H0,H0}.
intermédiaire is defined
```

En remplaçant les deux branches identiques par un `Exact intermédiaire`, *le script de preuve du lemme principal (*`factor`*) devient*

```
Intros.
Elim H;Exact intermédiaire.
```

Le terme de preuve correspondant est alors

```
factor < Show Proof.
LOC:
Subgoals
Proof: [H:A\/A](or_ind A A A/\A intermédiaire intermédiaire H)
```

Si nous ne voulons pas surcharger l'environnement, en ajoutant un lemme comme `intermédiaire` *nous pourrons ajouter un* `Cut`*. Pour cela, il faut remonter à l'ancêtre commun aux deux sous-buts* `SG` *et le précéder d'un* `Cut SG`*; la preuve de* `SG` *est à faire à la fin. Pour notre exemple cela donne le script:*

```
Intros.
Cut A->A/\A.
Intros.
Elim H.
Exact H0.

Exact H0.

Intros H0;Split.      (*                                *)
Exact H0              (*                                *)
                      (* ou réutiliser le sous-terme     *)
Exact H0              (*                                *)
```

5.2.1 Principe de fonctionnement

La figure 5.1 montre le principe de fonctionnement de la factorisation une fois que l'on a detecté deux sous-buts semblables S_1 et S_2, et S_1 etant prouver.

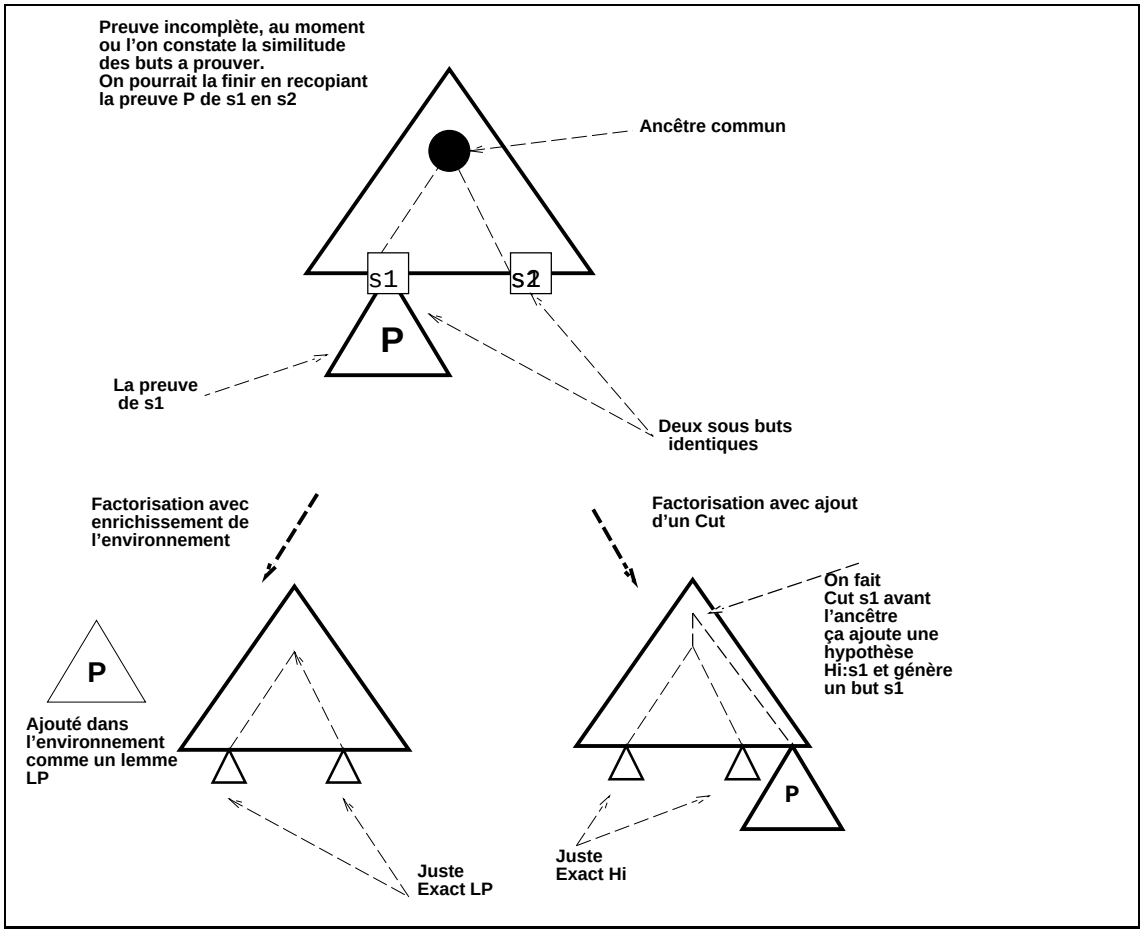


FIG. 5.1 – *Illustration du principe de factorisation*

En pratique détecter deux sous-buts identiques dans une grosse preuve est beaucoup trop coûteux puisque, dès que l'on aura plus d'une branche dans l'arbre de preuve, il faudra comparer deux à deux chacun des nœuds. Par contre si l'on sait qu'un résultat a déjà été prouvé, il est facile de le retrouver et de l'exploiter. C'est aussi l'objet de la tactique `Similar` décrite ci-dessous.

La tactique Similar

La tactique `Similar` vise, comme son nom l'indique, à exploiter les similarités de certaines sous preuves. Pour une question de coût, ces similarités qui se limitent pour l'instant à une convertibilité ne sont pas détectées systématiquement, c'est à l'utilisateur de se dire "tiens j'ai déjà prouvé quelque chose comme ça !", et d'appliquer alors `Similar`. On parcourt alors l'arbre de preuve à la recherche d'un but convertible avec le but courant. Si un tel but existe on calcule quelles hypothèses ont été nécessaires à sa preuve. Si ces hypo-

thèses sont satisfaites dans le contexte courant, on peut réutiliser la preuve qui est alors mise en facteur.

5.3 Conclusion

La lemmification permet une meilleure structuration des développements. Elle permet de faire cette structuration a posteriori. Enfin alliée à la factorisation, elle peut entraîner un gain de temps certain lorsqu'on rejoue un script et lors de la vérification du terme de preuve généré.

En effet, en Coq, une fois le terme de preuve généré, il est soumis au vérificateur qui infère son type et le compare avec le but initial. Lorsque pendant l'inférence de type, ce vérificateur rencontre un identificateur il a directement accès à son type alors que s'il rencontre le sous terme correspondant, parfois à plusieurs occurrences, il devra à chaque fois inférer le type de ce sous-terme, ce qui peut être très coûteux si le terme est gros.

Deuxième partie

Etude globale des théorèmes

Introduction

Notion de théorie

En mathématique une théorie est l'ensemble des résultats déductibles d'un système d'axiomes. Dans les assistants de preuve, la représentation de la notion de théorie est très variable. Elle dépend autant de l'état de développement des systèmes que de la logique sur laquelle ils reposent. Dans certains démonstrateurs, c'est un concept du langage avec des opérations propres (HOL a, par exemple, des commandes `open theory` et `close theory` qui permettent respectivement d'ouvrir et de fermer une théorie), dans d'autres il n'existe pas de support syntaxique associé à la notion de théorie. Elle transparaît alors simplement dans l'organisation du développement des preuves. Dans un assistant de preuve, un développement est un ensemble de définitions et de théorèmes dans lequel les parties sémantiquement liées sont regroupées entre elles. Un tel regroupement forme alors une théorie. La notion la plus appropriée pour regrouper des résultats est celle de **module**. Des systèmes comme Larch, Imps ou PVS ont une notion de théorie ou de module. Ces modules peuvent être paramètres et permettent comme dans les langages de programmation une meilleure structuration des développements. De nombreux travaux portent actuellement sur la notion de module dans les systèmes de preuve (par exemple les travaux de David Aspinall [7] pour Isabelle ou de Judicaël Courant [28] pour Coq). Cependant, actuellement dans la majorité des systèmes, on doit se contenter de regrouper les résultats par fichiers et répertoires. En Coq ces fichiers sont compilables séparément et le système fournit aussi un mécanisme de `Section` qui permet d'organiser un peu plus finement les résultats mais nous ne considérons pas cette possibilité pour le moment. Dans la suite de ce chapitre, nous appellerons donc théorie un fichier ou un ensemble de fichiers.

Dépendances au niveau des théories

Avec la conception minimaliste de théorie que nous avons retenue les dépendances entre théories se ramènent à un calcul de dépendance entre fichiers.

Maintenir la cohérence d'un ensemble de fichiers lorsque des modifications sont faites dans l'un (ou plusieurs) d'entre eux est une tâche difficile. Dans l'environnement *Unix*, des programmes comme *make* [44] sont classiquement utilisés pour imposer le maintien de la cohérence en gérant l'ordre de recompilation des fichiers et en recompilant ceux qui doivent l'être en cas de modification. Les relations de dépendance entre les fichiers sont déclarées par l'utilisateur dans un fichier de description utilisé par *make*, le *Makefile*. Des outils pour calculer les dépendances entre les fichiers dans un format lisible par *make* sont désormais classiques. On a par exemple *makedepend* pour les dépendances entre fichiers *C*. Ce programme parcourt les sources et utilise les `#include` pour calculer les dépendances. Dans le monde des langages fonctionnels, comme par exemple *Caml* [140, 71] on dispose d'outils comme *camldep*, ou dans le monde des systèmes de preuves comme *Coq* on dispose du programme *coqdep*. Ils procèdent tous de la même façon, en recherchant les unités de compilation externes pour fournir une description des dépendances au format *Make*. C'est une étude purement syntaxique des fichiers sources.

L'utilisation classique de ces outils de calcul permet la compilation séparée des différentes bibliothèques d'un système. Mais comme le montrent les exemples de la fin de ce chapitre, ils peuvent aussi servir de base à un système de visualisation graphique des relations entre théories, offrant ainsi à l'utilisateur une meilleure compréhension de ses développements.

Les outils travaillant avec une granularité plus fine que celle des fichiers sont moins classiques. Dans le cadre des langages de programmation on peut mentionner les systèmes d'analyse de flot de données. Dans le cas particulier des langages fonctionnels signalons aussi le travail de Boquist et Sparud [19] qui décrit un environnement de développement pour *Standard ML* [6] sous *GNU Emacs* dans lequel ils maintiennent les dépendances entre fonctions et les utilisent pour simuler une liaison dynamique alors que la liaison de *SML* est statique. Ainsi, lorsqu'une fonction est modifiée, toutes les fonctions qui l'utilisent feront référence à la nouvelle version de la fonction modifiée.

Au chapitre 6 nous nous intéressons au calcul des dépendances entre théorèmes et définitions dans les systèmes d'aide à la preuve, c'est-à-dire à l'intérieur d'une théorie donnée. Nous présentons une modélisation de ces dépendances au travers de la notion de graphe de dépendance, ainsi que différents moyens de les calculer, puis nous présentons un panorama d'outils les utilisant (visualisation, "Reset logique", coupe de théorie, déplacement et réorganisation de code, propagation de modifications ...) . Nous concluons le chapitre en présentant un outil de visualisation des dépendances entre théories (fichier, répertoire) et de visualisation des hiérarchies de type. Le chapitre 7 présente en détail une première application utilisant le graphe de dépendance entre objets, la coupe de théorie qui est une notion analogue à celle de coupe de programme (slicing) en génie logiciel. Nous montrons les lacunes de notre calcul de dépendance mises en évidence lors de la réalisation de cet outil (et que l'on retrouvera dans la réalisation des différents outils des chapitres suivants) et proposons plusieurs alternatives. Le chapitre 8 s'intéresse à la propagation des modifications. L'objectif est de savoir lorsqu'un théorème, une preuve ou une définition est modifié, quels sont les objets (preuve, définition...) qui doivent être vérifiés (dans le cas des outils d'aide à la preuve, on dit qu'ils doivent être rejoués). On développe ensuite un outil interactif d'aide à la propagation des modifications. Notons que la méthode proposée est aussi

valable dans le cadre les langages fonctionnels mais que, comme nous le montrons, dans le cas particulier des systèmes de preuve, le mode de propagation des modifications permet d'en limiter la portée. Enfin, le chapitre 9 introduit les problèmes liés aux procédures de décisions automatiques et discute les solutions proposées. .

Chapitre 6

Calcul et applications des dépendances au niveau des théories

6.1 Introduction

Ce chapitre présente une modélisation des dépendances entre objets d'une théorie et les différents moyens de les calculer.

Dans le cas local, étudié au chapitre 3, alors que l'utilisateur manipule un script de preuve nous utilisons un arbre de dérivation pour représenter une preuve en mettant en valeur les dépendances entre les différentes partie de la preuve. Ici alors que l'utilisateur manipule un script dans lequel il définit les objets les uns après les autres nous utilisons un graphe pour représenter la structure d'une théorie et les dépendances entre les différents objets.

6.2 Modèle abstrait de théorie

Note: Dans tout ce document on emploie indifféremment les mots contexte et environnement.

On définit l'ensemble des définitions, noté $\mathcal{ED}$, par la donnée d'un triplet $(\mathcal{N} \times \mathcal{SPEC} \times \mathcal{R})$, ou $\mathcal{N}$ est un ensemble d'**identificateurs**, $\mathcal{SPEC}$ un ensemble de **spécifications** et $\mathcal{R}$ un ensemble de **réalisations**.

Exemple 6.1

Dans le système Coq, l'identificateur est le nom de l'objet que l'on est en train de définir. Il est unique. La spécification correspond au type de l'objet et la réalisation au terme

qui lui est associé (terme de preuve ou terme fonctionnel). On peut la visualiser par la commande `Print`. *On a par exemple les définitions suivantes:*

```
plus =                                        (* l'identificateur *)
Fix plus
  {plus [n:nat] : nat->nat :=
     [m:nat]Cases n of                        (* la réalisation *)
               O => m
             | (S p) => (S (plus p m))
             end}
     : nat->nat->nat                          (* la spécification *)
```

ou

```
plus_sym =
[n,m:nat]
 (nat_ind [n0:nat](plus n0 m)=(plus m n0) (plus_n_O m)
   [y:nat; H:((plus y m)=(plus m y))]
    (eq_ind nat (S (plus m y)) [n0:nat](S (plus y m))=n0
      (f_equal nat nat S (plus y m) (plus m y) H) (plus m (S y))
      (plus_n_Sm m y)) n)
     : (n,m:nat)(plus n m)=(plus m n)
```

On constate par ailleurs qu'il n'y pas de différence entre la définition d'un théorème ou d'une fonction. Il est d'ailleurs possible de définir la fonction `plus` *comme une preuve particulière d'un théorème d'énoncé* `nat->nat->nat`. ♢

On introduit une relation $\mathcal{R}_{dep}$ sur $\mathcal{ED} \times \mathcal{ED}$ dite **relation de dépendance directe**. Pour deux objets d_1 et d_2 on a $d_1 \mathcal{R}_{dep} d2$ si la définition (c'est-à-dire la spécification ou la réalisation) de d_2 utilise d_1. On se donne une fonction $dep :\ \mathcal{ED} \rightarrow \mathcal{P}(\mathcal{ED})$ dite **fonction de dépendance directe** qui associe à toute déclaration l'ensemble des déclarations dont elle dépend directement.

On a évidemment:

$$\forall d_1, d_2 \in \mathcal{ED}, d_1 \mathcal{R}_{dep} d2 \Leftrightarrow d_1 \in dep(d_2)$$

.

La fermeture transitive de $\mathcal{R}_{dep}$ nous donne la relation de pré-ordre partiel $\prec_{dep}$ dite relation de dépendance. On a donc:

$$\forall d_1, d_2 \in \mathcal{ED},\ d_1 \prec_{dep} d_2 \Leftrightarrow d_1 \mathcal{R}^{+}_{dep} d_2$$

.

On définit aussi par transitivité la fonction *depAll* qui associe à une déclaration toutes les déclarations dont elle dépend. On a:

$$depAll(d) = \{d_i | d_i \prec_{dep} d\}$$

Nous appellerons **théorie** tout ensemble de définitions partiellement ordonné par la relation $\prec_{dep}$. On note $\mathcal{TH}$ l'ensemble des théories.

Pour représenter les liens entre les différents objets de la théorie on utilise un graphe de dépendance. C'est un graphe orienté sans circuit (GOSC). Un tel graphe est constitué d'un ensemble de sommets (ou nœuds) S et d'un ensemble d'arcs A. Un arc est un couple (s,t) de sommets. Lorsque l'on utilise un graphe dans la représentation d'une théorie, chaque sommet de ce graphe est une référence à un objet de la théorie. La relation binaire $\prec_{dep}$ permet de définir les arcs. Il existe un arc (d_1,d_2) si $d_1 \mathcal{R}_{dep} d_2$. S'il existe un arc (d_1,d_2) on dit que d_2 est adjacent à d_1.

Un développement de preuve que nous modélisons à travers la notion d'environnement, est un ordonnancement des définitions d'une théorie. Au fur et à mesure que des objets sont définis par l'utilisateur du système de preuve, ils sont ajoutés dans un **environnement**. Nous représentons l'ordre de définition des objets d'une théorie avec une opération notée `rang`. La fonction $rang : \mathcal{ED} \times \mathcal{TH} \rightarrow integer$ associe à toute définition d'une théorie son indice dans la liste des objets de l'environnement.

Cela permet de définir un ordre total $<_{hist}$ entre les objets d'une théorie par:

$$\forall d_1,d_2 \in \mathcal{ED}, d_1 <_{hist} d2 \Leftrightarrow rang(d_1) < rang(d_2)$$

.

Un environnement est donc simplement un ensemble totalement ordonné de définitions géré par le système de preuve.

Mais les scripts correspondant au développement d'un utilisateur se répartissent généralement sur plusieurs fichiers, de sorte que la position dans l'environnement ne se traduit pas forcément par une position dans un fichier. Pour les activités de maintenance il est important de savoir dans quel fichier un objet a été défini. On se donne donc une fonction $Fichier\_Definition : \mathcal{ED} \rightarrow Fichier$ qui associe à un objet le nom du fichier où il a été défini. Remarquons que cette fonction nous permet de calculer trivialement à partir du graphe de dépendance d'une théorie, dont les définitions sont reparties sur plusieurs fichiers, le graphe de dépendance entre ces fichiers.

Comme nous l'avions fait pour les notions d'arbre et de script dans le cas local, on peut ici mettre en correspondance, les notions d'environnement et de théorie. Pour associer une théorie à un environnement on fera un calcul de dépendance. Ce calcul se fait à l'aide de la fonction *dep*. La mise en œuvre de ce calcul est discutée à la section 6.4.

Pour toute théorie, on peut trouver un ordre total $<$, compatible avec l'ordre partiel $\prec_{dep}$ de la théorie. Autrement dit on peut associer une liste $(d_1 \ldots d_n)$ croissante des objets

de la théorie telle que si $d_i \prec_{dep} d_j$ alors $i < j$. Une telle suite est un environnement. Trouver un tel ordre s'appelle faire un tri topologique. Mais un tel ordre n'est pas unique. Ainsi plusieurs environnements peuvent correspondre à une même théorie.

Pour pouvoir ajouter un objet à un environnement, il faudra que tous les objets dont il dépend soient déjà présents dans l'environnement. Pour exprimer ce fait on introduit un prédicat $isValid : \mathcal{ED} \times \mathcal{E}nv \rightarrow boolean$ qui vérifie qu'une déclaration est valide dans un environnement donné. En particulier:
si $isValid(d,e) = true$ alors $\forall d_i \in dep(d), d_i \in e \wedge d_i <_{hist} d$

Un environnement est dit valide si toutes ses déclarations sont valides. Notons que dans un système comme Coq, il n'est pas possible de construire à la main une théorie non valide (c'est-à-dire à laquelle on ne peut pas associer un environnement valide) mais lorsque l'on transforme une théorie, à l'aide outils de manipulation de l'environnement, il faut prendre garde à ne pas l'invalider. La section 6.3 définit une notion de transformation valide qui permet de définir une classe de transformations préservant la validité de l'environnement.

Les relations et fonctions de dépendance que nous avons introduites permettent de déterminer de quoi dépend un objet donné. Cette information est statique en ce sens qu'elle est déterminée lors de la définition de l'objet. La notion réciproque qui permet de déterminer quelles déclarations dépendent d'une déclaration donnée est une notion dynamique dans la mesure où l'ensemble image croît quand l'environnement croît. On définit ainsi, pour un environnement donné la relation: ${\mathcal{R}_{dep}}^{-1}$ par:

$$\forall d_1, d_2 \in \mathcal{ED}\ d_1 {\mathcal{R}_{dep}}^{-1} d_2 \Leftrightarrow d_2 \mathcal{R}_{dep} d_1$$

et la fonction

$$dep^{-1}(d_1) : \{d_2 | d_2 \mathcal{R}_{dep} d_1\}$$

6.3 Manipulation d'environnements

Soit $\mathcal{TRANSF}$ l'ensemble des transformations sur un environnement. Soit $Transf : \mathcal{E}nv \rightarrow \mathcal{E}nv$ une transformation. On dira qu'une transformation est valide si elle est compatible avec les relations d'ordre partiel existantes sur les théories associées à l'environnement de départ, c'est-à-dire si:

$$\forall e \in \mathcal{E}nv, \forall d \in Transf(e), \forall d_i \in dep(d), d_i \in Transf(e) \wedge rang(d_i) < rang(d)$$

ou encore si $\forall d \in Trans(e), isValid(d, Transf(e))$.

6.4 Implémentation du calcul des dépendances entre objets d'une théorie

Nous présentons maintenant l'implémentation des fonctions de calcul de dépendance, *dep* et *depAll*. La première question que l'on doit se poser lorsque l'on veut calculer les

dépendances entre les objets d'une théorie est:**"Sur quoi calculer les dépendances?"**
Deux réponses sont possibles:

1. Sur le script.
2. Sur l'énoncé et l'objet preuve.

Dans un système comme Coq lorsque qu'une preuve est finie, la seule chose qui est conservée par le système est le terme de preuve. D'autre part, les scripts écrits par l'utilisateur sont conservés dans les fichiers texte (suffixés par `.v`) tandis que la version compilée de ces fichiers (suffixés par `.vo`) ne contient que les termes de preuve. Pour répondre à la question précédente, nous reprenons un de nos exemples favoris: les listes et plus particulièrement le théorème d'associativité à gauche de la concaténation. On se place dans un environnement où tous les fichiers du `Prelude` de Coq[1] ont été chargés et où on a défini les listes (`list`), la concaténation sur les listes (`app`) et les deux théorèmes suivants.

```
app_nil_end : (l:list)(l=(app l nil)).
app_ass : (l,m,n : list)(app (app l m) n)=(app l (app m n)).
```

- **Calcul des dépendances sur le Script.**
  On peut énoncer et prouver l'associativité à gauche par le script suivant:

  ```
  Lemma ass_app : (l,m,n : list)(app l (app m n))=(app (app l m) n).
  Proof.
  Intros.
  Apply sym_equal.
  Auto.
  Save.
  ```

  Les identificateurs libres qui apparaissent dans le script sont:
  - `list`, qui correspond à la définition inductive des listes
  - `app`, qui correspond à la définition de la fonction de concaténation des listes
  - `sym_equal`, qui correspond à la symétrie de l'égalité

  Ainsi à partir de ce script on peut savoir que notre théorème utilise la définition des listes, de la concaténation et la symétrie de l'égalité. Mais on ne sait pas ce que fait la tactique `Auto`. C'est "une boîte noire" qui correspond à une procédure de décision automatique. Grossièrement cette procédure essaye d'appliquer les théorèmes précédemment définis et marqués comme pouvant être utilisés par les procédures de décision automatique[2]. Donc notre théorème peut potentiellement dépendre de tout ce qui a été précédemment défini et marqué.
- **Calcul des dépendances sur le terme de preuve.**
  Nous considérons maintenant le terme de preuve de l'associativité à gauche de la concaténation des listes. C'est le terme associé à l'identificateur `ass_app` soit :

  ```
  ass_app =
  ```

1. Le Prelude est un ensemble de fichiers chargés par défaut et qui contiennent les définitions de base notamment la définition de l'égalité, des entiers et quelques unes de leurs propriétés de base.
2. En Coq ce marquage se fait à l'aide de la commande `Hint`

```
[l,m,n:list]
(sym_equal list (app (app l m) n) (app l (app m n)) (app_ass l m n))
     : (l,m,n:list)(app l (app m n))=(app (app l m) n)
```

Les identificateurs libres qui apparaissent dans le terme sont les suivants:

- `list`, qui correspond à la définition inductive des listes
- `app`, qui correspond à la définition de la fonction de concaténation des listes
- `sym_equal`, qui correspond à la symétrie de l'égalité
- `app_ass`, qui correspond à l'associativité à droite de la concaténation.

On constate qu'il apparaît un identificateur de plus que dans le calcul fait sur le script. Cet identificateur `app_ass` est associé à un théorème qui a été utilisé par la tactique `Auto`. On en déduit que l'objet associé à l'identificateur `ass_app` dépend des objets associés aux identificateurs `list`, `app`, `sym_equal` et `app_ass` (par abus de langage on dira souvent que `ass_app` dépend de `list`, `app`, `sym_equal` et `app_ass`).

Il semble donc que le terme de preuve soit une structure plus "informative" que le script, même si nous verrons plus loin que cela reste à nuancer.

C'est donc sur ce terme de preuve que nous avons d'abord basé notre calcul de dépendance entre objets. Mais il est important de garder à l'esprit que notre objectif est principalement d'utiliser ces dépendances pour faciliter la maintenance et la manipulation du **script**.

D'autre part même lorsque l'on calcule les dépendances sur le terme de preuve, il faut systématiquement prendre en compte les identificateurs qui apparaissent dans l'énoncé mais qui peuvent très bien ne plus apparaître dans le terme.

Appliquant récursivement le calcul fait sur les termes à tous les termes de l'environnement, on peut construire le graphe de dépendance d'une théorie. La figure 6.1 montre le graphe de dépendance d'une infime partie de la théorie des listes.

6.4.1 Implémentation dans Coq

Comme nous venons de le voir notre calcul de dépendance va se faire sur l'énoncé et le terme de preuve. Au cours d'une session de preuve, ces termes sont ajoutés dans un environnement au fur et à mesure de leur définition. L'environnement a donc grossièrement une structure de liste. Un objet est accessible dans l'environnement grâce à un chemin d'accès, (la terminologie Coq parle de `section_path`). Ce chemin d'accès est construit à partir du nom de l'objet et celui de la section ou du fichier où il est défini (pour une description plus fine on se reportera au manuel de référence de Coq).

Pour calculer les dépendances d'un objet on reconstruit donc (à partir du nom de l'objet et du nom des `sections` où cet objet est défini) le `section_path` qui lui correspond. Cela nous permet d'accéder au terme qui lui est associé. Un parcours de ce terme nous permet de savoir de quels autres objets il dépend.

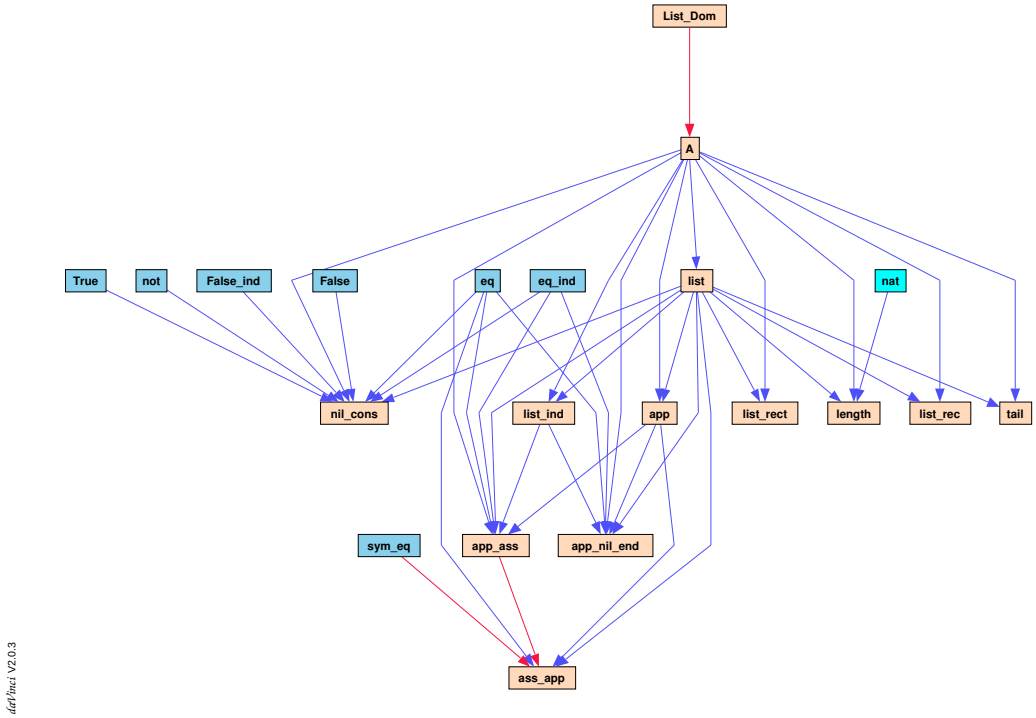


FIG. 6.1 – *Graphe de dépendance d'une partie de la théorie des listes*

Pour la majorité des outils de maintenance que nous avons définis, le graphe de dépendance sera calculé une fois le développement de preuve terminé mais on peut aussi vouloir utiliser ce graphe au cours du développement comme une aide à la compréhension.

On propose donc deux modes de calcul des dépendances, incrémentalement de façon interactive ou en "batch" sur un développement fini.

6.4.2 Mode interactif

En mode interactif dès qu'un identificateur est ajouté à l'environnement on calculera ses dépendances et on l'ajoutera au graphe de dépendance. On pourra visualiser ces dépendances graphiquement ou par une navigation dans le script. Cette visualisation graphique peut servir de support à une interface de manipulation de l'environnement en cours de développement. Nous proposons par exemple à la section 6.5.1 un **Reset Logique** qui est l'analogue, au niveau des théories du retour-arrière logique du chapitre 3.

6.4.3 Mode batch

Le mode batch travaille sur les fichiers. On peut choisir de travailler avec les fichiers de script (suffixés par (.v)) en se contentant de les rejouer et en utilisant la machinerie du mode interactif pour calculer les dépendances.

On peut aussi travailler sur les fichiers compilés, (suffixés par `.vo`). Dans ce cas tous les objets qui on été définis dans le développement correspondant au fichier compilé sont ajoutés directement dans l'environnement. On parcourt alors cet environnement en calculant les dépendances de chaque objet.

6.4.4 Implémentation des graphes

Il existe plusieurs façons de représenter les graphes orientés (voir par exemple [3]). On peut utiliser une matrice d'adjacence, c'est-à-dire une matrice carrée M de booléen indicée par les sommets du graphe, telle que $M(s_1,s_2)$ soit vraie si s_2 est adjacent à s_1. L'avantage d'une telle représentation est que le temps d'accès à un élément est un temps constant indépendant du nombre de nœuds et d'arêtes. Il y a néanmoins deux inconvénients à cette représentation: elle se prête mal à l'ajout de nœuds, elle s'accompagne d'une complexité en espace de $\Omega(n^2)$ même si le graphe possède beaucoup moins de n^2 arêtes ce qui est généralement le cas des graphes de dépendance que nous manipulons. La lecture d'une matrice se fait en $O(n^2)$, ce qui exclut d'office les algorithmes en $O(n)$ de manipulation de graphe contenant $O(n)$ arcs.

On préfère donc utiliser une représentation par liste d'adjacence. La liste d'adjacence l d'un sommet s est une liste des nœuds adjacents à s. Le graphe est représenté par une liste de couples, (s,l). La complexité en espace est alors proportionelle à la somme du nombre de nœuds et d'arêtes. Le désavantage est néanmoins que tester si deux nœuds sont adjacents peut maintenant atteindre une complexité en $O(n)$.

Pour stocker les graphes on peut utiliser la représentation sous forme de liste d'adjacence commune aux fichiers de dépendance utilisable par *Make*. La figure 6.2 montre un exemple d'un fichier de dépendance pour la théorie des listes de la bibliothèque standard. Il est alors possible d'utiliser l'outil **dependtohtml** décrit en 6.6 pour fournir une image du graphe de dépendance sensible à la souris.

6.5 Brève présentation des principales applications réalisées en utilisant le graphe de dépendance

Nous passons maintenant en revue quelques transformations sur les environnements et les théories, dont certaines seront détaillées dans les chapitres suivants.

```
Le/le_elim_rel: Le/le_elim_rel
Le/le_n_O_eq: Le/le_n_O_eq
Le/le_antisym: Le/le_antisym
Le/le_Sn_n: Le/le_Sn_n
Le/le_Sn_O: Le/le_Sn_O
Le/le_S_n: Le/le_S_n
Le/le_trans_S: Le/le_trans_S
Le/le_pred_n: Le/le_pred_n
Le/le_O_n: Le/le_O_n
Le/le_n_Sn: Le/le_n_Sn
Le/le_trans: Le/le_trans
Le/le_n_S: Le/le_n_S
List/incl_app: List/incl_app List/list List/A List/In List/app List/in_app_or List/incl
List/incl_cons: List/incl_cons List/A List/list List/In List/incl
List/incl_appr: List/incl_appr List/list List/incl List/A List/In List/in_or_app List/app
List/incl_appl: List/incl_appl List/list List/incl List/A List/In List/in_or_app List/app
List/incl_tran: List/incl_tran List/list List/incl List/A List/In
List/incl_tl: List/incl_tl List/A List/list List/incl List/In List/in_cons
List/incl_refl: List/incl_refl List/list List/A List/In List/incl
List/incl: List/incl List/list List/A List/In
List/in_or_app: List/in_or_app List/list List/A List/list_ind List/In List/app
List/in_app_or: List/in_app_or List/list List/A List/list_ind List/In List/app
List/in_cons: List/in_cons List/A List/list List/In
List/in_eq: List/in_eq List/A List/list List/In
List/In: List/In List/A List/list
List/lel_nil: List/lel_nil List/list List/list_ind List/lel List/A Le/le_Sn_O List/length
List/lel_tail: List/lel_tail List/A List/list List/length Le/le_S_n List/lel
List/lel_cons: List/lel_cons List/A List/list List/length List/lel
List/lel_cons_cons: List/lel_cons_cons List/A List/list List/length Le/le_n_S List/lel
List/lel_trans: List/lel_trans List/list List/length Le/le_trans List/lel
List/lel_refl: List/lel_refl List/list List/length List/lel
List/lel: List/lel List/list List/length
List/length: List/length List/list List/A
List/nil_cons: List/nil_cons List/A List/list
List/tail: List/tail List/list List/A
List/ass_app: List/ass_app List/list List/app List/app_ass
List/app_ass: List/app_ass List/list List/list_ind List/app List/A
List/app_nil_end: List/app_nil_end List/list List/list_ind List/app List/A
List/app: List/app List/list List/A
List/list_rect: List/list_rect List/list List/A
List/list_rec: List/list_rec List/list List/A
List/list_ind: List/list_ind List/list List/A
List/list: List/list List/A
List/A: List/A List/List_Dom
List/List_Dom: List/List_Dom

Chaque objet est précédé de son fichier de définition (ici Le.v ou List.v). Par défaut les objets du Prélude ne sont pas mentionnés.
```

FIG. 6.2 – *Dépendance entre objets de la théorie des listes au format Make*

6.5.1 Visualisation et interface

La figure 6.1, page 103 donne une idée de la représentation graphique d'une théorie. Pour la mise en œuvre nous avons utilisé le système daVinci déjà mentionné au chapitre 3 dans le cas des arbres.

En mode interactif, on peut avoir une interface utilisant la représentation graphique de la théorie pour manipuler l'environnement. Notre première expérience dans ce domaine est l'implémentation du *Reset logique*.

Le Reset Logique:

Le Reset Logique est le pendant du retour-arriere logique développé pour les théorèmes dans la première partie de cette thèse.

Normalement dans le système Coq, la Commande `Reset` qui permet de supprimer un objet de l'environnement est historique. C'est-à-dire que lorsqu'on supprime un objet tout ceux qui ont été définis après lui seront aussi effacés même s'ils n'en dépendent pas.

Il nous a paru intéressant d'exploiter l'information fournie par le graphe de dépendance pour pouvoir supprimer de l'environnement uniquement les objets dépendants effectivement de l'objet à effacer.

Si les objets effacés appartiennent tous au fichier en cours de développement, ils pourront, comme nous le faisions avec les tactiques dans le cas du retour-arrière logique, être réécrits à la fin du script courant dans la fenêtre d'édition.

Néanmoins l'utilisateur ne peut pas être sûr de pouvoir les rejouer sans problème. Le déplacement des théorèmes entraîne en effet un changement du contexte dans lequel ils seront rejoués et cela peut changer le comportement des procédures de décision automatique. Ce problème de sensibilité au contexte des procédures de décision automatique que nous rencontrons dans presque tous nos outils est étudié en détail au chapitre 9.

Nous avons expérimenté le *reset logique* en réalisant un prototype basé sur notre interface Ocaml/daVinci.

6.5.2 La coupe (ou épuration) de théorie

Nous commencerons par une comparaison avec la notion de coupe de programme (slicing) [141] issue des méthodes d'analyse de flot en génie logiciel.

La coupe de programme peut se résumer à une question du genre "de qui dépend la valeur de X à la ligne 45 ?". Un "slice" de programme est un programme exécutable obtenu en effaçant du programme original tout ce qui est sans effet sur la valeur spécifiée.

De façon similaire dans le domaine des preuves formelles nous pouvons nous poser la question "de qui dépend la validité du théorème T dans la théorie Z". Et on veut, étant donnée une théorie, prendre un théorème particulier et ne conserver de la théorie que ce qui était effectivement utile à la preuve du théorème sélectionné.

On ne conserve donc de nos scripts que les théorèmes, définitions etc. dont dépend (transitivement) le théorème que l'on a choisi. On obtient la "théorie minimale" sur laquelle repose ce théorème, autrement dit une coupe (slice) de la théorie de départ par rapport au théorème étudié.

Notons que la notion de minimalité est relative au développement initial. Le seul ordre dont on dispose naturellement sur "l'ensemble des scripts" est un ordre partiel donné par la relation d'inclusion (qui s'exprime dans ce cas par "`est un sous-script de`"). Il existe une infinité de scripts incomparables pour cette relation pouvant mener à la preuve d'un même théorème.

La coupe de théorie correspond à un affaiblissement du contexte global. Cela correspond au calcul du contexte minimal dans lequel une déclaration d reste valide. Ce contexte

est trivialement donné par la fonction *depAll*. L'épuration de théorie est évidement une transformation valide (au sens de 6.3), néanmoins lorsque nous avons implémenté notre outil, il est apparu que les résultats obtenus ne pouvaient pas toujours être rejoués. Les problèmes sont liés à notre façon de calculer les dépendances (sur l'objet preuve alors que nous manipulons le script). Le chapitre 7 présente en détail la mise en œuvre de l'outil, les problèmes rencontrés et les solutions proposées.

6.5.3 Modification de théorie

Pour continuer notre parallèle avec les notions de l'analyse de flot nous nous sommes ensuite intéressé à ce qui apparaît parfois sous le nom de "ripple effect analysis" [147]. C'est un peu la réciproque du "slicing". Cela cherche à répondre à des questions du genre "si le calcul de Y à la ligne 45 est changé quelles autres variables sont affectées et à quel endroit dans le programme ?".

Côté preuves formelles, nous avons donc cherché à fournir une aide à la modification de théorie. C'est-à-dire qu'étant donnée une modification sur un théorème ou une définition, on veut fournir à l'utilisateur la liste des théorèmes susceptibles de devoir être à leur tour modifiés pour rester valides. On veut aussi lui fournir des outils pour l'assister dans la propagation des modifications. Pour calculer quelles autres déclarations de la théorie dépendent des déclarations modifiées, on dispose de la fonction $dep^{-1} : \mathcal{ED} \times \mathcal{E}nv \rightarrow \mathcal{P}(\mathcal{ED})$ et évidemment de $depAll^{-1}$. Intuitivement il semble que seuls les théorèmes qui dépendent du théorème modifié puissent être affectés. Néanmoins nous montrons plus loin que les tactiques implémentant des procédures de décision automatique (`Auto, Trivial...`) introduisent des liens plus complexes entre les objets et compliquent sensiblement les choses.

Un algorithme interactif de propagation des modifications est présenté au chapitre 8. Les outils d'aide à la modification et les problèmes liés aux procédures de décision automatique font l'objet du chapitre 9.

6.5.4 Réorganisation de développement

Assurer que des théorèmes d'un même domaine soient réellement stockés dans la même zone est important pour leur réutilisation ultérieure[3] et la maintenance de preuve à long terme.

On rencontre par exemple une situation qui conduit fréquemment à avoir des résultats mal placés quand l'utilisateur découvre qu'il manque un théorème à sa théorie au moment où il l'utilise dans un autre développement (par exemple alors qu'il développe une théorie des polynômes en utilisant la théorie des listes, il s'aperçoit qu'il lui manque une propriété des listes). La plupart du temps, il prouve alors le résultat dont il a besoin dans le contexte où il se trouve et le stocke dans la théorie qu'il est en train de développer (dans notre exemple la théorie des polynômes). Cela conduit à dupliquer du code et du travail car les

3. Même si des outils de recherche dans les bibliothèques (comme `Searchisos`) peuvent aussi aider à retrouver un résultat dans un ensemble de bibliothèques.

autres utilisateurs ne sauront pas que ce résultat a été prouvé puisqu'il n'est pas à sa place naturelle. Il est donc important de permettre de réorganiser la théorie. Mais remettre un théorème à sa place "naturelle" n'est pas toujours évident. Il faut d'abord pouvoir affirmer que le résultat à déplacer n'utilise pas de résultats de la théorie courante. Pour vérifier cela on utilisera la relation *dep*. D'autre part supprimer un théorème d'un contexte valide pour l'ajouter dans un autre contexte valide modifie sensiblement ces contextes, et comme nous l'avons mentioné au paragraphe précédent des modifications de contexte peuvent modifier le comportement des procédures de décision automatique et compromettre la validité d'un script.

La réorganisation d'un développement de preuve pouvant être vue comme un cas particulier de la modification de théorie nous ne la présenterons pas plus longuement dans ce document.

Signalons néanmoins que nous n'avons mentionné que la réorganisation d'une théorie a posteriori. Dans l'environnement Emacs pour HOL, Curzon [32] a développé la notion de "virtual theories" et implémenté une réorganisation interactive du code. Il est alors possible à tout moment d'étendre une théorie, un "censeur" se chargeant de vérifier le fondement du résultat prouvé par rapport à sa théorie, c'est-à-dire qu'il n'utilise que des résultats de la théorie où l'on veut l'ajouter. La mise en œuvre d'un tel mécanisme dans Coq se heurte au mécanisme de *Section*. Lorsqu'on ferme une section, les variables locales qui y sont définies sont abstraites, ce qui empêche de rouvrir simplement une section comme cela peut être fait en HOL.

6.6 Dépendances entre théories et autres outils de visualisation

– **Dépendances entre théories**

Les deux graphes visualisés ci-après, ont été générés automatiquement à partir d'un fichier de dépendance au format *Make*, en utilisant l'outil `dependtohml` [110] que nous avons développé conjointement avec Loïc Pottier.
Le résultat est un fichier *html* pouvant être visualisé dans un navigateur Web et contenant une image au format *gif* sensible à la souris. Les liens correspondant aux nœuds terminaux peuvent permettre d'accéder aux fichiers correspondants ou visualiser sur un nouveau graphe les dépendances entre les différents objets de ce fichier.
Nous donnons ici deux exemples de visualisation de l'organisation d'un gros développement. **Attention**, la notation $A \rightarrow B$ exprime ici que A dépend de B.
La figure 6.3 montre la structure d' un développement dû à Laurent Théry [136] conduisant à une preuve de l'algorithme de Buchberger pour le calcul de bases de Gröbner.

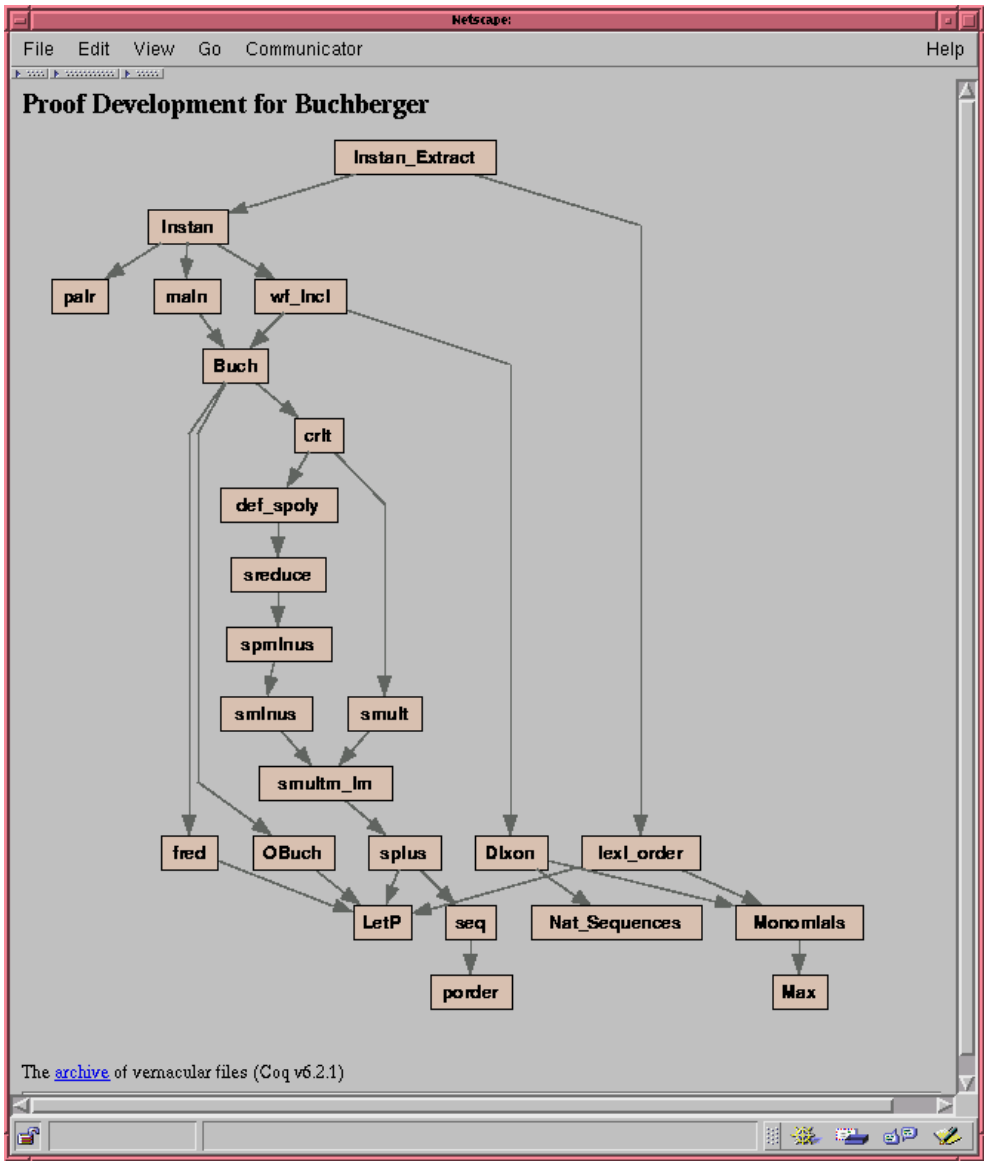


FIG. 6.3 – *Dépendances du développement de preuve de l'algorithme de Buchberger*

Sur la figure 6.4 on visualise les relations entre les différents fichiers d'une preuve de compilateur faites par Yves Bertot [14].

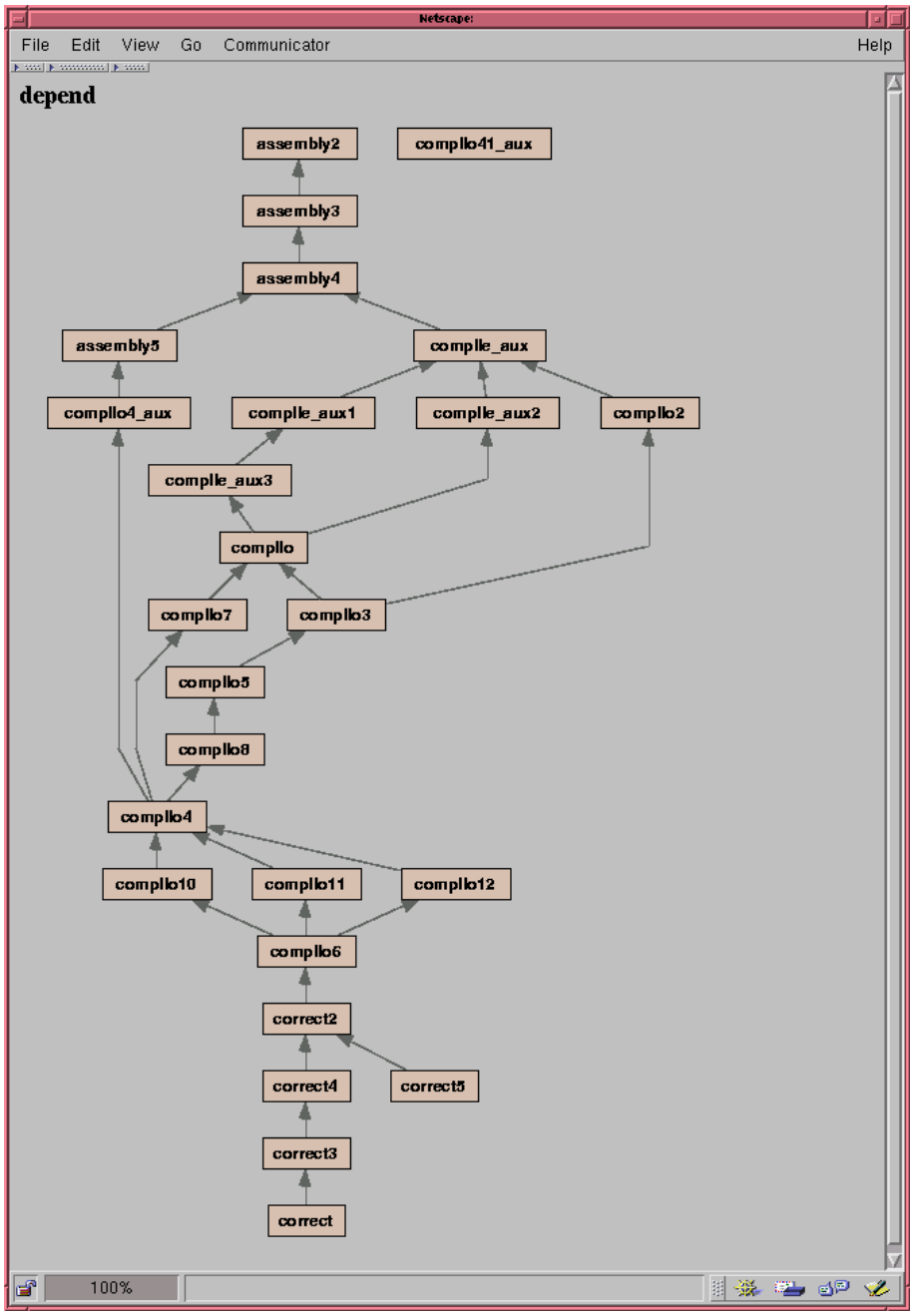


FIG. 6.4 – *Dépendance du développements de preuve d'un mini compilateur*

– **Visualisation d'une hiérarchie de type**
Enfin, nous mentionnons un dernier outil de visualisation qui permet de tracer les graphes de "coercion"[4] . En effet en mathématique, on a naturellement une hiérarchie des théories, ainsi un anneau est aussi un groupe qui est lui même aussi un monoïde etc. Cela pose le problème du sous-typage. Dans le système Coq, on peut

4. le terme français serait coercition mais n'est jamais utilisé. On lui préfère l'anglicisme coercion

simuler le sous-typage via les "coercions" implicites [118, 119]. Il nous a paru utile de permettre de visualiser graphiquement la hiérarchie de type ainsi définie. L'exemple de la figure 6.5 montre le cas d'un développement sur les anneaux dû à Loïc Pottier [109].

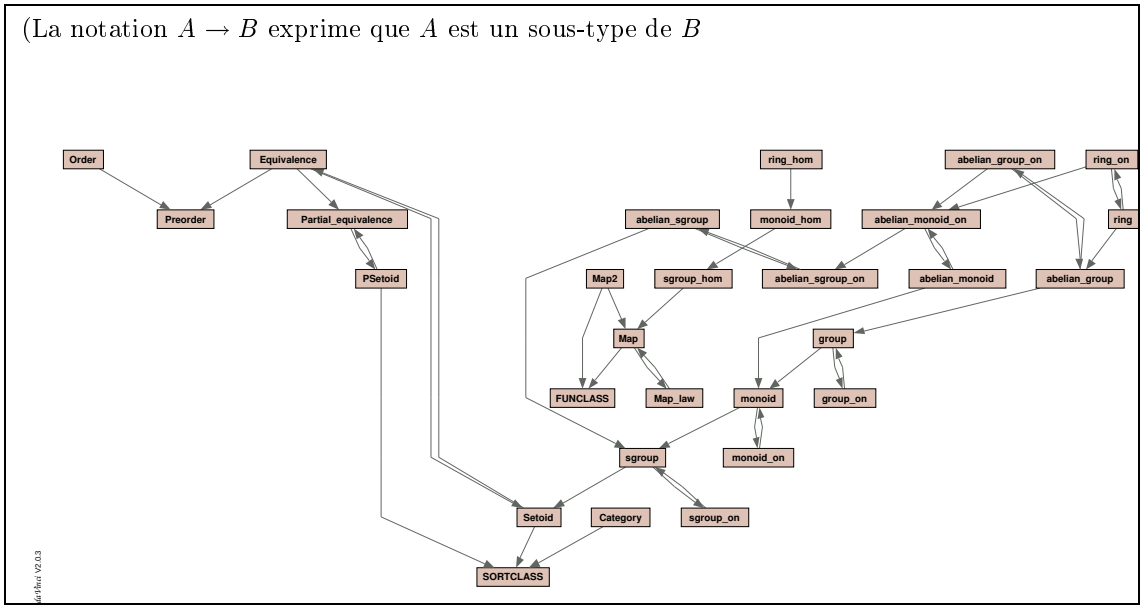


FIG. 6.5 – *Le graphe de coercion de la théorie des anneaux*

Chapitre 7

La coupe de théorie

Introduction

La coupe de théorie consiste, étant donnés une théorie et un théorème, à supprimer de la théorie tout ce qui n'est pas nécessaire à la preuve de ce théorème. Cela correspond donc au calcul du contexte minimal dans lequel un théorème t reste valide. Ce chapitre décrit l'implémentation de la coupe de théorie et les problèmes que la mise en œuvre pratique a permis de mettre en lumière.

7.1 Implémentation

Nous utilisons le graphe de dépendance d'une théorie sous forme de liste d'adjacence. Chaque nœud a pour étiquette un identificateur et est annoté par le nom du fichier source et des éventuelles sections où il est introduit.

7.1.1 Mécanisme de coupe de théorie

Soit T une théorie et t un théorème.

- Nous calculons la fermeture transitive du sous-graphe issu de t. (On évite d'introduire des répétitions en mémorisant les identificateurs lors de leur première introduction grâce à une table de hachage).
- Nous récupérons la liste des fichiers sources où apparaissent ces identificateurs.
- Il nous reste alors à parcourir ces fichiers et à y supprimer tous les objets qui ne sont pas associés à un des identificateurs de la fermeture transitive du sous-graphe issu de t.

Pour cela, nous avons écrit en utilisant Caml-lex un utilitaire qui analyse les fichiers sources et supprime tous ces objets. En Coq, Il faut aussi supprimer les références éventuelles à ces objets dans les commandes `Hint` ainsi que les références à des fichiers qui n'apparaissent pas dans la liste de fichiers sources dans des commandes comme `Require` ou `Load`.

7.2 Insuffisance des dépendances calculées sur l'objet preuve.

Dans nos premières expériences avec les coupes de théories, il est apparu que les scripts des théories minimales que nous avions calculés ne pouvaient pas forcément être rejoués sans problème. Certains objets manquaient. Dés lors une question s'imposait:

Les informations dans le terme de preuve forment-elles réellement un sur-ensemble de celles du script ?

Nous allons voir que la réponse est négative et que cela est lié à l'élimination des coupures pour reprendre le langage du calcul des séquents ou à la normalisation du λ-terme si l'on préfère la terminologie du λ-calcul.

On peut constater que lorsque le script contient ce que nous appellerons des tactiques de coupure, comme `Cut` (ou des tactiques composées à partir de `Cut` comme `LApply` ou `Generalize`), le λ-terme stocké par le système ne contient pas forcément tous les identificateurs qui apparaissent dans le script de preuve, comme le montre l'exemple suivant:
Exemple 7.1

On définit un lemme trivial sur les entiers naturels

```
Lemma trivial_lemma_1 :(x:nat)(x=O)->(pred (S x))=O.
Auto.
Save.
```

Puis on va prouver la tautologie suivante en faisant explicitement apparaître le lemme précédent (`trivial_lemma_1`*)*[1] *dans le script bien que cela soit complètement inutile.*

```
Lemma toto :(A:Prop)A->A.
LApply (trivial_lemma_1 O).
Auto.
Auto.
```

Si on observe le terme de preuve avant de sauver la preuve, il n'est pas réduit et `trivial_lemma_1` *apparaît:*

```
toto < Show Proof.
LOC:
```

1. `LApply (trivial_lemma_1 O).` produit deux sous-buts `(pred (S O))=O->(A:Prop)A->A` et `O=O`

```
Subgoals
Proof:
  ([_:(pred (S O))=O]
     [A:Prop][H0:A]H0 (trivial-lemma-1 O (refl_equal nat O)))
```

mais après que la preuve ait été sauvegardée, le λ-terme est réduit et la constante `trivial_lemma_1` *n'y apparaît plus*

```
Coq < Print toto.
toto = [A:Prop][H0:A]H0
     : (A:Prop)A->A
```

Le système garde en effet le terme de preuve sous sa forme normale, et la réduction sous forme normale, qui a lieu lorsque la preuve est sauvegardée, peut faire disparaître des identificateurs du λ-terme, comme dans l'exemple précédent.

Le fait qu'un identificateur n'apparaisse pas dans un terme de preuve alors qu'il apparaît dans le script signifie qu'il existe une démonstration qui n'utilise pas cet indentificateur et donc que le script peut être réarrangé pour correspondre à cette démonstration. Cela revient à éliminer les coupures inutiles. Nous discutons à la section 7.3 des possibilités de réarrangement automatique du script.

Le calcul des dépendances que nous avons fait sur l'objet preuve tel qu'il est sauvegardé, c'est-à-dire sous une forme normalisée, s'avère donc insuffisant pour définir des outils travaillant sur les scripts comme la coupe de théorie.

Au vu de l'exemple précédent deux solutions peuvent être envisagées:

1. calculer les dépendances sur les termes non normalisés
2. enrichir le calcul de dépendance fait sur l'objet preuve avec les dépendances calculées sur le script.

Nous montrons maintenant que la seconde n'est pas satisfaisante puis nous présentons les contraintes introduites par la première.

7.2.1 Une première solution...inadéquate

Sur l'exemple 7.1 (`trivial_lemma_1`), l'ensemble des dépendances de `toto`, calculées sur l'objet preuve, est vide (puisque le terme de preuve est clos) et le calcul fait sur le script donne `trivial_lemma_1`.

On pourra effectivement rejouer avec succès le script du lemme `toto` dès que `trivial_lemma_1` sera présent dans l'environnement. Il semble donc que ce calcul de dépendance soit satisfaisant pour servir de base à des outils de manipulation de script. Ce n'est pourtant pas vrai.

On peut en effet, à cause des procédures de décision automatique, rencontrer des cas où un identificateur n'apparaît ni dans le script ni dans le terme normalisé mais est néanmoins utilisé dans la preuve. L'exemple 7.2 montre une telle situation.

Exemple 7.2

```
Lemma titi :(x,y,z:nat)(plus x y)=(plus x z)->y=z.
Induction x;Simpl;Auto.
Save.
Hint titi.

Lemma toto :(A:Prop)A->A.
Cut (x,y,z:nat)(plus x y)=(plus x z)->y=z.
Auto.
EAuto.
```

L'identificateur `titi` *n'apparaît pas dans le script. Il apparaît bien dans le terme non normalisé. Il a été utilisé par une procédure de décision automatique.*

```
toto < Show Proof.
LOC:
Subgoals
Proof: ([_:(x,y,z:nat)(plus x y)=(plus x z)->y=z][A:Prop][H0:A]H0
         [x,y,z:nat][H:(plus x y)=(plus x z)](titi ?539 y z H))
```

On peut aussi noter que le terme de preuve contient encore une variable existentielle `?539` *qui devrait être instanciée si nous voulions stocker le terme sous sa forme non normalisée. La réduction fait "disparaître" la variable existentielle, le terme peut donc être sauvegardé sous sa forme réduite. De plus après réduction, l'identificateur* `titi` *n'apparaît plus dans le terme.*

```
Save.
Coq < Print toto.
toto = [A:Prop][H0:A]H0
     : (A:Prop)A->A
```

Ainsi en utilisant seulement le script et le terme normalisé, on ne décèle pas que `toto` dépend de `titi`, pourtant si ce dernier n'est pas présent dans l'environnement le script du premier ne pourra pas être rejoué avec succès.

Ici, c'est donc l'utilisation conjointe des tactiques de coupure et de décision automatique qui masque certaines dépendances. C'est pourtant en liaison avec les procédures de décision automatique, qu'il est intéressant d'introduire des résultats par coupures sans être sûr qu'ils soient utiles pour faire la preuve. L'idée est d'enrichir suffisamment le contexte pour que la procédure de décision puisse prouver le résultat courant.

7.2.2 Calcul sur un terme non normalisé

On opte donc pour un calcul sur les termes non normalisés qui peut servir de base à une implémentation sur de la coupe de théorie[2]. Cependant la mise en œuvre de ce calcul introduit de nouvelles contraintes. La normalisation du terme de preuve à lieu au moment où l'on sauvegarde la preuve. Ainsi en mode interactif, il suffit de calculer les dépendances avant de sauvegarder la preuve.

Lorsqu'un fichier est compilé, ce sont les termes de preuve normalisés qui sont stockés. On ne peut donc plus utiliser les fichiers compilés pour calculer les dépendances.

Nous pourrions donc envisager de sauvegarder les théorèmes sous leur forme non normalisée mais cela n'est pas toujours possible à cause des variables existentielles. Ainsi dans l'exemple 7.2 le terme de preuve de `toto` ne peut pas être stocké sans être normalisé car il contient une variable existentielle qui n'est pas instanciée. Il faut donc au moment de la compilation générer un fichier séparé contenant le graphe de dépendance des théorèmes du fichier.

7.3 Réorganisation du script de preuve et suppression de code mort

On peut aussi conserver le calcul de dépendance initial (sur le terme normalisé) et essayer de réarranger le script pour y supprimer le code mort.

Commençons par le cas simple: celui où un identificateur apparaît dans le script mais pas dans le terme normalisé. Le principe est le suivant:

1. Détecter qu'un identificateur apparaissant dans le script n'est pas nécessaire à la preuve en comparant l'ensemble des identificateurs qui apparaissent dans le script avec ceux qui apparaissent dans le terme normalisé.
   Soit p_i les chemins correspondants aux tactiques contenant un tel identificateur.
2. Retrouver la tactique de coupure qui a introduit le résultat prouvé par cet identificateur.
3. Effacer la branche issue de cette tactique en utilisant le mécanisme de retour-arrière logique du chapitre 3 éventuellement après avoir expansé le script de preuve à l'aide des outils du chapitre 4.

Seul le second point est un peu délicat. On recherche en remontant dans l'arbre de preuve, la première tactique de coupure t de chemin p telle que:

$$\forall i, \exists p'_i \; p_i = p.2.p'_i$$

2. Il reste néanmoins de très rares cas pathologiques liés aux procédures de décision automatique et que nous discutons au chapitre 9.

C'est-à-dire telle que tous les p_i appartiennent à la seconde branche issue de t.

Le cas où les résultats inutiles introduits par une coupure sont prouvés par une procédure de décision automatique est un peu plus compliqué.

Adaptons le principe précédent:

1. Trouver les identificateurs qui disparaissent lors de la normalisation.
2. Trouver toutes les procédures de décision automatique utilisant un tel identificateur. Soit a_i ces procédures et p_i les chemins correspondants.
3. Retrouver la tactique de coupure qui a introduit le résultat prouvé par a_i.
4. Effacer la branche issue de cette tactique en utilisant le mécanisme de retour-arrière logique du chapitre 3 éventuellement après avoir expansé le script de preuve à l'aide des outils du chapitre 4.

Cet algorithme a donc une étape supplémentaire pour laquelle il faut être capable de savoir ce qu'a fait une procédure de décision. Il faut donc pouvoir ouvrir la boîte noire. Cela n'est pas toujours évident. Dans le système Coq, cela peut se faire en utilisant la structure d'arbre de preuve maintenue par le système, ou plus simplement en utilisant le combinateur `Info`. Dans ce dernier cas il suffit lors de l'envoie d'une ligne de script de de l'interface vers le système de preuve d'ajouter automatiquement le mot clef `Info`. Il faut ensuite recéper du côté de l'interface l'information fournit par `Info` du cote du prouveur.

Chapitre 8

Une méthode interactive de propagation des modifications

8.1 Graphe de dépendance et ordre des modifications

L'objectif de ce chapitre est de présenter une méthode utilisant le graphe de dépendance pour aider l'utilisateur à modifier une théorie. On suppose donc que l'on dispose du graphe de dépendance des objets d'une théorie T, et que l'on désire modifier certains objets de cette théorie. A chaque objet, on associe une date de modification par la fonction:

$$date : \mathcal{ED} \rightarrow integer$$

Cela nous permet d'introduire une notion de cohérence. Une théorie T est dite **cohérente** si:

$$\forall d_1,d_2 \in T,\ d_1 \prec_{dep} d_2 \ \Rightarrow\ date(d_2) \ \leq\ date(d_1)$$

Un ou plusieurs objets ayant été modifiés, le problème est d'abord de savoir quels autres objets de la théorie doivent être modifiés pour rétablir la cohérence et surtout comment propager ces modifications, c'est-à-dire dans quel ordre les réaliser pour minimiser le travail de remise en cohérence. Le point clef de notre démarche est d'éviter de vérifier et éventuellement de modifier plus d'une fois le même objet. C'est-à-dire qu'à chaque étape on ne veut vérifier que des nœuds qui ne dépendent pas eux mêmes de nœuds devant être vérifiés.

Pour savoir quels objets doivent être vérifiés ou modifiés, on va considérer les sous-graphes issus des objets modifiés. La relation d'ordre partielle sur ces graphes est la relation $\prec_{dep}$ définie au chapitre 6.

On veut trouver un ordre total $<$, compatible avec cet ordre partiel. Autrement dit on veut exhiber une liste $(S_1 \ldots S_n)$ croissante pour la relation $<$, de tous les sommets du graphe telle que si l'on a $S_j \prec_{dep} S_i$ alors $j < i$. La fonction $rang$ que nous avons définie

au chapitre 6.2 donne évidement un tel ordre. C'est l'ordre dans lequel les objets ont été initialement introduits mais il n'est pas unique. Sur le graphe de la figure 8.1, on peut

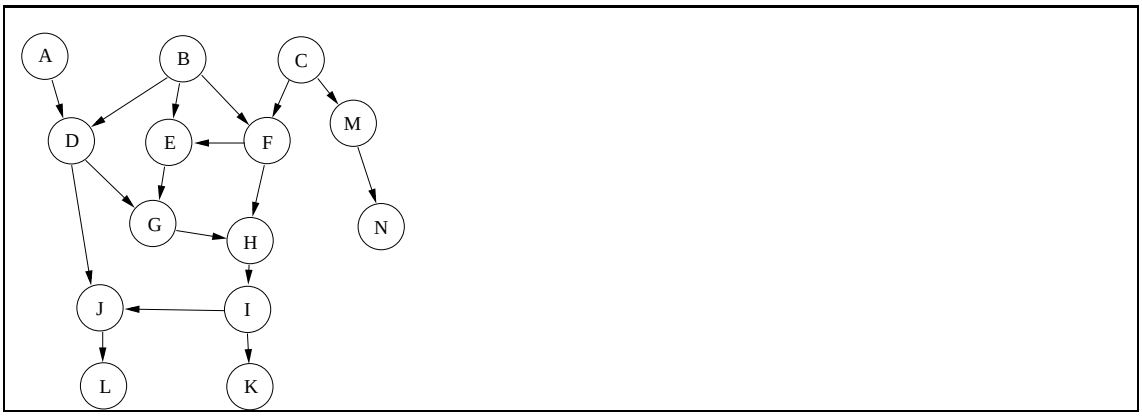


FIG. 8.1 – *Le graphe de départ*

obtenir [C; M; N; B; F; E; A; D; G; H; I; J; L; K] mais [A; B; C; D; F; E; G; H; I; J; K; L; M; N] convient tout autant.

L'utilisateur peut vouloir choisir son ordre de modification pour commencer par exemple par faire les "changements mineurs" ou au contraire vérifier en priorité que les preuves les plus "difficiles" sont encore faisables dans le nouveau contexte. De manière générale, trouver un tel ordre s'appelle faire un tri topologique.

Les algorithmes de tri topologique permettent de fournir un ordre pour propager les modifications mais ne permettent pas une bonne interactivité avec l'utilisateur à chaque étape de modification. De plus dans le cas des dépendances entre théorèmes, la propagation des modifications peut être minimisée et cette optimisation est difficile à mettre en place avec les algorithmes classiques de tri topologique. On va donc fournir un algorithme qui permet à chaque étape de modification de laisser l'utilisateur choisir le nœud qu'il veut modifier à l'étape suivante et de connaître l'état exact de la propagation des modifications.

8.2 Méthode générale

On dispose d'une théorie dans laquelle certains objets ont été modifiés et du graphe de dépendance obtenu par un calcul de dépendance sur la théorie avant modification.

On introduit la notion de **degré d'incohérence** d'une théorie par la fonction:

$$d^o_{inc} : \mathcal{TH} \rightarrow integer$$

définie par:

$$d^o_{inc}(th) = card(\{d \in th,\ \exists d_1 \in th, d_1 \prec_{dep} d\ \wedge\ date(d)\ \leq\ date(d_1)\})$$

Si une théorie th est cohérente on a évidemment $d^o_{inc}(th) = 0$.

8.2.1 Principe de propagation des modifications

Les nœuds du graphe de dépendance sont annotés par un statut qui peut prendre 4 valeurs:

- `NORMAL`,
- `MODIFIE`,
- `DOUTEUX`,
- `A_MODIFIER`

On dira que n_1 dépend directement de n si $n\mathcal{R}_{dep}n_1$ et simplement que n_1 dépend de n si $n \prec_{dep} n_1$

Le statut `NORMAL` représente les nœuds en l'absence de modification, le statut `MODIFIE` les nœuds qui ont été modifiés, le statut `DOUTEUX` les nœuds qui sont douteux , c'est-à-dire qui dépendent d'un nœud modifié, enfin les statuts `A_MODIFIER` représentent les nœuds qui dépendent directement d'au moins un nœud modifié et n'ayant pas de parent douteux. Les nœuds ayant le statut `A_MODIFIER` sont ceux qui peuvent être vérifiés à l'étape courante.

Au départ tous les nœuds sont annotés par `NORMAL` et le système est cohérent. A chaque fois qu'un nœud prend le statut `MODIFIE` sa date est incrémentée. L'algorithme est le suivant:

P-1: Les nœuds modifiés sont annotés à MODIFIE et leur date est incrémentée. Cela introduit un degré d'incohérence n.

P-2: Tous leurs descendants dans le graphe sont annotés à `DOUTEUX`, les autres restant `NORMAL`.

Les opérations suivantes vont contribuer à réduire le degré d'incohérence introduit.

E-1: Parmi les nœuds `DOUTEUX`, ceux qui n'ont pas de parents `DOUTEUX` prennent le statut `A_MODIFIER`.

E-2: L'utilisateur choisit un des nœuds ayant le statut `A_MODIFIER`.

E-3: Après vérification et modification éventuelle, le nœud choisi à l'étape précédente est annoté à `MODIFIE` et sa date est incrémentée, tandis que, dans ses descendants directs, ceux qui n'ont pas de parent `DOUTEUX`, sont annotés à `A_MODIFIER` et pourront donc être attaqués à l'étape suivante.

E-4: Tant qu'il reste des nœuds annotés par `A_MODIFIER`, qui peuvent être attaqués, on reprend à l'étape E-2

T-1: Quand la propagation est terminée, tous les nœuds ayant le statut `MODIFIE` sont remis à `NORMAL` et toutes les dates peuvent être remises à zéro.

Le système gère l'ordre de vérification en orientant le choix de l'utilisateur, de telle sorte qu'à chaque étape le degré d'incohérence ne puisse que diminuer, mais c'est l'utilisateur qui choisit à chaque étape parmi les nœuds proposés.

8.2.2 Exemple

Au départ figure 8.1 tous les nœuds ont un statut `NORMAL` (en blanc).

Après modification des nœuds `A` et `B`, qui sont incomparables pour l'ordre partiel sur le graphe, ils prennent le statut statut `MODIFIE` (en gris cerclé de noir). Tous leurs descendants deviennent `DOUTEUX` (en gris), puis parmi leurs descendants directs, ceux qui n'ont pas de parent `DOUTEUX` prennent le statut `A_MODIFIER` (en blanc cerclé de noir) et peuvent être modifiés à leur tour pour maintenir la cohérence. L'utilisateur doit alors décider s'il poursuit

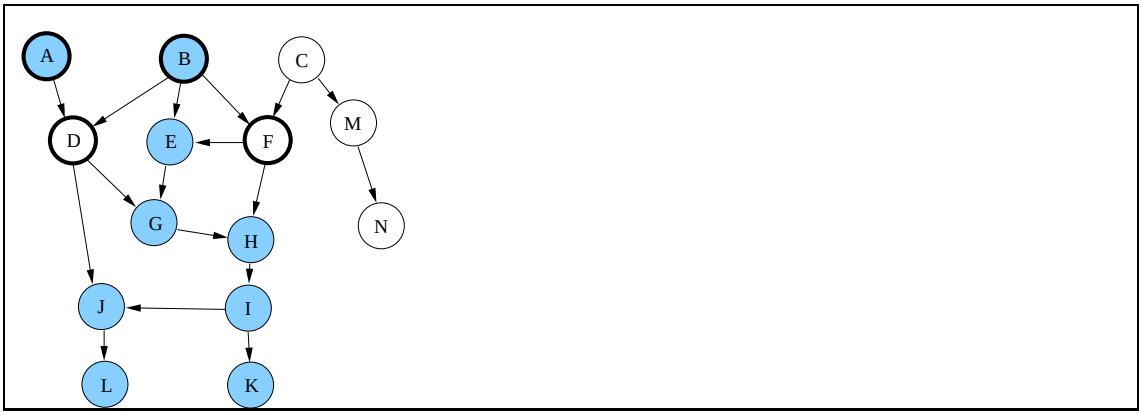


FIG. 8.2 – *Début de traitement après modification de A et B*

en modifiant le nœud `D` ou le nœud `F`. Ici il choisit alors de modifier le nœud `D`, qui prend donc le statut `MODIFIE`. Ses descendants qui n'ont plus de parents `DOUTEUX` deviennent `A_MODIFIER` et pourront donc être attaqués à l'étape suivante. Ici tous les descendants de `D` ont des parents `DOUTEUX` qui doivent donc être modifiés avant eux, ils ne changent donc pas de statut.

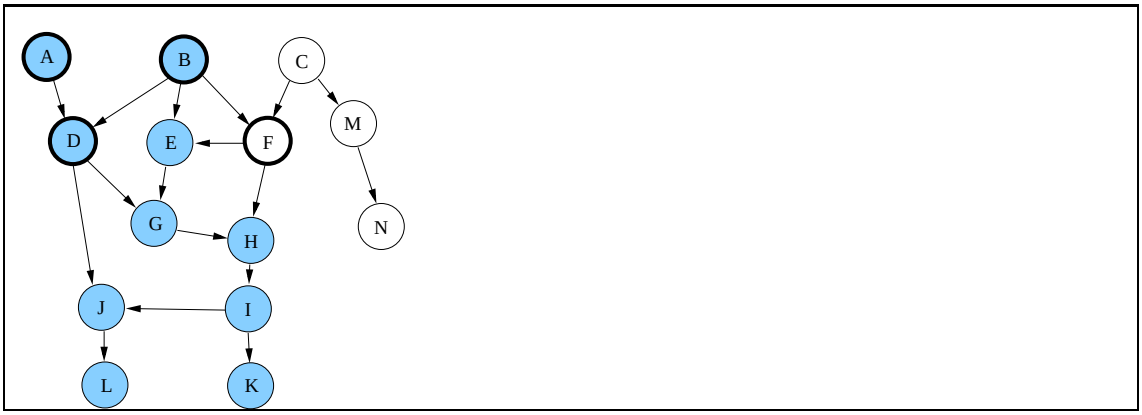


FIG. 8.3 – *Après avoir propagé la modification au nœud D*

Le seul nœud modifiable (`A_MODIFIER`) reste `F`, on le modifie donc à son tour. Il prend le statut `MODIFIE` tandis que le seul de ses fils qui n'a pas de parents `DOUTEUX`, (`E`) devient `A_MODIFIER`.

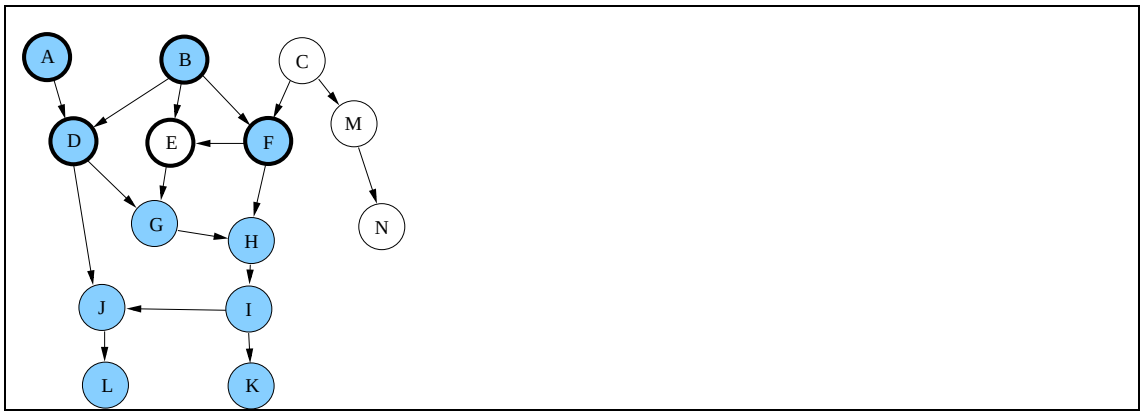


FIG. 8.4 – *Après avoir propagé la modification au nœud F*

On attaque ensuite le nœud `E` qui devient `MODIFIE` tandis que `G` qui n'a plus de parents `DOUTEUX` devient `A_MODIFIER`.

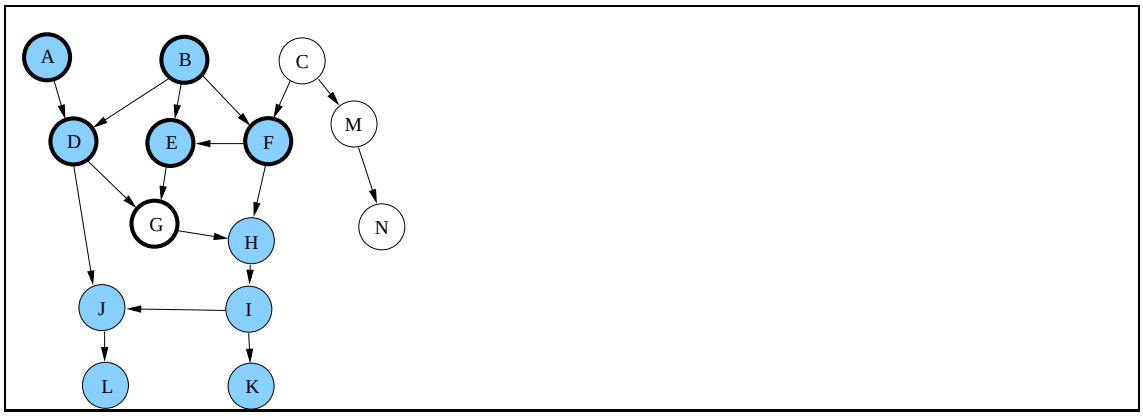


FIG. 8.5 – *Après avoir propagé la modification au nœud E*

On continue en attaquant `G` qui devient `MODIFIE` tandis que son fils `H` qui n'a plus de parents `DOUTEUX` devient `A_MODIFIER` et ainsi de suite.

Lorsque l'utilisateur choisit un nœud il peut le traiter à la main, c'est-à-dire, le rejouer pas à pas et le modifier si nécessaire. Le système peut aussi rejouer automatiquement le résultat sélectionné. Si cela réussit il passe à l'étape suivante sinon il redonne la main à l'utilisateur.

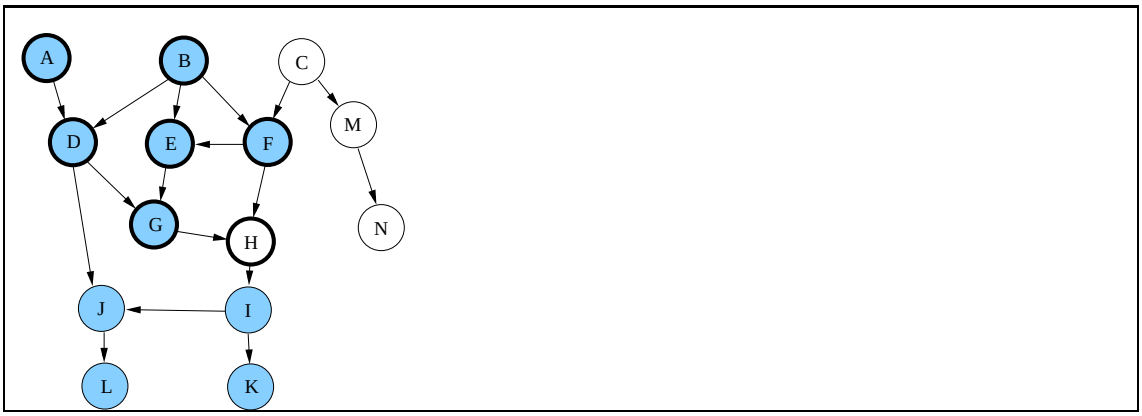


FIG. 8.6 – *Après avoir propagé la modification au nœud G*

8.2.3 Propriétés

8.2.4 Remarques

La propagation des modifications est gérée par les règles E-1 à E-4 qui, par definition, ne s'appliquent que sur des nœuds ayant les statuts `DOUTEUX` ou `A_MODIFIER`.

C'est à l'etape E-3 qu'ont lieu les modifications et l'incrémentation de la date; c'est donc la seul étape qui modifie le degrés d'incohérence de la théorie considérée.

8.2.5 Propriétés

1^o: Seuls les nœuds qui dépendent effectivement des nœuds de départ sont considérés;

2^o: Aucun nœud n'est modifié plus d'une fois;

3^o: Chaque pas de propagation des modifications (à l'etape E-3) diminue le degré d'incohérence.

8.2.6 Preuves

1^o: Si un nœud n ne dépend pas d'un nœud modifié en P-1, il ne prend pas le statut `DOUTEUX` en P-2 mais garde le statut `NORMAL`. Dès lors, il n'interfère plus avec les étapes E-i ou qui ne considèrent que des nœuds ayant les statut `DOUTEUX` et `A_MODIFIER`.

2^o: Lorsqu' un nœud est modifié il prend le statut `MODIFIE`. Il n'existe pas d'étape permettant de changer ce statut. La seule étape qui permet de modifier un nœud après les modifications initiales est E-3 mais elle ne s'applique pas à un nœud ayant le statut `MODIFIE`. Ainsi, une fois qu'il a pris le statut `MODIFIE`, un nœud ne peut pas être à nouveau modifié.

3^o: Soit Th une théorie on note S, l'ensemble $\{m|\exists l, l \prec_{dep} m \wedge date(l) > date(m)\}$. Au départ la théorie est cohérente. A l'étape P-1 on modifie les nœuds $n_1, \ldots n_k$.

Les dates des n_i ont donc été incrémentées. A l'étape P-2, les nœuds de l'ensemble $D = \{m | \exists i \in \{1..k\},\ n_i \prec_{dep} m\}$ descendants des noeuds modifiés en P-1 prennent le statut `DOUTEUX`
On note S, l'ensemble $\{m, \exists l, l \prec_{dep} m \wedge date(l) > date(m)\}$ dont le cardinal donne le degré d'incohérence. Avant de commencer la propagation S est égal à D.
Au passage par l'étape E-1, l'ensemble S est partitionné en deux ensembles $S1$, $S2$. Les nœuds de $S1$ ont le statut `DOUTEUX`, ceux de `S2` le statut `A_MODIFIER`. Lors des passages ultérieurs par l'étape E-3, certains nœuds de $S1$ pourront changer de statut pour passer dans $S2$ mais, par définition on a toujours les propriétés

$$\forall x \in S2\ \nexists n \in S1,\ n \mathcal{R}_{dep} x \quad (a)$$

et

$$\forall x \in S2\ \nexists n \in S2,\ n \mathcal{R}_{dep} x \quad (b)$$

A chaque passage par l'étape E-3, l'utilisateur modifie un nœud k choisi dans $S2$ à l'étape E-2. Ce nœud prend le statut `MODIFIE` et est donc supprimé de $S2$ et sa date est incrémenté. Les propriétés (a) et (b) permettent de conclure que $\forall l,\ l \prec_{dep} k \Rightarrow date(n) > date(k)$, ce qui signifie que k n'est plus dans S et par conséquent $d^o_{incn}(th) = d^o_{incn-1}(th) - 1$ (où l'indice `n` est le nombre de passage par E-3).

8.2.7 Stockage en cours de modification

La maintenance d'un document et la propagation des modifications est une activité de longue haleine. Il faut prévoir de pouvoir stocker les états courant de propagation lorsque l'on interrompt la tâche de reprise. Le plus simple est de stocker une représentation du graphe avec les nœuds marqués et les indications de date, ce qui permet de reprendre à l'étape E-3 lorsque l'on reprend le traitement.

8.3 Spécificités des dépendances entre théorèmes

On appellera dépendances logiques les dépendances exprimées par la relation $\mathcal{R}_{dep}$ définie au chapitre 6. Les dépendances exprimées par la relation $<_{hist}$ seront quant à elles appelées dépendances historiques. Dans un ensemble d'objets ordonné par $<_{hist}$, pour former un environnement valide, les dépendances logiques sont un "affinement" de la notion de dépendance historique.

Au paragraphe 8.3.1, nous montrons comment la relation $\mathcal{R}_{dep}$ peut elle même être affinée dans le but d'optimiser l'algorithme du paragraphe 8.2.1, puis au paragraphe 8.3.2 nous présentons l'algorithme optimisé.

8.3.1 Dépendances opaques et dépendances transparentes

Généralement en mathématiques, les théorèmes dépendent les uns des autres par leurs énoncés et non par leurs preuves. On parle (en utilisant un anglicisme) de "proof-irrelevance".

Dans un système basé sur la théorie des types, l'isomorphisme de Curry-Howard permet de dire que ces objets dépendent les uns des autre par leurs types et non par leurs termes. Par contre dès que l'on parle d'objets qui ont un comportement calculatoire comme les fonctions, le terme, qui est la description de ce comportement, devient souvent fondamental. Ainsi les fonctions d'addition et de multiplication sur les entiers ont le même type `nat->nat->nat` mais leurs termes sont bien évidement distincts. De même lorsqu'un objet est défini inductivement, c'est le corps de la définition inductive qui est important et non le type de l'objet, ainsi `nat` et `list` sont tous deux de type `Set` mais sont complètement différents (ils ont des constructeurs dont les noms et les types diffèrent).

D'une manière générale, nous pouvons distinguer deux types de dépendances logiques:

1° **Les dépendances opaques**
Les dépendances opaques sont celles qui ne prennent en compte que la spécification (dans Coq cela est donné par le type) des objets. Un terme `T1` dépend de manière opaque d' un terme `T2` s' il l'utilise mais sans accéder à la structure du terme.
Du point de vue de la maintenance, cela entraîne que si l' on modifie `T2` sans changer son type, `T1` ne sera pas affecté.

2° **Les dépendances transparentes**
Les dépendances transparentes prennent en compte la valeur des objets (qui dans Coq correspond au terme). Un terme `T1` dépend de manière transparente d' un terme `T2` s'il utilise la structure de ce terme.
Du point de vue de la maintenance cela signifie que toute modification d'un terme `T2` utilisé par `T1` risque d'affecter `T1` même si le type de `T2` n'a pas changé.

Mais il n'est pas toujours évident de déterminer la nature d'une dépendance. Illustrons cela par un exemple:

Exemple 8.1

On définit les fonctions:

```
Fixpoint addition [n:nat] : nat -> nat :=
   [m:nat]Cases n of
      O   => m
  | (S p) => (S (addition p m)) end.

Fixpoint  multiplication [n:nat] : nat -> nat :=
   [m:nat]Cases n of O => O
               | (S p) => (addition m (multiplication p m)) end.
```

`multiplication` *est définie en fonction d'*`addition` *mais la dépendance est-elle opaque ou transparente? Dans les deux cas le terme associé à* `multiplication` *est:*

```
Fix multiplication
  {multiplication  [n:nat] : nat->nat :=
```

```
[m:nat]
 Cases n of
   O => O
 | (S p) => (addition m (multiplication p m))
 end}
: nat->nat->nat
```

Si la dépendance est transparente, `multiplication` *peut utiliser le terme associé à* `addition` *dans ses réductions, on pourra donc prouver un lemme comme le suivant:*

```
Lemma multiplication_n_O : (n:nat)(O=(multiplication n O)).
Proof.
 Induction n; Simpl; Auto.
Qed.
```

qui dépendra de façon transparente de `multiplication`. *Si* `multiplication` *dépend de façon opaque d'*`addition`*, on ne pourra pas le réduire puisqu'on ne peut réduire* `addition`*(la tactique* `Simpl` *ne fera rien) et on ne pourra donc pas prouver le lemme* `multiplication_n_O`.

Mais on peut toujours prouver des résultats qui dépendront de façon opaque de multiplication comme le lemme trivial suivant:

```
Lemma ex_nat_nat_nat:(Ex [f:nat->nat->nat] True)
Apply (ex_intro nat->nat->nat [x:(nat->nat->nat)]True multiplication).
 Exact I.
Save.
```

Une approximation de la nature des dépendances logiques

En pratique, à tout moment chaque objet de l'environnement a un statut opaque ou transparent. Si un objet est opaque, on ne peut accéder à son terme et donc tous les objets qui l'utilisent en dépendent forcement de manière opaque. Si un objet est transparent on peut accéder à son type comme à son terme. On ne peut alors savoir a priori si les objets qui l'utilisent, utilisent ou non son terme. On ne peut donc savoir avec certitude la nature des dépendances. Mais une approximation sans risque est de considérer qu'alors la dépendance est transparente. Cela est loin d'être optimal mais permet néanmoins d'optimiser le processus de propagation des modifications que nous abordons à la section suivante.

En utilisant le script de preuve, on peut arriver à une meilleure approximation de la nature des dépendances mais cela sera forcément plus coûteux. Deux options sont possibles.

Une solution consiste à rendre le terme opaque et à rejouer le script. Chaque fois qu'une preuve échoue, on sait que la dépendance est transparente, on rend alors de nouveau le terme transparent afin de pouvoir rejouer la preuve puis de nouveau opaque pour passer à l'objet suivant dans le développement. Toutes les fois que le script repasse sans problème on sait que la dépendance est opaque. Dans l'exemple 8.1 si `addition` ou a fortiori

`multiplication` est opaque le script de preuve de `multiplication_n_O` échoue, ce théorème dépend donc de façon transparente de `multiplication` et à fortiori d'`addition`.

Remarque: Cette solution suppose que la statut d'un terme est fixé lors de sa définition. Le système Coq autorise cependant à changer la nature des termes à n'importe quel nomment du développement. Cette possibilité bien que dangereuse est parfois utile pour limiter la porté des tactiques de réduction. En effet l'application de la tactique `Simpl` essaye de réduire tous les termes transparents alors que l'on voudrait parfois ne réduire que certains de ces termes. Une solution est alors de rendre momentanément les autres opaque le temps de la réduction. Dans ce cas notre solution qui considère comme transparent un terme rendu opaque reste néanmoins sûr bien que non optimale. Ce qui pourrait invalider notre calcul serait que l'utilisateur soit amené à rendre transparent un terme précédemment définie comme opaque mais cette opération nous semble révéler une erreur de conception et donc devoir être bannie.

On notera respectivement $\mathcal{R}_{depOp}$ les dépendances opaques. La relation de dépendance transparente correspond a $\mathcal{R}_{dep}$ mais nous la noterons parfois $\mathcal{R}_{depTrans}$ pour spécifier que l'on est dans un contexte où on la différencie de $\mathcal{R}_{depOp}$.

Nous allons voir maintenant comment les dépendances opaques permettent de limiter la propagation des modifications et comment optimiser la méthode du paragraphe 8.2.1 pour tenir compte de cela.

8.3.2 Adaptation de la méthode de propagation des dépendances

Le principe reste le même que celui que nous avons décrit au § 8.2.1, seule l'étape E-3 est modifiée, nous la détaillons maintenant:

E-3: (a) Si le terme de preuve correspondant au nœud choisi à l'étape précédente est modifié tout se passe comme précédemment, le nœud est annoté à `MODIFIE` et sa date est incrémentée. Dans ses descendants directs ceux qui n'ont pas de parent `DOUTEUX`, sont annotés à `A_MODIFIER` et pourront donc être attaqués à l'étape suivante.

(b) Si la spécification (c'est-à-dire, l'énoncé ou encore le type) correspondant au nœud choisi à l'étape E-2 n'est pas modifiée, le nœud courant est annoté à `MODIFIE`, mais les nœuds qui ne dépendent transitivement que de lui et par une relation de dépendance opaque n'ont plus à être considérés.
Ils prennent donc le statut `NORMAL`[1] et leur dates sont incrémentées (dans un ordre compatible avec la relation $\prec_{dep}$).
Les nœuds qui dépendent (de façon opaque ou transparente) du nœud modifié, mais qui ont aussi des parents annotés à `DOUTEUX` ou à `A_MODIFIER`, gardent le statut `DOUTEUX`.

1. Ici, il est tout a fait possible d'annoter tout ces nœuds avec le statut `MODIFIE` mais le fait d'utiliser `NORMAL` permet de mettre en valeur le fait qu'il n'ont pas eu à être modifiés.

Enfin, parmi les autres nœuds qui dépendent du nœud modifié, ceux qui ont des parents à `MODIFIE`, ou ceux qui ne dépendent transitivement que de lui mais de façon transparente, prennent le statut `A_MODIFIER`.

Les propriétés de l'algorithme de la section précédente sont toutes conservées. Pour les propriétés 1 et 2 c'est évident. Pour la propriété 3, il suffit de remarquer que lorsque la règle E-3 (b) est appliquée, d'une part les nœuds qui prennent le statut `NORMAL` ne dépendent d'aucun nœud ayant le statut `DOUTEUX` ou `A_MODIFIER` qui pourrait être modifié par la suite, d'autre part que l'ordre dans lequel on incrémente les dates de ces nœuds est compatible avec la relation $\prec_{dep}$. Cela permet de conclure que lorsque l'application de la règle E-3 (b) fait passer k nœuds au statut `NORMAL` le degrés d'incohérence diminue de k.

Exemple 8.2

Reprenons le graphe de la figure 8.6, mais en supposant que `I` *dépende de* `H` *de façon opaque et que* `J` *dépende de* `I` *de façon opaque, toutes les autres dépendances étant transparentes. On attaque* `H` *mais il apparaît qu'il n'a pas à être modifié, il prend alors le statut* `NORMAL` *et sa date est incrémentée.* `I` *qui en dépend de façon opaque n'a pas a être vérifié, il prend le statut* `NORMAL` *et sa date est incrémentée. Mais* `K` *dépend de façon transparente de* `I` *et donc de* `H`*, mais aussi de* `F`*,* `G` *. . . et peut donc être affecté par leur modification. Il prend donc le statut* `A_MODIFIER` *et la propagation peut continuer sur cette branche.* `J` *dépend de façon opaque de* `I` *mais il dépend aussi de* `D` *qui a le statut* `MODIFIE` *donc il doit être vérifié et prend le statut* `A_MODIFIER` *et la propagation peut continuer sur cette branche.*

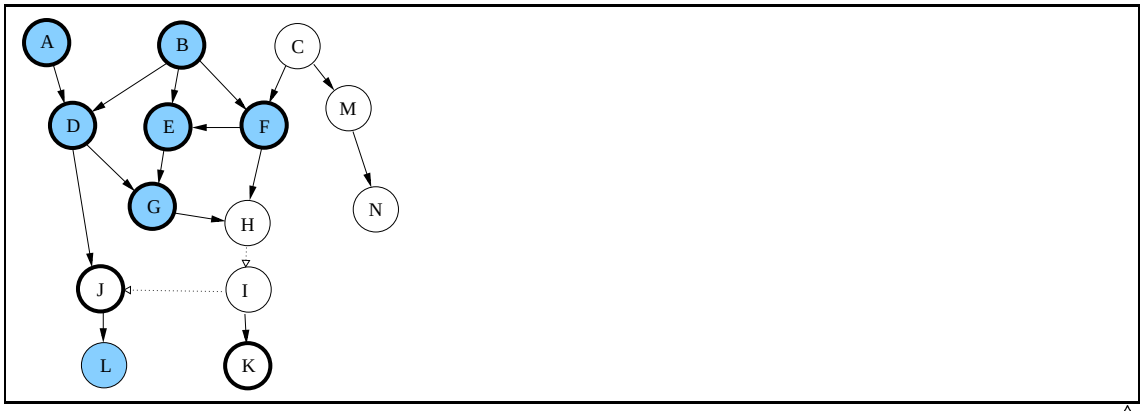


FIG. 8.7 – *Optimisation de la propagation des modifications*

Attention, le fait que le terme associé à un nœud n ne soit pas modifié ne suffit pas pour arrêter la propagation des modifications. La propagation ne s'arrête que vers les nœuds qui dépendent de n de façon opaque. L'exemple 8.1 montre que si on modifie `addition` (ou même si on le rend simplement opaque) même si `multiplication` n'est pas modifié, on ne pourra plus prouver le lemme `multiplication_n_O` mais on pourra toujours prouver `ex_nat_nat_nat`.

8.4 Mise en œuvre

On dispose maintenant d'un mécanisme permettant de décider incrémentalement d'un ordre de vérification et éventuellement de modification des objets d'une théorie. On veut fournir un outil pratique basé sur ce mécanisme, il faut donc pouvoir à chaque étape mettre le système dans un état où la vérification d'un résultat est possible. C'est-à-dire un état où tous les résultats (modifié ou non) nécessaires à sa preuve sont présents.

8.4.1 Résumé du principe d'utilisation

Disposant d'une théorie et de son graphe de dépendance, l'utilisateur modifie son axiomatique. Puis les étapes suivantes sont exécutées par l'interface:

1° Lancer l'algorithme de propagation pour calculer quels nœuds peuvent être modifiés.
2° Donner le choix à l'utilisateur (avec ou sans pré-visualisation des théorèmes concernés).
3° Positionner la sélection courante (ou le curseur) dans l'éditeur sur le théorème choisi et mettre le système de preuve dans un état où tous les résultats qui, d'après le graphe de dépendance calculé avant modification, sont nécessaires à la preuve sont présents dans l'environnement.
4° Attendre que l'utilisateur rejoue le théorème et éventuellement le modifie.
5° Reprendre à l'étape 1

Le point clef restant à étudier est l'étape 3. Il faut que tous les théorèmes dont dépend le résultat à vérifier aient déjà été joués. Cela est à première vue très simple, il suffit de rejouer, s'ils ne sont pas déjà présents dans l'environnement, les résultats nécessaires à la vérification du résultat étudié. Mais des problèmes peuvent survenir lorsque ces résultats se trouvent dans d'autres sections que celle du résultat à vérifier.

Rappelons que le mécanisme de "sectionnement" de Coq permet d'organiser une preuve en `sections` structurées. A l'intérieur d'une section, il est possible de faire des déclarations locales. Quand la section est fermée toutes ces déclarations sont déchargées, c'est-à-dire que les variables locales sont abstraites dans chacun des termes de la section où elles apparaissent. Un fichier est implicitement une section.

Supposons que notre développement soit composé des 3 fichiers de la figure 8.8 (où $x \rightarrow y$ indique que x dépend de y).

Nous modifions `Ob1`. On peut alors vérifier indifféremment `Oc2` ou `Ob2`. Si l'on choisit de vérifier `Oc2`, il faut que le système contienne `Ob1` et `Ob3`. Mais on ne peut pas simplement utiliser la commande `Require B` car cela chargerait l'ancienne version compilée du fichier `B` (c'est-à-dire la version du fichier `B` antérieure à la modification de `Ob1`). On ne peut pas,

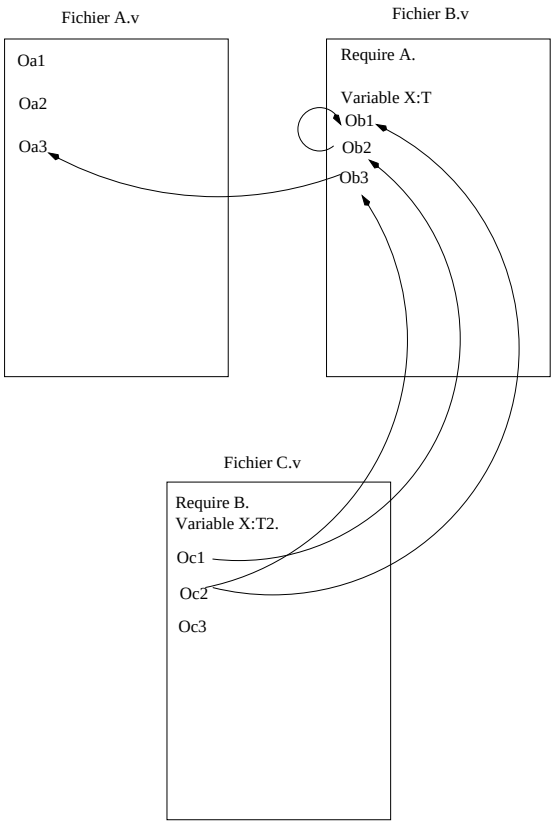


FIG. 8.8 – *Exemple de développement sur plusieurs fichiers*

non plus, recompiler le fichier `B` car `Ob2` n'ayant pas été traité, il n'est pas cohérent. On va donc devoir rejouer `Ob1, Ob3`. Cela pose plusieurs problèmes:

- Il faut charger le fichier `A` puisque `Ob3` dépend d'un objet de `A`. Comme il n'a pas à être modifié, on peut charger la version compilée. (`Require A.`)
- Il faut ouvrir une section dans laquelle on déclare les variables locales du fichier `B`, jouer `Ob1` puis `Ob3` et, pour finir, refermer la section.

On peut alors vérifier et éventuellement modifier `Oc2`.

Pour vérifier `Ob2` à l'étape suivante, il faut que l'environnement contienne `Ob1`. C'est bien le cas mais puisque la section où `Ob1` a été définie, la version de `Ob1` présente dans l'environnement est celle où les variables locales ont été abstraites alors qu'elles ne le sont pas dans la version utilisée par `Ob2`. On est donc conduit à supprimer de l'environnement la section précédemment introduite ainsi que tout ce qu'elle contient (donc `Ob1`), puis à rouvrir une section où l'on rejoue `Ob1` afin d'être dans un état où `Ob2` est vérifiable.

Cela est assez lourd, aussi il nous semble plus opportun d'ajouter une restriction à notre algorithme de propagation pour qu'à chaque étape les nœuds correspondant à des objets définis dans la même section soient prioritaires.

Ainsi les nœuds sont aussi annotés par leur section de définition. Et si un objet de la section A dépend d'un objet de la section B, il ne pourra être traité que si tous les objets de la section B ont été traités précédemment (i.e. ont un statut `NORMAL` ou `MODIFIE`).

Chapitre 9

Problèmes liés aux procédures de décision automatique

9.1 Présentation du problème

Lorsque l'on modifie une portion du code ou que l'on déplace des objets dans une théorie ou d'une théorie à une autre, cela modifie sensiblement le contexte dans lequel les autres théorèmes sont prouvés. Si les modifications ou déplacements de code ont été faits en respectant les dépendances logiques, c'est-à-dire de manière cohérente avec le graphe de dépendances, on pourrait penser que cela ne pose pas de problèmes. Mais en pratique, lorsque l'on veut rejouer un script de preuve, de telles modifications du contexte ne sont pas sans conséquences. En effet les procédures de décision automatique peuvent devenir plus puissantes (pour des procédures comme la tactique `Auto` de Coq, on sait qu'elles ne deviennent pas moins puissantes car tous les lemmes qu'elles ont utilisés avant la modification apparaissent dans le graphe de dépendances et ont donc été conservés dans le nouveau contexte).

Illustrons ceci par un exemple:
Exemple 9.1

```
L1|   Variable op:nat->nat->nat.

L4|   Lemma exemple : (x,y,z,k:nat)z=y/\k=x/\k=y->(op x z)=(op y x).
L5|   Intros x y z k H;Elim H;Intros.
L6|   Rewrite H0;Auto with v62.
L7|   Elim H1;Intros.
L8|   Rewrite <-H2;Rewrite <-H3;Trivial.
Save.
```

Si ayant prouvé ce lemme, on le déplace dans un script qui, outre la ligne `L1`*, contient les lignes* `L2` *et* `L3` *suivantes:*

```
L2|   Axiom op_com:(x,y:nat)(op x y)=(op y x).
L3|   Hints Resolve op_com : v62.
```

qui établissent la commutativité de l'opération `op` *et l'ajoute dans la base de données des procédures de décision automatique, le script de preuve ne pourra pas être rejoué. En effet, en présence de* `op_com` *la tactique* `Auto` *de la ligne* `L6` *sait terminer la preuve et les lignes* `L7` *et* `L8` *n'ont plus lieu d'être.* ♢

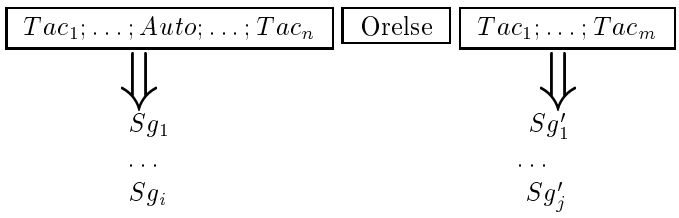
Nous nous sommes focalisés sur les problèmes posés par cette dépendance au contexte, dans la réalisation des outils d'aide à la propagation de modifications du chapitre 8, mais des difficultés semblables surgissent pour nos outils de déplacement de code et de réorganisation de théories.

Les questions principales qui se pose à nous sont:

1^o **Le nouveau script doit-il toujours être un sous-ensemble de l'ancien?**
2^o **L'ancien script est il suffisant pour adapter le nouveau?**

Autrement dit, toute tactique composée est-elle une "fonction décroissant" (la relation d'ordre étant la relation d'inclusion des ensembles de sous-buts) du pouvoir des procédures de décision automatique qu'elle contient (c'est-à-dire grossièrement pour Coq, des bases de données de résultat utilisée par ces procédures, que l'on appelle listes de Hints).

Au vu de l'exemple 9.1 on pourrait être tenté de répondre par l'affirmative. C'est globalement le cas mais il existe cependant des situations où cela peut ne pas être vrai. Ces cas pathologiques que nous allons maintenant étudier sont liés à l'utilisation des combinateurs de tactique que l'on qualifiera de non monotones (en Coq principalement Orelse et Try). Nous donnons ici un contre-exemple en utilisant conjointement la tactique `Auto` et la tacticielle `Orelse`. Le schéma général du problème est le suivant:



Dans le contexte le plus faible la partie gauche du `Orelse` échoue, c'est donc la partie droite qui est exécutée produisant un ensemble de sous-buts $Sfaible = \{Sg'_1 \ldots Sg'_j\}$ qui peut être vide. Si l'on rejoue ce script dans un contexte plus fort, la partie gauche peut ne pas échouer, elle produit alors un ensemble de sous-buts $Sfort = \{Sg_1 \ldots Sg_i\}$. Les deux ensembles de sous-buts $Sfort$ et $Sfaible$ peuvent être totalement disjoints.

Schéma 1

Voyons maintenant une instanciation de ce schéma sur un cas concret que nous détaillons.

Exemple 9.2

On fait les déclarations suivantes.

```
Inductive A: Prop -> Prop -> Prop ->Prop :=
   A_intro: (a, b, c:Prop) a -> b -> c ->(A a b c).

Inductive B: Prop -> Prop -> Prop -> Prop ->Prop :=
   B_intro: (a, b, c, d:Prop) a -> b -> c -> d ->(B a b c d).
Parameters H1, H2, H3, H4:Prop.
Parameter th1:H1.
Parameter th:(B H1 H2 H3 /\ H3 H3 /\ H3).
Parameters th2:H2; th4:H4.
```

On se place dans le contexte où seul le théorème `th1` *est présent dans la liste de* `Hint` *et peut donc être utilisé par la tactique* `Auto`.

```
Hint th1.
```

On joue le script suivant:

```
Goal (A (B H1 H2 H3 /\ H3 H3 /\ H3) (H4 /\ H4) /\ (H4 /\ H4) True).
Split; Auto.
 (Split; Auto; Apply [x, y:Prop] [h:x /\ y]h) Orelse Apply th.
```

A ce stade on obtient deux fois le sous-but `H4/\H4` *et on peut finir la preuve par*

```
Split; Apply th4.
Split; Apply th4.
```

Détaillons maintenant ce qui se passe:
La partie gauche du `Orelse` *a échoué. En effet le premier* `Split` *produit trois sous-buts:*

```
3 subgoals

  ============================
   (B H1 H2 H3/\H3 H3/\H3)

subgoal 2 is:
   (H4/\H4)/\H4/\H4
subgoal 3 is:
   True
```

Comme `th` *et* `th4` *ne sont pas dans la liste de* `Hint`, `Auto` *ne peut résoudre que le troisième.*

On continue sur le premier sous-but, le second `Split` *casse* `B` *en quatre sous-buts.*

```
 H1
subgoal 2 is:
 H2
subgoal 3 is:
 H3/\H3
subgoal 4 is:
 H3/\H3
```

`H1` *est résolue par* `Auto` *mais pas* `H2` *puisque* `th2` *n'est pas dans la liste de Hints. On essaye donc d'appliquer* `Apply [x, y:Prop] [h:x /\ y]h)` *à* `H2` *ce qui échoue puisque* `H2` *n'est pas une conjonction. On oublie donc la sous-preuve qui vient d'échouer et on applique la partie droite, soit* `Apply th.`*, qui résout le sous-but* `(B H1 H2 H3/\H3 H3/\H3)`*.*

Sur le second sous-but, la partie gauche du `Orelse` *se contente de casser le sous-but* `(H4/\H4)/\H4/\H4` *en* `H4/\H4` *et* `H4/\H4}`*.*

Si maintenant nous enrichissons le contexte par:

```
Hint th2.
Hint th4.
```

et que l'on rejoue le même script on obtient deux sous-buts:

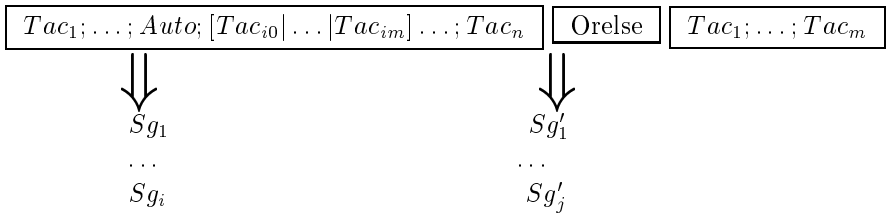
```
2 subgoals

  ============================
H3/\H3

subgoal 2 is:
 H3/\H3
```

En effet, le comportement du premier `Split` *et du premier* `Auto` *n'est pas modifié. Le second* `Split` *casse le premier sous-but restant comme précédemment mais* `H1` *et* `H2` *sont résolues par* `Auto`*.* `Apply [x, y:Prop] [h:x /\ y]h)` *s'applique donc sur les sous-buts* `H3/\H3` *qu'il laisse inchangés.*

Le second `Split` *casse, comme auparavant, le second sous-but,* `H4/\H4` *qui est maintenant résolu par* `Auto` *puisque* `H4` *a été ajouté à la liste de Hints.*

Bien que cela soit a priori moins évident, ce problème de non monotonicité peut aussi se rencontrer lorsque l'on affaiblit le contexte même si on conserve dans le contexte tous les

objets qui au vu du calcul de dépendance, sont nécessaires à la preuve. Le schéma général du problème est donné par le schéma 2.



Dans le contexte le plus fort la partie gauche du `Orelse` échoue car `Auto` produit moins de m sous-buts. C'est donc la partie droite qui est exécutée produisant un ensemble de sous-buts $Sfort = \{Sg'_1 \ldots Sg'_j\}$ qui peut être vide. Si l'on rejoue ce script dans un contexte plus faible, la partie gauche peut ne pas échouer (si Auto produit m sous-buts), elle produit alors un ensemble de sous-buts $Sfaible = \{Sg_1 \ldots Sg_i\}$. Les deux ensembles de sous-buts $Sfort$ et $Sfaible$ peuvent être totalement disjoints.

Schéma 2

Voyons maintenant une instanciation de ce schéma sur un cas concret que nous détaillons.

Exemple 9.3

On fait les déclarations suivantes:

```
L1|     Inductive A: Prop -> Prop -> Prop -> Prop :=
              A_intro: (a, b, c:Prop) a -> b -> c  ->(A a b c ).
L2|     Parameters H1, H2, H3:Prop.
L3|     Parameter th2:H2;th3:H3.
L4|     Parameter thg:H2->(A H1 H2 H3).

L5|     Parameter thg2:H3->H1.

L6|     Parameter th1:H1.
L7|     Hints Resolve th1: v62.
```

et on joue le script suivant:

```
L8|     Lemma exemple2: (A H1 H2 H3).
L9|    (Split;Auto with v62 ;
           [Try Apply th1|Apply th2|Apply th3] ) Orelse Apply thg.
L10|    Apply th2.
```

`Split` *génère 3 sous-buts, et* `Auto` *résout le premier en utilisant le théorème* `th1`*. Donc l'opérateur de composition parallèle qui attend 3 sous-buts échoue, on passe donc dans la branche droite du* `Orelse` *qui produit le but* `H2`*. On peut terminer la preuve par*

`Apply th2`. *Si l'on calcule les dépendances sur le terme de preuve ainsi obtenu on a:* $dep(exemple2) = \{thg,th2\}$.

On constate que ni `th1` *ni* `th2` *ou* `thg2` *n'apparaissent dans cet ensemble de dépendance. On pourrait donc supposer que le script pourra être rejoué dans un contexte ne contenant pas* `th1` *(c'est-à-dire en supprimant les lignes* `L6` *et* `L7`*) mais il n'en est rien.*

Dans un tel contexte `Split` *génère toujours 3 sous-buts mais* `Auto` *ne fait rien. On peut donc faire les applications parallèles de* `Try Apply th1` *(qui ne fait rien car* `th1` *n'est plus dans le contexte)* `Apply th2` *et* `Apply th3`*. Le sous-but restant n'est plus* `H2` *mais* `H1` *que l'on prouve par* `Apply thg2;Apply th3` *(qui remplace la ligne L10).*

En pratique la tacticielle `Orelse` semble néanmoins fort peut utilisée (71 fois sur les quelques 118037 lignes des "contributions" de la distribution standard de Coq soit 0,06 % du code) et nous n'avons pas trouvé de cas comme celui de l'exemple 9.3.

Hormis ces cas pathologiques nous essayons d'utiliser l'ancien script pour automatiser la modification du nouveau. Les outils de déplacement et de modification de l'arbre de preuve développés dans la première partie de cette thèse pourront être utilisés. Et même dans les cas pathologiques ils pourront servir à isoler la partie de preuve à refaire.

Nous avons donné au chapitre 8 un algorithme interactif aidant à propager les modifications du code à partir du graphe de dépendance. Le début de ce chapitre a montré qu'en pratique chaque modification change le contexte et donc tous les théorèmes qui seront rejoués après un théorème modifié sont susceptibles de changements. La section suivante présente les solutions proposées pour minimiser les problèmes posés.

9.2 Minimiser les problème liés aux procédures de décision automatique

La méthode la plus brutale consiste à "ouvrir les boîtes noires", c'est-à-dire à remplacer les procédures de décision automatique par le code qu'elles appliquent effectivement (en Coq, on peut savoir ce qui a été fait par la tactique `Auto` en la précédent de l'opérateur `Info`). Néanmoins cela ne nous parait pas satisfaisant. D'abord les preuves deviennent beaucoup plus sensibles à des changement mineurs (comme l'ajout d'une hypothèse). Ensuite on perd en généralité. En effet grâce aux procédures de décision automatique, on peut par exemple reprendre par un simple copier-coller une grand partie des scripts de preuve établissant des propriétés des listes pour prouver des propriétés équivalentes sur les vecteurs (ou listes de longueur n). Enfin, cela risque de rendre le script beaucoup plus volumineux et donc en compliquer la compréhension et la maintenance.

Une autre solution consiste à restreindre la liste des théorèmes potentiellement utilisés par les procédures automatiques.

On peut par exemple associer à chaque théorème une base de données de théorèmes réduite (la terminologie Coq parle de liste de `Hints`). Connaissant les dépendances d'un

théorème t dans un contexte ou les résultats utilisables par les procédures de décision automatique forment une base de données BD, on veut rejouer le script de preuve dans un contexte plus large (en particulier contenant au moins toutes les dépendances de t). On construit un nouvelle base de données de théorèmes pour les procédures de décision automatique suivant la formule:

Formule 1

$$Restricted\_BD \ = \ BD\_dans\_l\_ancien\_contexte \cap dep(t)$$

La mise en œuvre pratique dans le système Coq est facilitée par les réorganisations introduites dans la version 6.2.3. Depuis la version 6.2.3, Coq permet en effet de définir plusieurs bases de données. La commande `Hints Résolve ident_de_theoreme : ident_de_bd` ajoute le théorème `ident_de_theoreme` à la base `ident_de_bd`. L'utilisateur stipule ensuite explicitement quelles bases de données peuvent être utilisées par `Auto` en utilisant la tactique `Auto with ident_de_bd`.

Ainsi, pour restreindre la base de théorèmes utilisée dans la preuve d'un théorème t, on commence par créer une nouvelle base en utilisant la formule 1 puis on remplace tous les `Auto` (ou seuls ceux qui posent problème lorsque l'on veut rejouer le script) qui apparaissent dans le script de preuve par `Auto with` $Restricted\_BD$.

9.3 Une aide à la modification de théorèmes

Comme nous l'avons dit, les problèmes liés aux tactiques non monotones sont assez rares. Dans tous les autres cas, lorsque l'on rejoue un script de preuve après avoir étendu le contexte, le nouveau script est un sous-ensemble de l'ancien. Dans cette section, on propose une méthode pour essayer d'automatiser les modifications qui doivent être faites sur les scripts lorsqu'un résultat est invalidé par un changement de comportement des procédures de décision automatique.

Pour réaliser un tel outil, il faut connaître non seulement les dépendances du théorème à rejouer mais aussi son arbre de preuve dans l'ancien contexte. On a donc besoin d'un mode `debug` qui sauvegarde les arbres de preuve. Mais les arbres de preuve sont des structures volumineuses et il peut être très coûteux d'en maintenir une sauvegarde. On peut aussi faire tourner deux processus Coq en parallèle. Dans le premier on rejoue l'ancienne version du script pour obtenir, de façon dynamique, les informations désirées sur l'arbre de preuve de façon dynamique. Ces informations sont utilisées pour modifier le script que l'on joue dans le second processus. Dans le cas de la première solution l'inconvénient majeur est le coût en espace, alors que dans le cas de la seconde c'est le coût en temps puisque rejouer un script peu être très lent.

Si un de ces résultats logiquement indépendants ne peut être rejoué (on dit qu'il échoue), le système reprendra sa preuve au début en comparant chaque pas de preuve dans les deux

versions afin de déterminer quelle procédure automatique est devenue plus puissante et quelles sont les branches de l'ancien arbre de preuve qui sont désormais résolues automatiquement. Après avoir supprimé les lignes de script correspondant à ces branches désormais inutiles et décalé les indices des lignes restantes, le script sera rejoué sans encombre et on passera au théorème suivant.

Quand on rejoue un script et qu'une ligne de script comprend des tactiques composées avec `Auto` en utilisant le combinateur `Then`, il est important de savoir à combien de sous-buts `Auto` s'applique, lesquels il résout, et sur lesquels il échoue. On peut pour cela utiliser le mécanisme d'expansion de l'arbre de preuve décrit au chapitre 4.3 dans la première partie de cette thèse. Si l'on a deux sessions Coq en parallèle, on peut aussi utiliser la décomposition de l'arbre de preuve maintenue de façon interne par le système Coq pendant la durée de la preuve.

Chapitre 10

Généralisation et réutilisation de preuves

10.1 Introduction

Comme le mathématicien, l'utilisateur d'un système de preuve a parfois envie, voire besoin, de généraliser les résultats obtenus pour pouvoir les réutiliser dans d'autres développements. Pour cela, il faut montrer que les théorèmes prouvés restent vrais dans un contexte plus faible. Par exemple, les résultats de base concernant l'addition dans **N** peuvent être généralisés à tout ensemble muni d'une structure dont les lois internes vérifient les axiomes de groupe abélien.

Il est donc important de comprendre quelle partie d'un raisonnement peut être généralisée en vue d'une réutilisation ultérieure, de comprendre les analogies entre preuves et comment cela peut être exploité pour aider au développement de nouvelles théories. La généralisation et le raisonnement par analogie ont été très étudiés en Intelligence Artificielle et dans les systèmes de démonstration automatique, beaucoup moins dans le cadre des systèmes de preuve interactifs basés sur une théorie typée.

La base théorique de la généralisation est l'anti-unification [105]. Trouver la généralisation la plus spécifique de deux termes du premier ordre est désormais une chose bien comprise. Mais la généralisation des termes d'ordre supérieur le semble beaucoup moins. Dans [104] Pfenning étudie l'anti-unification dans un sous-ensemble des termes du Calcul des Constructions et montre quelques exemples d'expériences faites dans LF [103]. Robert Hasker et Uday Reddy [60] se sont intéressés à la généralisation de termes d'ordre supérieur avec une approche catégorique, et montrent que dans le cas du second ordre, l'ensemble complet des généralisations les plus spécifiques peut être calculé.

Moins ambitieux, Kolbe [67] propose d'abstraire les termes fonctionnels et c'est surtout cette démarche que nous avons regardée. On ne se propose donc pas de trouver la généralisation la moins générale de deux termes mais simplement d'aider l'utilisateur à définir des généralisations de théorèmes et de preuves déjà écrits sans présumer des instanciations futures de ces théorèmes.

Le problème n'est donc plus de prendre deux termes t_1 et t_2 et de trouver le triplet (g,θ,σ) où g est un terme et θ et σ deux substitutions tels que $\theta(g) = t_1$ et $\sigma(g) = t_2$ mais, simplement de considérer un triplet (s,t,l) où t est de type s et où l est une liste d'identificateurs libres dans s,et de trouver le plus petit ensemble l' d'identificateurs contenant l tel que si s' et t' sont le résultat de l'abstraction des éléments de l' dans s et t, t' soit de type s'.

Ce chapitre présente essentiellement au travers d'exemples les expériences pratiques que nous avons menées dans ce domaine en utilisant Coq, dans lequel, rappelons-le, un théorème correspond à un type et une preuve de ce théorème est un terme de ce type. Nous ne présentons pas de réalisations pratiques réellement utilisables mais nous avons essayé de comprendre ce qui pourrait être fait et de tracer les grandes lignes de la mise en œuvre des outils qui nous ont paru intéressants.

10.2 Le principe de base sur un exemple

Si l'on consulte la bibliothèque standard d'arithmétique du système Coq on y trouve par exemple, le lemme suivant concernant la multiplication des entiers naturels:

```
Lemma mult_permute : (n,m,p:nat) ((mult n (mult  m p))=(mult m (mult n p))).
Proof.
Intros;Rewrite -> (mult_assoc_l m n p);Rewrite -> (mult_sym m n);Auto.
Qed.
```

On désire généraliser l'énoncé du théorème `mult_permute` en un énoncé dans lequel l'opérateur ne sera plus spécifiquement `mult` mais un opérateur quelconque. Pour cela on va abstraire l'identificateur `mult` dans l'enoncé de `mult_permute`. On obtient donc l'enoncé généralisé: `(f:nat->nat->nat)(n,m,p:nat)((f n (f m p))=(f m (plus n p)))` (dans lequel on note `f` l'opérateur abstrait pour mieux le différencier de l'énoncé spécifique à `mult`). Mais cet énoncé n'est plus vrai dans le cas général.

Le problème est de trouver les propriétés que doit vérifier l'opérateur `f` pour que cet énoncé soit encore un théorème. Ici, intuitivement l'énoncé restera vrai pour toute fonction associative et commutative sur les entiers bien que ces deux propriétés n'apparaissent pas dans l'énoncé de `permute_mult`. Mais dans le cas général, pour trouver quelles propriétés doit vérifier le terme que l'on abstrait, il faut observer le terme de preuve du théorème

initial. Nous utiliserons aussi ce terme pour construire une preuve de l'énoncé généralisé. Observons donc le terme de preuve de `permute_mult`:

TERME DE PREUVE DE PERMUTE_MULT:

```
[n,m,p:nat]
  ( eq_ind_r nat (mult (mult m n) p)
    [n0:nat](mult n (mult m p))=n0
    (eq_ind_r nat (mult n m)
      [n0:nat](mult n (mult m p))=(mult n0 p)
      ( mult_assoc_l n m p) (mult m n) ( mult_sym m n))
    (mult m (mult n p)) ( mult_assoc_l m n p))
```

Outre `mult`, `nat`, et `=`, qui apparaissaient déjà dans l'énoncé, les identificateurs libres qui apparaissent dans ce terme sont **eq_ind_r**, **mult_assoc_l** et **mult_sym**. Parmi ces derniers, il faut déterminer lesquels sont relatifs à l'identificateur `mult` que l'on a abstrait dans l'énoncé.

On recherche donc leurs énoncés (c'est-à-dire le type qui leur est associé) dans l'environnement. On trouve les énoncés respectifs:

- `(A:Set; x:A; P:(A->Prop))(P x)->(y:A)y=x->(P y)`
- `(n,m:nat)(mult n m)=(mult m n)`
- `(n,m,p:nat)(mult n (mult p))=(mult (mult n m) p)`.

Il apparaît que seuls les deux derniers font référence à l'identificateur `mult`. On en déduit que les seules propriétés de la multiplication qui soient utilisées sont la symétrie et l'associativité à gauche qui correspondent à `mult_sym` et `mult_assoc_l` [1].

Il faut donc aussi abstraire ces deux identificateurs dans l'énoncé généralisé afin d'exprimer les contraintes sur l'opérateur `f`. On obtient l'énoncé suivant:

```
Lemma generalized_permute :
(f:nat->nat->nat)
(f_assoc_l :(n,m,p:nat) (f n (f m p))=(f (f n m) p))
(f_sym :(n,m:nat)(f n m)=(f m n))
(n,m,p:nat)
((f n (f m p))=(f m (f n p))).
```

1. Sur notre exemple ces deux identificateurs apparaissent aussi dans le script, ce n'est bien sur pas toujours le cas.

La preuve de ce théorème s'obtient en abstrayant `mult`,`mult_sym` et `mult_assoc_l` dans le terme de preuve initial. Ce qui conduit au terme suivant:

```
Terme de preuve de generalized_permute:
Proof [f:nat->nat->nat]
          [f_assoc_l:(n0,m0,p0:nat)(f n0 (f m0 p0))=(f (f n0 m0) p0)]
          [f_sym:(n0,m0:nat)(f n0 m0)=(f m0 n0)]
           [n,m,p:nat]
           (eq_ind_r nat (f (f m n) p) [n0:nat](f n (f m p))=n0
             (eq_ind_r nat (f n m) [n0:nat](f n (f m p))=(f n0 p)
               (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
             (f_assoc_l m n p))
```

Instanciations: On montre que l'on peut instancier `generalized_permute`, avec l'addition usuelle sur les entiers et les preuves de l'associativité et de la commutativité de cette dernière, pour obtenir une instance plus spécifique du théorème, soit:

```
Lemma plus_permute : (n,m,p:nat) ((plus n (plus m p))=(plus m (plus n p))).
Proof (generalized_permute plus plus_assoc_l plus_sym).
```

Dans la section suivante nous décrivons les grandes lignes d'un outil interactif aidant l'utilisateur à réaliser de telles généralisations de fonctions.

10.3 Un outil d'aide à la généralisation

10.3.1 Principe de fonctionnement

Étant donné un énoncé `E` dont le terme de preuve est `P`, l'utilisateur fournit le nom des fonctions qu'il désire abstraire dans `E` (pour simplifier les explications suivantes on supposera qu'il ne sélectionne qu'un identificateur de fonction que l'on notera `f`). Dans l'environnement `CtCoq`, cela peut se faire par sélection à la souris. Les opérations suivantes sont faites automatiquement

1. Rechercher le type `t` associé à l'identificateur `f`
2. Récupérer la liste de tous les identificateurs libres qui apparaissent dans le terme de preuve `P` et n'apparaissent pas déjà dans le `E`.
3. Rechercher le type associé à ces identificateurs.
4. Parmi ces identificateurs, sélectionner tous ceux dont le type fait référence à l'identificateur `f` abstrait. On notera fP_i les identificateurs retenus et tP_i leurs types (qui expriment les propriétés de `f` utilisées).

5^o Construire l'énoncé du théorème généralisé en abstrayant `f` et les fP_i dans `E`. On obtient l'énoncé généralisé de la forme:

$$(f : t)(fP_1 : tP_1) \ldots (fP_i : tP_i) \ldots E$$

6^o Construire le terme de preuve associé à cet énoncé en abstrayant sur `P`. On obtient un terme de la forme:

$$[f : t][fP_1 : tP_1] \ldots [fP_i : tP_i] \ldots P$$

10.3.2 Choix de nom des identificateurs abstraits

Comme on peut le constater sur l'énoncé du lemme `generalized_permute` donné à la section précédente, les noms donnés aux identificateurs abstraits ne sont pas choisis au hasard. On propose un petit algorithme pour nommer les propriétés sur les termes abstraits.

Il est d'usage d'exprimer une propriété d'un opérateur par le nom de cet opérateur suivi du nom de la propriété (souvent séparés par un caractère *souligné*). Pour bien exprimer qu'il s'agit d'une fonction quelconque, on nommera `f` (puis f_i les suivants si on abstrait plus d'un opérateur) l'opérateur abstrait . Il faut donc aussi substituer dans l'énoncé `f` à toutes les occurrences de cet opérateur.

Lorsque l'on a récupéré la liste d'identificateurs issus de l'étape 4 de l'algorithme précédent, on utilise leur nom pour former de nouveaux noms simplement en substituant `f` au nom de l'opérateur dans l'ancien nom de la propriété; ainsi `mult_assoc` devient `f_assoc` qui reste significatif. Ensuite on substitue `f` à toutes les occurrences du nom de l'opérateur abstrait dans les types des identificateurs de la liste récupérée à l'étape 3. Enfin on substitue `f` et les noms de propriétés formés sur `f` dans le terme de preuve.

10.4 Où l'on abstrait encore...

Jusqu'à présent on a limité l'abstraction aux fonctions. Reprenons l'exemple de la section 10.2. La propriété de permutation reste vraie pour toute fonction `f:E->E` associative et commutative, quel que soit l'ensemble `E`. Essayons de continuer notre généralisation en abstrayant le type des arguments de la fonction. On obtient un nouvel énoncé:

```
Lemma more_generalized_permute :
(E:Set)
(f:E->E->E)
(f_assoc_l :(n,m,p:E) (f n (f m p))=(f (f n m) p))
(f_sym :(n,m:E)(f n m)=(f m n))
(n,m,p:E)
((f n (f m p))=(f m (f n p))).
```

Que l'on prouve en abstrayant sur le terme de preuve, pour obtenir:

```
Terme de preuve de more_generalized_permute:
Proof [E:Set]
         [f:E->E->E]
          [f_assoc_l:(n0,m0,p0:E)(f n0 (f m0 p0))=(f (f n0 m0) p0)]
           [f_sym:(n0,m0:E)(f n0 m0)=(f m0 n0)]
            [n,m,p:E]
            (eq_ind_r E (f (f m n) p) [n0:E](f n (f m p))=n0
              (eq_ind_r E (f n m) [n0:E](f n (f m p))=(f n0 p)
                (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
              (f_assoc_l m n p))
```

Instanciations On vérifie ci-dessous que l'on peut toujours instancier cette généralisation avec la multiplication sur les entiers:

```
Lemma mult_permute : (n,m,p:nat) ((mult n (mult m p))=(mult m (mult n p))).
Exact (more_generalized_permute nat mult mult_assoc_l mult_sym ).
Subtree proved!
```

Mais essayons de l'appliquer à d'autres structures de données comme les monômes à n variable définis par Loïc Pottier[2].

Là encore, on peut utiliser notre lemme généralisé:

```
Lemma mon_permute :
(k:nat)(n,m,p:(mon k))
   ((mult_mon k n (mult_mon k m p))=(mult_mon k m (mult_mon k  n p))).
Exact [k:nat](more_generalized_permute  (mon k)
  ([i:(mon k)][j:(mon k)](mult_mon k i j ))
 ([i:(mon k)][j:(mon k)][l:(mon k)](mult_mon_assoc k i j l))
 ([i:(mon k)][j:(mon k)](mult_mon_com k i j))).
Subtree proved!
```

10.5 Limitations

On pourrait être tenté de pousser encore l'abstraction. En effet, pourquoi se limiter à la relation d'égalité `eq`, la seule propriété de cette relation qui est utilisée est le principe de récurrence suivant:
`eq_ind_r : (A:Set)(x:A)(P:A->Prop)(P x)->(y:A)y=x->(P y).`

2. (`http://www-sop.inria.fr/croap/CFC/buch/Monomials.html`)

L'énoncé reste donc vrai pour toute relation R vérifiant ce principe d'où la généralisation suivante:

```
Lemma  more_more_generalized_permute :
(E:Set)
(f:E->E->E)
(R:(A:Set)A->A->Prop)
(R_ind_r:(A:Set)(x:A)(P:A->Prop)(P x)->(y:A)(R A y x)->(P y))
(f_assoc_l :((n,m,p:E)(R E (f n (f m p)) (f (f n m) p))))
(f_sym :(n,m:E)(R E (f n m) (f m n)))
(n,m,p:E)
(R E(f n (f m p)) (f m (f n p))).
```

que l'on prouve par le terme de preuve suivant:

```
Terme de preuve de more_more_generalized_permute:
Proof [E:Set]
        [f:E->E->E]
          [R:(A:Set)A->A->Prop]
           [R_ind_r:(A:Set)(x:A)(P:A->Prop)(P x)->(y:A)(R A y x)->(P y)]
            [f_assoc_l:(n0,m0,p0:E)
                       (R E (f n0 (f m0 p0)) (f (f n0 m0) p0))]
             [f_sym:(n0,m0:E)(R E (f n0 m0) (f m0 n0))]
              [n,m,p:E]
              (R_ind_r E (f (f m n) p) [n0:E](R E (f n (f m p)) n0)
                (R_ind_r E (f n m) [n0:E](R E (f n (f m p)) (f n0 p))
                  (f_assoc_l n m p) (f m n) (f_sym m n)) (f m (f n p))
                (f_assoc_l m n p)).
```

Instanciations: On vérifie que l'on peut toujours instancier cette généralisation: Avec la multiplication sur les entiers:

```
Lemma mult_permute : (n,m,p:nat) ((mult n (mult m p))=(mult m (mult n p))).
Intros.
Exact (more2_generalized_permute
         nat n m p mult eq eq_ind_r mult_assoc_l mult_sym).
Subtree proved!
```

Mais eq_ind_r caractérise l'égalité de Leibnitz, et on ne pourra donc pas réutiliser ce lemme avec d'autres égalités définies.

Considérons par exemple les polynômes tels qu'ils ont été définis par Loïc Pottier et Laurent Théry. Ils sont représentés par le tableau de leurs coefficients. L'addition de deux polynômes s'obtient simplement en sommant leurs coefficients de même degré. Une relation

d'égalité `eqP` est définie pour identifier par exemple 0 et $O + (O * X)$ qui sont différents pour l'égalité `eq`.

On a donc essayé de prouver `permute` pour l'addition de ces polynômes, soit le lemme suivant:

```
Lemma plusP_permute :
   (n,m,p:P) (eqP (plusP n (plusP m p)) (plusP m (plusP n p))).
```

Mais il n'est pas possible d'utiliser le terme de preuve précédent. On peut néanmoins terminer la preuve. On pourrait vouloir établir une généralisation de `eq_ind_r` comme par exemple dans le cas des polynômes à coefficients entiers.

```
Lemma eq_ind_r_int:
( x:(P nat); P1:((P nat)->Prop))
   (P1 x)->(y:(P nat))(eqP nat(eq nat) [x:nat]x=0  y x)->(P1 y)
```

Mais ce résultat n'est pas vrai, sinon en prenant `[z:(P nat)]x=z)` pour `P1` on montre que `eqP` implique `eq`, ce qui n'est pas vrai.

On peut néanmoins terminer la preuve de `plusP_permute` par le script suivant:

```
Intros.
(Apply eqP_trans with (plusP (plusP m n) p); Auto).
(Apply eqP_trans with (plusP (plusP n m) p); Auto).
```

Dont le terme de preuve n'a rien à voir avec ce que l'on avait précédemment:

```
Proof: [n,m,p:P]
        (eqP_trans (plusP n (plusP m p)) (plusP (plusP m n) p)
          (plusP m (plusP n p))
          (eqP_trans (plusP n (plusP m p)) (plusP (plusP n m) p)
            (plusP (plusP m n) p) (plusP_associative n m p)
            (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
              (eqP_sym (plusP (plusP n m) p) (plusP (plusP m n) p)
                (eqP_sym (plusP (plusP m n) p) (plusP (plusP n m) p)
                  (plusP_compl (plusP m n) (plusP n m) p
                    (plusP_commutative m n))))))
          (eqP_sym (plusP m (plusP n p)) (plusP (plusP m n) p)
            (plusP_associative m n p)))
```

Cet exemple montre qu'il n'est donc pas forcément souhaitable de généraliser au maximum, c'est-à-dire d'abstraire tous les identificateurs libres apparaissant dans un énoncé.

10.6 Robustesse et problèmes

En pratique, on ne peut garantir que notre généralisation d'un théorème `T` fonctionnera que si sa preuve ne fait référence qu'aux types des identificateurs généralisés et n'utilise pas leurs structures internes, c'est-à-dire si la dépendance vis-à-vis de ces termes est opaque. Pour les types inductifs, cela veut dire que l'on ne fait pas de raisonnement par cas ou par induction, d'ailleurs s'il est toujours possible d'abstraire un récurseur, bien que l'exemple précédent montre que ce n'est pas très utile, on ne pourrait rien faire lors que c'est un `Fix` ou un `Case` qui apparaît dans le terme de preuve.

Cette restriction est aussi vraie pour les termes fonctionnels, nos premiers exemples ne fonctionnent que parce que les termes de preuve ne font pas directement référence à la structure de `mult`; les propriétés de `mult` ont été prouvées séparément et peuvent donc être abstraites. Si ces propriétés de base étaient prouvées directement dans la preuve on ne pourrait pas abstraire.

En effet pour prouver la plupart des propriétés basiques des fonctions, on a besoin d'avoir accès à leur structure de base. Ainsi, pour prouver la commutativité de l'addition, il faut utiliser sa structure interne et des règles de réécriture qui permettent par exemple de réduire `(O+x)` en `x`. La réduction d'un terme fonctionnel défini par de telles règles s'appelle la ι-réduction. Cela conduit à des preuves très différentes pour la commutativité de l'addition ou de la multiplication des entiers, et à une preuve encore plus différentes la commutativité de la multiplication des monômes.

Kolbe et Walter [67] ont étudié la réutilisation de telles preuves, mais ils semblent se limiter à de petits exemples et malgré cela le résultat reste encore très dépendant de la structure définie. Ainsi certaines preuves faites sur l'addition pourront s'appliquer à la multiplication si on la définit par les règles : `O*m =>O` et `(S p)*m =>m + ( p * m))` mais pas si on la définit par le système `O*m =>O` et `(S p)*m =>( m * p) + m)`, qui est pourtant équivalent.

10.7 Restriction dans l'utilisation de l'abstraction.

Pour contourner ces problèmes, on n'autorisera à abstraire sur un identificateur de fonction que s'il est opaque ou si on ne le réduit pas dans la preuve. De même, on n'autorisera à abstraire un type inductif que s'il correspond à un "type muet" c'est-à-dire si la preuve ne contient pas de raisonnement par cas ou par induction sur la structure de ce type. Pour vérifier que l'on ne réduit pas un terme, on traquera dans le script les tactiques pouvant faire des réductions comme `Simpl`, `Change` .... De même on cherchera dans le script les tactiques permettant de déstructurer un type inductif comme `Induction`, `Split`, `Case` .... On vérifiera aussi que les constructeurs du type n'apparaissent ni dans le script ni dans le terme de preuve.

Mais cette méthode n'est pas fiable à 100% ne serait-ce que parce que l'utilisateur peut définir ses propres tactiques.

10.8 Conclusion et perspectives

L'idée principale de cet outil est d'éviter de repartir de zéro lorsque l'on va développer des théories abstraites comme par exemple la théorie des groupes. En effet dans les développements existants de nombreuses preuves n'utilisent que quelques propriétés des objets sur lesquelles elles travaillent.

On peut concevoir de fournir un mode "debug" dans lequel chaque propriété utilisée dans les preuves sur des fonctions sera généralisée et stockée. Lorsque par la suite on aura à définir une structure, on donnera ses propriétés abstraites et en comparant avec notre stockage et, modulo les restrictions du § 10.7, on pourra récupérer gratuitement dans les développements existants tous les théorèmes qui sont valables pour notre structure.

La certification d'algorithmes mathématiques destinés au calcul formel doit s'appuyer sur une implémentation certifiée des structures mathématiques de base (groupes, anneaux, corps etc.). A terme, la généralisation devrait permettre de récupérer, en les généralisant, les preuves qui ont été faites pour des instances de ces théories comme les entiers ou les polynômes. En relation avec l'introduction des modules, elle pourrait permettre de développer des théories paramètrées à partir d'instances spécifiques, un peu comme le font dans le domaine des langages de programmation Michael Siff et Thomas Reps [125], lorsqu'ils essayent de générer du code C++ générique à partir de code C.

Chapitre 11

Conclusion

Nous avons montré l'importance des activités de maintenance pour les systèmes de preuve.

En nous inspirant de ce qui se fait en génie logiciel, nous avons fourni des outils assistant l'utilisateur dans le développement et la maintenance. Ces outils s'appuient sur la notion fondamentale de dépendances.

Nous avons discerné plusieurs niveaux de dépendance:

- Les dépendances entre les différentes parties d'une preuve, que nous qualifions de dépendances locales.
- Les dépendances entre théories et entre objets d'une théorie, que nous qualifions de dépendances globales.

Dans un premier temps, nous nous sommes focalisés sur la maintenance des scripts de preuve. La notion de script de preuve et les notions connexes de tactique et de tacticielle issues de LCF sont en effet communes à de nombreux assistants de preuves comme nous l'avons montré dans une présentation rapide de quelques-uns des principaux prouveurs. A tout script de preuve, on peut associer de façon naturelle, une structure d'arbre. Cette structure reflète les dépendances entre les différents sous-buts. En nous basant sur cette notion d'arbre nous avons fourni des outils génériques pour assister l'utilisateur dans ses activités de développement, de maintenance et de transformation de script. Les principales aides sont les suivantes:

- Compréhension, et visualisation de la structure d'une preuve:
  (visualisation graphique de l'arbre et outils de navigation dans le script.)
- Retour-arrière logique;
- Expansion et contraction de script;
- Lemmifcation et factorisation de lemme.

Il est apparu que dans les systèmes manipulant des variables existentielles, l'arbre de preuve ne suffit plus à représenter les dépendances entre les différents sous-buts. Nous avons proposé une modélisation des dépendances supplémentaires introduites par le partage de variables existentielles entre les sous-buts. Et nous avons tenté quand cela été possible d'adapter les outils précités pour qu'ils prennent en compte ces contraintes supplémentaires.

Dans un second temps nous avons brièvement étudié les dépendances entre théories puis nous nous sommes focalisés sur les dépendances entre objets d'une théorie. Nous les avons modélisées au travers de la notion de graphe de dépendance.

Nous avons proposé différentes utilisations basées sur ce graphe:

- La coupe de théorie;
- L'aide à la de modification de théories, pour laquelle nous avons conçu un algorithme interactif de propagation des modifications;
- La réorganisation de théorie.

Toutes ces notions sont assez génériques, néanmoins la manière dont les dépendances entre objets sont calculées est relativement dépendante du système utilisé. Dans les systèmes basés sur une théorie des types on pourra, comme nous l'avons fait dans le cas de Coq, calculer ces dépendances sur le terme de preuve. Dans le cas des systèmes comme HOL, ou il n'y a pas de terme de preuve, on pourra par exemple utiliser la sauvegarde sous forme de règles primitives.

Le problème principal que nous avons rencontré dans cette partie est la sensibilité au contexte des procédures de décision automatique que l'on retrouve sous des formes différentes dans la plupart des systèmes. Ce problème fragilise nos manipulations mais, là encore, nous avons essayé d'envisager des solutions, parfois restrictives, augmentant la sûreté de ces transformations.

Enfin dans un dernier chapitre on s'est intéressé à l'abstraction sur les théories. Mais notre contribution s'est limitée à essayer de cerner au travers de quelques expériences ce qui pourrait être fait.

En conclusion, précisons que cette thèse est avant tout un travail de défrichage. Il n'existe à notre connaissance pas de contribution globale sur les problèmes de maintenance de théorie dans les systèmes de preuve même si de nombreux articles expriment le besoin de tels outils. Nous espérons, par les outils développés et les idées exprimées, avoir ouvert la voie à de tels développements qui, alors que l'utilisation des systèmes de preuve formelle se répand et que les utilisateurs s'attaquent à des preuves de plus en plus conséquentes, devraient devenir de plus en plus indispensables.

Annexes

Annexe A

Exemple de sessions de preuve dans différents systèmes

A.1 HOL

Dans cet exemple, on a utilisé la version HOL98 de K. Slind [127].

Après le lancement de HOL, on est dans la boucle d'interaction de Moscow ML, on peut taper des commandes ML classiques. Les termes de la logique sont des objets de type `term` définis dans le module `Term`. Les théorèmes ont le type `thm` du module `Thm`. On entre un nouvel énoncé avec la fonction `g:Term.term frag list ->GoalstackPure.proofs`.

La fonction `e` de type

```
(Term.term list * Term.term ->
       (Term.term list * Term.term) list * (Thm.thm list -> Thm.thm)) ->
       GoalstackPure.goalstack
```

permet d'appliquer une tactique. Les tactiques sont des fonctions dont le type est :

```
Term.term list * Term.term ->
      (Term.term list * Term.term) list * (Thm.thm list -> Thm.thm)
```

Elles prennent un couple, formé d'une liste de termes représentant les hypothèses et d'un terme représentant le but courant. Elles renvoient un couple dont le premier membre est une liste de couples, contenant la liste d'hypothèses et l'énoncé de chacun des sous-buts générés, et le second une fonction de validation qui est capable, à partir de la liste des preuves des théorèmes du premier membre, de reconstruire la preuve du théorème initial.

On lance HOL:

```
aenegada:/net/croap/lib/hol98/bin [47] > ./hol
Moscow ML version 1.43 (April 1998)
Enter ‘quit();’ to quit.
For HOL help, type: help "hol";

        HHH                 LL
        HHH                  LL
        HHH                   LL
        HHH                    LL
        HHH          OOOO       LL
        HHHHHHH     OO  OO       LL
        HHHHHHH     OO  OO       LLL
        HHH          OOOO        LLLL
        HHH                     LL  LL
        HHH                    LL    LL
        HHH                   LL      LL
        HHH                  LL        LL98 [Athabasca 2]

[closing file "/net/croap/lib/hol98/std.prelude"]

- (* on peut taper des commandes ml *)
 val a=1+3;
> val a = 4 : int
-
```

On doit d'abord charger quelques théories

```
load "bossLib";
open bossLib;
open arithmeticTheory;
load "prim_recTheory";
open prim_recTheory;
```

Puis on peut attaquer la preuve du théorème suivant.

$$\forall m\ n\ \in N, n < m \Longrightarrow \forall p \in N,\ m < n + p$$

```
g‘! m n. m < n ==> !p. m < n + p‘;;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !m n. m < n ==> (!p. m < n + p)



    : GoalstackPure.proofs
```

```
- - e(REPEAT GEN_TAC);
OK..
1 subgoal:
> val it =
    m < n ==> (!p. m < n + p)


    : GoalstackPure.goalstack
- e(STRIP_TAC);               (* introduction *)
OK..
1 subgoal:
> val it =
    !p. m < n + p
    ------------------------------------
      m < n

    : GoalstackPure.goalstack
- e(Induct_on `p`);            (* induction sur p produit deux sous-buts *)
OK..
2 subgoals:
> val it =
    m < n + SUC p
    ------------------------------------
      0.  m < n
      1.  m < n + p

    m < n + 0
    ------------------------------------
      m < n

    : GoalstackPure.goalstack
- r 1;                         (* change l'ordre des sous-buts *)
> val it =
    m < n + 0
    ------------------------------------
      m < n

    m < n + SUC p
    ------------------------------------
      0.  m < n
      1.  m < n + p

    : GoalstackPure.goalstack
- (* étape d'induction *)
e(IMP_RES_TAC LESS_SUC );
OK..
```

```
1 subgoal:
> val it =
    m < n + SUC p
    ------------------------------------
      0.  m < n
      1.  m < n + p
      2.  m < SUC (n + p)
      3.  m < SUC n

    : GoalstackPure.goalstack
- e(ASM_REWRITE_TAC [ADD_CLAUSES]);
OK..

Goal proved.
 [.] |- m < n + SUC p

Goal proved.
 [..] |- m < n + SUC p

Remaining subgoals:
> val it =
    m < n + 0
    ------------------------------------
      m < n

    : GoalstackPure.goalstack
- (* cas de base *)
e(ASM_REWRITE_TAC [ADD_CLAUSES]);
OK..

Goal proved.
 [.] |- m < n + 0

Goal proved.
 [.] |- !p. m < n + p

Goal proved.
|- m < n ==> (!p. m < n + p)
> val it =
    Initial goal proved.
    |- !m n. m < n ==> (!p. m < n + p)
    : GoalstackPure.goalstack
```

A.2 Isabelle

Pour cet exemple, on a utilisé la version Isabelle98-1 en SML-NJ.

La session de preuve reproduite ci-dessous montre une preuve de la commutativité de la conjonction.

```
aenegada:/net/croap/lib/Isabelle98-1/bin [8] > ./isabelle
val it = false : bool
> val commit = fn : unit -> bool
> goal HOL.thy
"(A & B)-->(B & A)";#        (* on utilise la théorie HOL *)

Level 0
A & B --> B & A
 1. A & B --> B & A

val it = [] : thm list
> br impI 1;                 (* introduction de l'hypothèse *)

Level 1
A & B --> B & A
 1. A & B ==> B & A

val it = () : unit
> br conjI 1;                 (* introduction de la conjonction *)

Level 2
A & B --> B & A
 1. A & B ==> B
 2. A & B ==> A

val it = () : unit
> be conjE 1;                 (* élimination de la conjonction dans le
                                 premier sous-but                      *)

Level 3
A & B --> B & A
 1. [| A; B |] ==> B
 2. A & B ==> A

val it = () : unit
> ba 1;                       (* Assumption *)

Level 4
A & B --> B & A
```

```
 1. A & B ==> A

val it = () : unit
> be conjE 1;

Level 5
A & B --> B & A
 1. [| A; B |] ==> A

val it = () : unit
> ba 1;

Level 6
A & B --> B & A
No subgoals!

val it = () : unit
> qed "and_comms";
val and_comms = "?A & ?B --> ?B & ?A" : thm
val it = () : unit
```

A.3 Lego

Au démarrage de Lego, on est, comme dans Coq, dans une boucle d'interaction propre au système. Il est impossible de taper du code ML. On ne peut pas comme en HOL, écrire de nouvelles fonctions pour étendre facilement le système de tactique.

Chargeons le paquetage `Logic` contenant les définitions des opérateurs logiques de base et définissons un type A et deux prédicats f et g sur A

```
Logic;
[A:Type][f,g:A->Prop];
```

On se propose maintenant de prouver:

$$\forall a \in A, (f\ a) \wedge (g\ a) \Longrightarrow \forall b \in A (g\ b) \wedge (f\ b)$$

On donne ci-dessous la session de preuve complète

```
Lego> Goal (<a:A>(and (f a)(g a)))-><b:A>(and (g b)(f b));
Goal
  &?0 : (<a:A>(f a /\ g a))-><b:A>(g b /\ f b)
```

```
Lego> Intros s;              (* introduction de l'hypothèse *)
Intros (1) s
  s : <a:A>(f a /\ g a)
  &?1 : <b:A>(g b /\ f b)
Lego> Intros #;              (* introduction d'une variable existentielle *)
Intros (0) #
  &?2 : A
  &?3 : g ?2 /\ f ?2
Lego> Next +1;               (* permutation des sous-buts *)
Next 3
Lego> andE s.2;              (* on casse le /\ dans l'hypothèse *)
and Elim
-- Start of Goals --
  s : <a:A>(f a /\ g a)
  H : f s.1
  H1 : g s.1
  &?6 : g ?2 /\ f ?2
-- End of Goals --
Lego> andI;                 (* Elimination du  /\ dans le but *)
and Intro
-- Start of Goals --
  s : <a:A>(f a /\ g a)
  H : f s.1
  H1 : g s.1
  &?9 : g ?2
  &?10 : f ?2
-- End of Goals --
Lego> Refine H1;            (* unification de H1 avec le but courant &?9 *)
Refine by  H1
  &?10 : f s.1
Lego> Immed;                (* tout est dans le contexte ! *)
Immediate
Discharge..  H1 H
Discharge..  s
*** QED ***   (* time= 0.010000  gc= 0.0  sys= 0.010000 *)
```

A.4 PVS

Pour cet exemple, on a utilisé la version 2.2 de PVS. L'exemple présenté est issu du cours introductif présenté par Judy Crow, Sam Owre, John Rushby, Natarajan Shankar et Mandayam Srivas [29] à WITF'95. On commence par construire une mini-théorie où l'on définit une fonction qui calcule la somme des n premiers entiers.

```
sum: THEORY
```

```
BEGIN

n: VAR nat

sum(n): RECURSIVE nat =
 (IF n = 0 THEN 0 ELSE n + sum(n - 1) ENDIF)
 MEASURE id

closed_form: THEOREM sum(n) = (n * (n + 1))/2
```

La vérification de type génère deux obligations de preuve. La première est liée au fait que le type de la division dépend d'un prédicat exprimant que le second argument n'est pas nul. La seconde est liée à la terminaison de la fonction, elle stipule simplement que l'on récure sur un ordre bien fondé..

```
sum_TCC1: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES n - 1 >= 0);
sum_TCC2: OBLIGATION (FORALL (n: nat): NOT n = 0 IMPLIES id(n - 1) <id(n));
```

Une fois ces obligations de preuve démontrées (ce qui ici se fait automatiquement), on va prouver que cette somme est toujours égale à $\frac{(n*(n+1))}{2}$. Pour cela on entre en mode preuve (`M-x prove`) après avoir sélectionné le lemme qui nous intéresse avec la souris. On rentre alors les tactiques comme dans les autres systèmes. Il est possible de visualiser l'arbre de preuve au fur et à mesure du développement (mais comme nous l'avons expliqué au chapitre 3, pour de "grosses preuves", cela devient vite impraticable). Pour faire notre preuve on rentrera la suite de tactiques suivante:

```
(INDUCT "n")
(EXPAND "sum")
(ASSERT)
(SKOLEM!)
(FLATTEN)
(EXPAND "sum" +)
(ASSERT)
```

Une fois finie, la preuve peut être sauvegardée et pourra être rechargée automatiquement lors d'une session ultérieure. Remarquons que les obligations de preuve sont comprises dans la preuve sauvegardée que l'on montre ci-dessous.

```
(|sum| (|sum_TCC1| "" (SKOSIMP) (("" (ASSERT) NIL)))
 (|sum_TCC2| "" (GRIND) NIL)
 (|closed_form| "" (INDUCT "n")
  (("1" (EXPAND "sum") (("1" (ASSERT) NIL)))
   ("2" (SKOLEM!)
    (("2" (FLATTEN) (("2" (EXPAND "sum" +) (("2" (ASSERT) NIL)))))))))))
```

Enfin nous pouvons demander de quoi dépend le théorème prouvé (`M-x spc`). Pour le théorème `closed_form` le système répond:

```
sum.closed_form has been PROVED.

  The proof chain for closed_form is COMPLETE.

  closed_form depends on the following proved theorems:
    sum.sum_TCC2
    identity.I_TCC1
    if_def.IF_TCC1
    sum.sum_TCC1
    naturalnumbers.nat_induction

  closed_form depends on the following definitions:
    notequal./=
    functions.bijective?
    functions.surjective?
    functions.injective?
    reals.>=
    sum.sum
    reals.<=
    identity.id
```

La figure A.1 montre l'environnemt PVS et la représentation de l'arbre de preuve. Cliquer sur les symboles $\vdash$ permet de visualiser le sous-but associé.

A.5 Alf

On termine en montrant une session de travail dans Alf. On y voit deux multi-fenêtres. La fenêtre de droite contient la preuve en cours tandis que la fenêtre de gauche contient la théorie en cours de construction. Une preuve terminée peut être ajoutée dans la théorie en la changeant de fenêtre. De même une preuve peut être copiée ou déplacée de la théorie vers la fenêtre de développement pour être modifiée.

Sur la figure suivante, dans la fenêtre du haut de la multi-fenêtre de gauche on est en train de prouver, le lemme:

$$\forall n,m \in N, m \leq (m+m)$$

La preuve se fait en éditant un terme ayant le type voulu. Sur la figure A.2,

il ne reste plus qu'un "trou" représenté par la variable $?_u$. La fenêtre du milieu de la multi-fenêtre visualise les contraintes sur cette méta-variable représentant le sous-terme

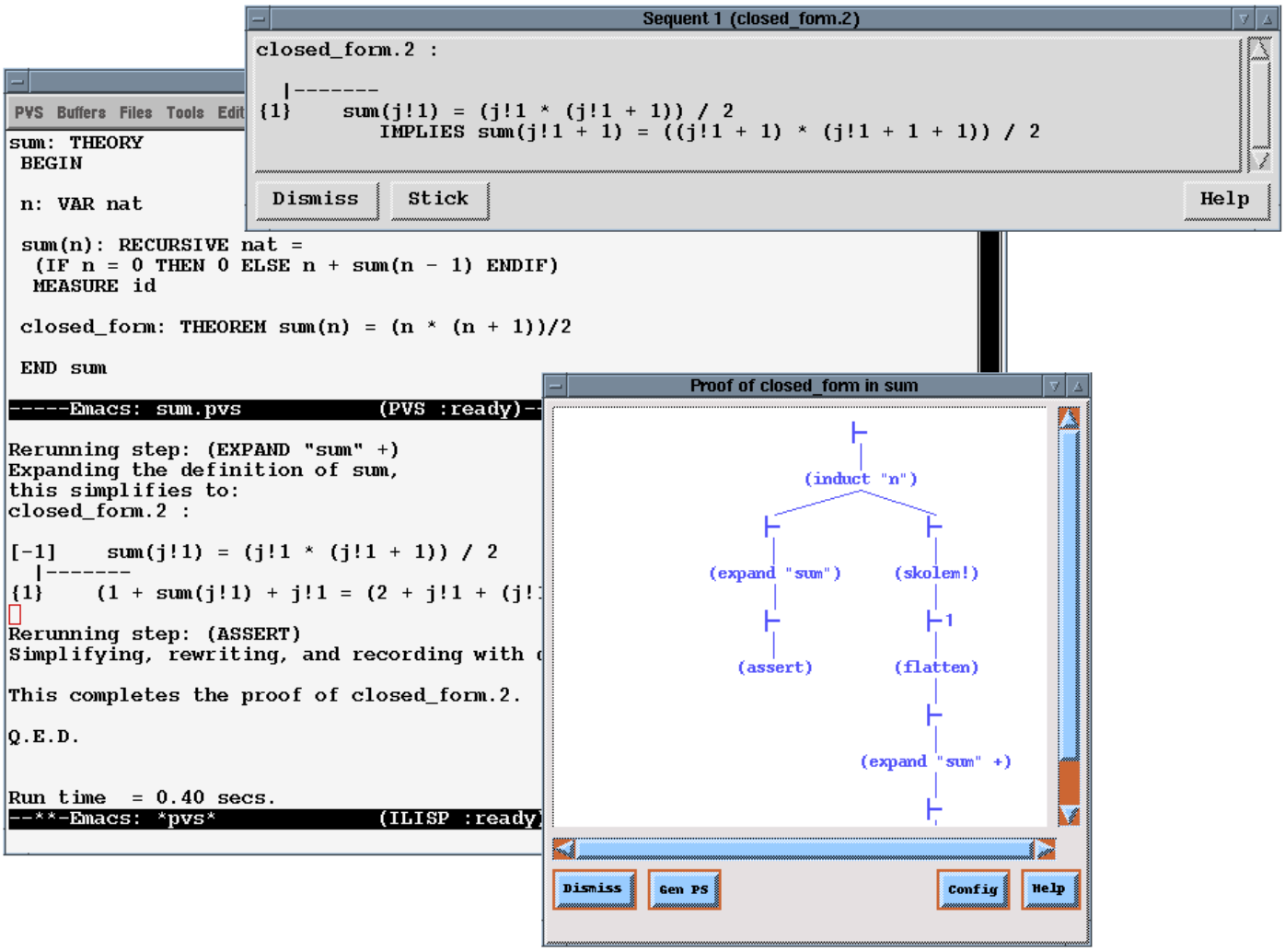


FIG. A.1 – *Environnent de travail du système PVS*

restant a construire. Enfin la fenêtre du bas montre le type du sous-terme sélectionné. Enfin une barre de menu donne accès à différentes aides pour l'édition du terme. Le menu `Matching` propose notamment une liste de constantes pour l'instanciation d'une méta-variable.

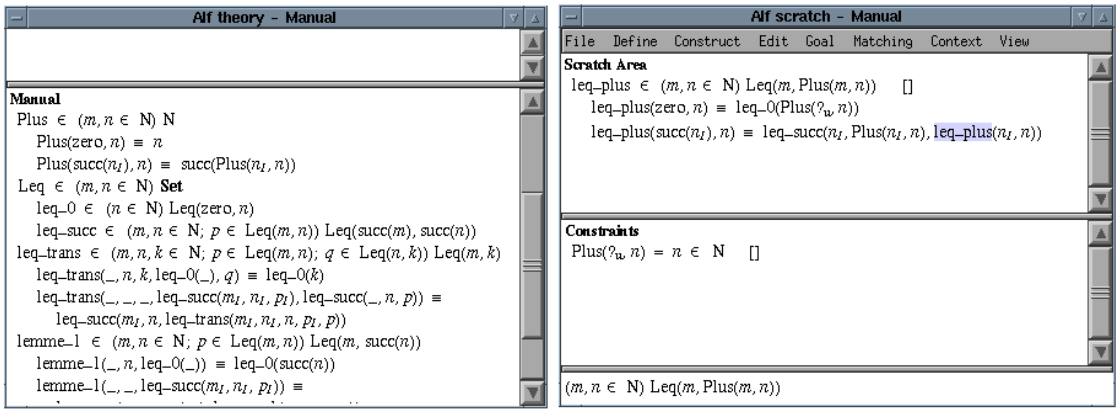


FIG. A.2 – *Environnent de travail du système Alf*

Annexe B

Une interface utilisant daVinci et Ocaml

Introduction

Nous décrivons ici une interface *Ocaml* [140] au logiciel de visualisation de graphe *daVinci* [51, 52] développé à l'université de Brême. Puis nous décrivons un prototype d'interface graphique pour le système Coq basé sur l'interface Caml.

B.1 Interfacer daVinci et Caml

L'objectif de ce package *ML* est de fournir un outil pour la manipulation et l'affichage de graphes en *Ocaml*. La base des fonctionnalités fournies est celle des API de *daVinci* (ajout/suppression de nœuds/arcs, modification des caractéristiques d'un nœud/arcs, vue multiple, navigation ...).

Le visualisateur daVinci

daVinci est un outil interactif de visualisation de graphes orientés, (écrit en *ASpecT* puis compilé vers du C, et disposant d'un interface en Tcl/Tk [91]). L'affichage peut être optimisé en tentant de minimiser le nombre de croisements des arêtes. L'algorithme utilisé est une variante de celui de Sugiyama [129, 130].

B.1.1 Une idée de l'implémentation

La structure interne des graphes de *daVinci* n'étant pas aisée à manipuler, on maintient notre propre structure de graphe en *ML* qui est ensuite compilée vers celle du visualisateur.

Représentation des graphes

Nous avons représenté les graphes comme par des listes d'adjacence doublement chaînées (on a la liste des fils et la listes des pères de chaque nœud). On pourrait aussi utiliser une représentation matricielle.

On donne ici la signature (simplifiée) du type abstrait GRAPH.

```
module type GRAPH =
  sig
    type 'a graph
    val create : ('a * 'a list) list  ->'a graph
    val createdouble : ('a*('a list*'a list)) list ->'a graph
    val empty : 'a graph
    val newGraph : unit ->'a graph
    val addNode :'a->'a graph ->'a graph
    val addEdge :'a graph ->'a->'a->'a graph
    val suppressNode   :'a->'a graph ->'a graph

    val listOfVertices :'a graph ->'a list
    val listOfSuccessors : 'a->'a graph ->'a list
    val listOfPredecessors  : 'a->'a graph ->'a list

    (* liste des suivants atteingnables *)
    val  listOfPossible : 'a->'a graph ->'a list
    (* liste des precedants atteingnables *)
    val  listOfPossibleUp :'a->'a graph ->'a list
    (* couper une branche *)
    val  cutSubGraph :'a->'a graph ->'a graph
  end;;
```

Communication daVinci-Ocaml

La connection entre Ocaml et daVinci se fait par "pipe". La figure B.1, issue de la documentation en ligne de *daVinci* en illustre le principe.

Cette communication se réalise via deux pipes UNIX: un pour envoyer les commandes aux API de *daVinci* , un pour recevoir les réponses. Côté Caml nous gérons la connection en utilisant les possibilités de la bibliothèque Unix [70].

L'envoi de message à daVinci. L'envoi de message vers *daVinci* se fait simplement en écrivant sur le canal de sortie. Nous aurons donc une fonction `sendMsg : string ->unit` qui envoie son argument au visualisateur en l'écrivant sur le bon canal. Puis on utilise cette fonction pour implémenter toute les fonctionnalités des API de *daVinci*.

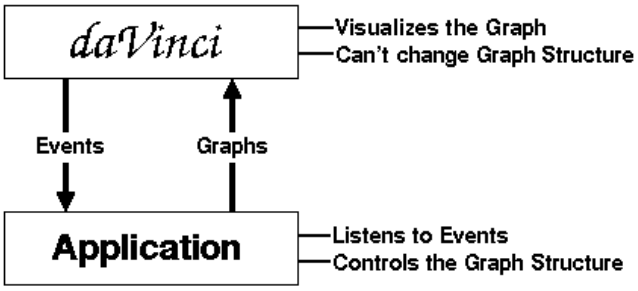


FIG. B.1 – *Communication* `daVinci-Ocaml`

Lire les réponses de daVinci. De même on récupère les messages du visualisateur en lisant sur le canal d'entrée. Mais il faut être sûr qu'il y ait quelque chose à lire, sinon le processus de lecture est bloqué jusqu'à ce qu'il y ait de nouveau quelque chose à lire (par exemple la réponse à la sélection d'un nœud).

Cette vérification se fait grâce à la fonction `select` de la bibliothèque UNIX d'Ocaml. qui permet d'attendre que l'opération de lecture soit possible sur le canal (elle attend qu'un "descripteur de fichier" change de statut).

Les réponses sont des chaînes de caractères qui correspondent à un message avec d'éventuels arguments, par exemple le message `node_selections_labels(["4"])` signale que le nœud identifié par "4" a été sélectionné.

Nous faisons donc l'analyse syntaxique de ces chaînes pour en récupérer le message et les éventuels arguments.

Le top level. Le traitement global se fait par une boucle dans laquelle via `select` on attend des événements au clavier ou sur le graphe.

```
let  main tc ta=
 let rec mainl() =
  let j = ref [] in
    while  (!j=[])
    do
    let indl,outdl,ol=
    select [(descr_of_in_channel !inc);(descr_of_in_channel  Pervasives.stdin)]
       [(descr_of_out_channel !outc);(descr_of_out_channel  Pervasives.stdout)]
       [] 2.0 in
   j:=indl
   done;
   List.iter
    (fun x -> let u=(input_line (in_channel_of_descr x)) in
      (try ta (reaction u) with |_ -> if u="fin" then raise Exit else (tc u)))
```

```
      !j;
    print_string"\n";flush  Pervasives.stdout;mainl()
 in mainl();;
```

C'est une fonction dont les deux arguments sont les fonctions de traitement du texte entrées au clavier et des messages reçus du visualisateur.

Utilisation du package.

Pour utiliser ce package Ocaml de manipulation de graphe, il faut d'abord charger le module d'interface puis définir les fonctions de réaction `reactionText` et `reactionInterface` appropriés à notre application. On peut éventuellement écrire une fonction `createBasicMenu` qui ajoutera des entrées au menu `Edit` de l'interface de visualisation. Puis on définit la fonction principale.

```
let init() =
 (* initialiastion des constantes *)
 .............
 (* creation de la fenetre de visualisation et d'un graphe vide *)
 initialisation();
 (* creation d'un eventuel menu *)
  createBasicMenu ();
  (* appel de la boucle principale
  avec en argument les deux fonctions de réaction*)
  main  reactionText reactionInterface;
 (* initialisation du menu *)
  createBasicMenu () ;;
```

B.2 Application

B.2.1 Un mini éditeur de graphe

A titre d'exemple on a écrit un éditeur de graphe très rudimentaire. On y crée un menu contenant quatre entrées: `Addnode` qui nous fait passer dans un mode où chaque click souris sur un nœud, crée un nouveau nœud ; `AddEdge` qui nous met dans un mode où l'on peut ajouter des arêtes, chaque "click impair" sélectionnant un nœud source et chaque "click pair" le nœud cible ; `CutSubgraph` qui permet de couper le sous graphe issu du nœud que l'on sélectionne. Enfin `NoInteraction` qui nous met dans un mode où toutes les sélections sont inactives. La fonction de réaction `reactionInterface` gère les réactions en fonction du mode dans lequel on se trouve. Par défaut la fonction `reactionText` ne fait rien.

B.2.2 Affichage de dépendances entre fichiers.

B.2.3 Une interface graphique pour *Coq*

On présente maintenant un prototype très rudimentaire d'interface graphique pour le système *Coq*. Les fonctionnalités proposées sont celles qui ont été décrites au cours de cette thèse.

Nous court-circuitons le "top-level" de *Coq*, la fonction de traitement du texte entré au clavier, reconnaît et traite les commandes *Coq* . On considère qu'une commande est complète dés qu'un point (le terminateur de commande Coq) est entré. On transmet alors la commande à l'analyseur syntaxique puis à l'évaluateur de *Coq* qui répond en affichant le résultat de la commande.

Plusieurs modes sont possibles:

- **L'affichage de l'arbre de preuve et undo logique:** Si on est bien dans une preuve et que la commande est une tactique, le système affiche les buts restant à prouver. Si l'évaluation a réussi et que la commande est une tactique on peut ajouter dans le graphe le nœud correspondant à cette tactique.
  Nous maintenons une table qui associe à chaque tactique le but qu'elle attaque. On peut ainsi visualiser en parallèle les buts et les tactiques qui les résolvent. On dispose d'un commande `KillProofNode` qui permet d'effacer une branche de l'arbre de preuve par un simple click sur le nœud correspondant.
- **Affichage du graphe de coercion:**
  Lorsque l'on développe de grosses théories en utilisant les coercions, notamment en algèbre, il peut devenir difficile savoir de quelles coercions on dispose. On propose donc de visualiser le graphe de coercions. On peut alors se déplacer dans le graphe, utiliser les fonctionnalités de zoom du visualisateur, obtenir de l'information sur un type ou une fonction de coercion par simple click souris.
- **Affichage de l'environnement et reset logique:** On peut visualiser le graphe de dépendance entre objets, qui croît chaque fois qu'un objet est ajouté à l'environnement. La figure B.2 illustre cette possibilité. Elle visualise les dépendances d'une session de preuve sur les listes en cours de développement.
  On a deux niveaux de visualisation. Dans le premier, tous les objets introduits depuis le début de la session sont représentés ainsi que tous les objets "extérieurs" qu'ils utilisent. Les objets "extérieurs" sont ceux qui sont définis dans d'autres fichiers qui peuvent être importés dans le développement courant. On différencie ces objets en associant une couleur différente à chaque fichiers. Dès que le développement devient trop général, ce niveau de visualisation devient inutilisable. On regroupera donc tous les objets d'un même fichier en un seul objet. On utilise aussi les couleurs pour distinguer les différents types de dépendances (le rouge pour les dépendances opaques, le bleu pour les dépendances transparentes).
  L'interface graphique peut aussi être utilisé pour "nettoyer l'environnement", c'est l'objet du *Reset logique*

Comme l'effacement d'une commande dans un script de preuve, l'effacement d'un objet de l'environnement est généralement historique. C'est-à-dire que pour effacer un objet il faut effacer tous les objets qui ont été définis après lui. Nous nous proposons d'utiliser la structure de graphe de l'environnement pour fournir un outil de nettoyage moins brutal permettant d'effacer un objet et tous ceux qui en dépendent sans toucher au reste de l'environnement.

Dans l'implémementation pour *Coq*, les choses se compliquent un peu à cause du mécanisme de section. Il est en effet impossible d'effacer un objet d'une section qui a été fermée, et on est obligé d'effacer toute la section.

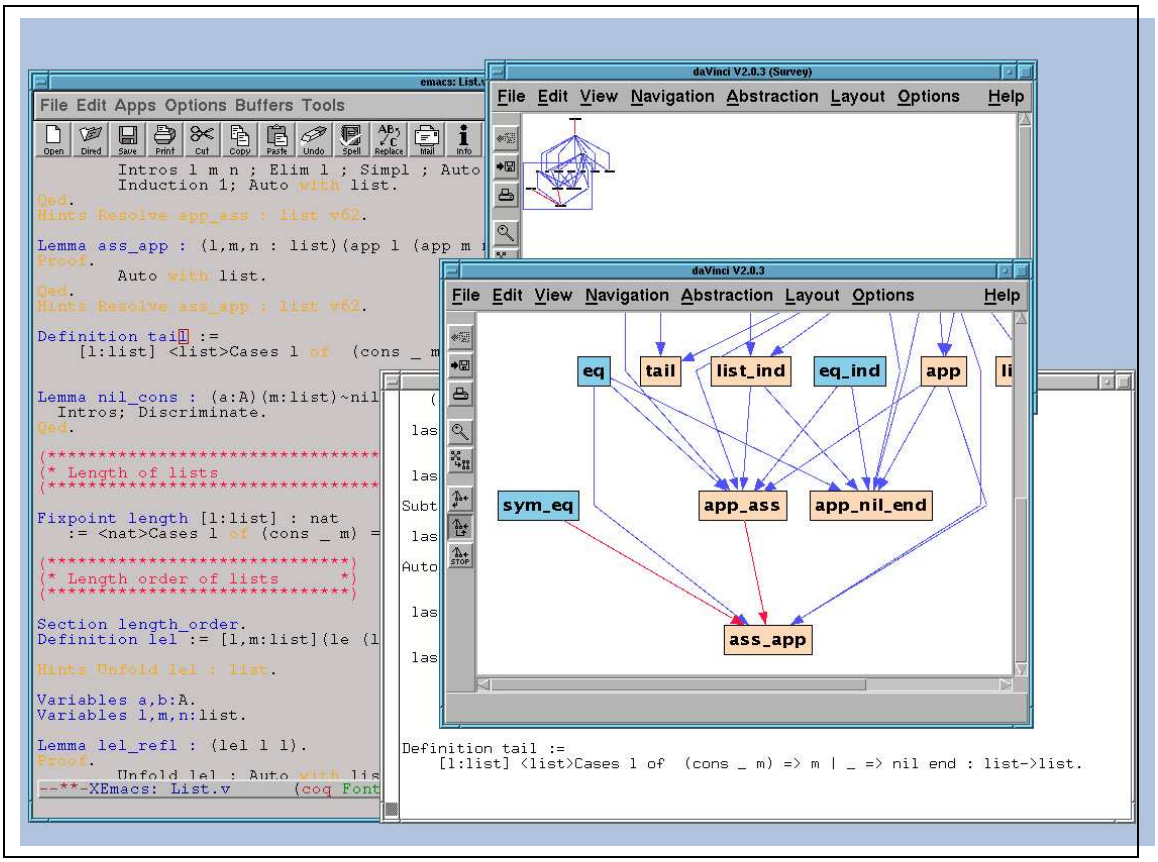


FIG. B.2 – *L'interface Coq/daVinci*

B.2.4 Conclusion

Notre paquetage n'est qu'un prototype minimal mais il offre déjà la possibilité de développer rapidement en *Ocaml* des prototypes d'interface nécessitant la visualisation de graphe.

Annexe C

Exemple d'utilisation de la tactique Similar

On donne ici un exemple trivial d'utilisation de la tactique `Similar`:

```
Coq < Variable A:Prop.
Coq < Lemma factor: A\/A->A/\A.
1 subgoal

  ============================
   A\/A->A/\A

factor < Intros.  (*  *)
Elim H. (* 1 *)
1 subgoal

  H : A\/A
  ============================
   A/\A

factor < 2 subgoals

  H : A\/A
  ============================
   A->A/\A

subgoal 2 is:
 A->A/\A

factor < Split. (* 11 *)
```

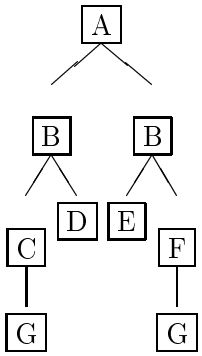
```
3 subgoals

  H : A\/A
  H0 : A
  ============================
   A

subgoal 2 is:
 A
subgoal 3 is:
 A->A/\A

factor < Idtac;Exact H0. (* 111 *)
2 subgoals

  H : A\/A
  H0 : A
  ============================
   A

subgoal 2 is:
 A->A/\A

factor < Exact H0. (* 112 *)
1 subgoal

  H : A\/A
  ============================
   A->A/\A

factor <Similar.
Subtree proved!
```

Annexe D

Preuves et exemples divers

D.1 Exemple d'utilisation de l'algorithme d'expansion

Soit le script `A;B;[C|D|E|F];G` dont l'arbre suivant donnne une version expanser:



Au départ, `pile-d-arguments` contient `A,B,(parallel C D E F),G`, `rang`, `indexet` `k` valent 1 et les piles `P1` et `P2` sont vide.

Nous commençons par dépiler `A` de `pile-d-arguments` et **on le joue**, cela produit deux sous-buts. La pile d'arguments, qui contient alors `B;(parallel C D E F);G`, n'est pas vide et son sommet n'est pas de la forme `(parallel...)`. On fait donc deux copies de pile-d-arguments qui sont empilées dans `P1`. Nous dépilons la première copie que nous allons expanser. Nous assignons donc la valeur dépilée à `pile-d-arguments` puis on dépile `B` de `pile-d-arguments` et **on le joue**, ce qui produit deux nouveaux sous-buts. `pile-d-arguments` contient maintenant `(parallel C D E F),G` et son sommet est donc de la forme `(parallel...)`. `P2` est vide aussi dépilons nous `(parallel C D E F)` de `pile-d-arguments` et empilons nous `F;G`, `E;G`, `D;G` puis `C;G` dans `P2`. On affecte alors à

index la valeur (1 + nombre de but généré par la dernière commande jouée) soit 3 puis nous dépilons `(B;(parallel C D E G);G)` de `P1` pour l'affecter à `pile-d-arguments`.

Nous dépilons alors `B` de `pile-d-arguments` pour **le jouer avec l'index 3**. Il vient deux nouveaux sous-buts et `pile-d-arguments` contient maintenant `(parallel C D E G);G)`. Ainsi le sommet de la pile a la forme `(parallel...)` mais cette fois `P2` n'est pas vide.

Comme `P1` est vide on recopie `P2` dans `P1` et on vide `P2`. Enfin On restaure l'index de départ (1).

Nous dépilons maintenant `c;g` de `P1`. La valeur dépilée est affecté a `pile-d-arguments` puis nous dépilons `C` de `pile-d-arguments` et **nous le jouons**. Il vient un sous-but. Nous dépilons ensuite `G` de `pile-d-arguments` et **nous le jouons**. Cela ne produit pas de sous-but et `pile-d-arguments` est désormais vide. On dépile donc `D;G` de `P1` pour affecter la valeur dépilée à `pile-d-arguments`. Puis on dépile `D` de `pile-d-arguments` et `on le joue`. Comme cela ne génère pas de sous-but on ne s'intéresse pas à la valeur suivante contenue dans `pile-d-arguments` mais on dépile `E;G` de P1 et on l'affecte à `pile-d-arguments`. Nous dépilons alors `E` de `pile-d-arguments` et **nous le jouons**, ce qui ne produit pas non plus de sous-but. On dépile donc `F;G` de `P1`, pour l'affecter à `pile-d-arguments`. Nous dépilons alors `F` de `pile-d-arguments` et **nous le jouons**. Cela produit un sous-but. Enfin nous dépilons `G` et **nous le jouons**. `pile-d-arguments`, `P1` et `P2` sont vide nous avons terminé l'expansion. Le script obtenu est le suivant:

```
1:A
1:B
3:B
1:C
1:G
1:D
1:E
1:F
1:G
```

D.2 Proprietés de la fonction rg'

Il sagit de montrer que la fonction rg' définie à partir d'une fonction de rang rg par: $rg' = \lambda c_1.if\ c \prec_p c_1$

$$then\ Card(\mathcal{T}') - Card(\{c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) > rg(c_1)\}$$

$$else\ rg(c_1) - Card(c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) < rg(c_1)\}$$

Est bien une fonction de rang, c'est-à-dire qu'elle vérifie bien les propriétés suivantes:

1^o rg' est bijective
2^o $c \prec_p c' \Rightarrow rg'(c) < rg'(c')$

Les deux démonstrations sont des études de cas assez semblable dans cette section on montre simplement que rg' est bijective

rg' est bijective Comme rg' est comme rg une fonction de $\mathcal{T}'$ vers $[1..Card(\mathcal{T}')]$, c'est-à-dire que le domaines et le codomaine ont un même cardinal fini, pour monter qu'elle est bijective il suffit de montrer qu'elle est invective.

On suppose donc (H1) $rg'(x_1) = rg'(x_2)$ et on va montrer qu'alors $x_1 = x_2$.

- Si $rg(x_1) = rg(x_2)$ alors $x_1 = x_2$ puisque rg est bijective et la démonstration est terminée.
- Supposons donc que $rg(x_1) \neq rg(x_2)$ on on va montrer que l'on arrive à une contradiction:

Distingons 3 cas:

1^o Si $c \prec_p x_1 \wedge c \prec_p x_2$
On a $rg(c) < rg(x_1) \wedge rg(c) < rg(x_2)$.
Sans perdre de généralité on peut supposer $rg(x_1) < rg(x_2)$ et ordonner l'ensemble

$$\{c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) > rg(x_2)\}$$

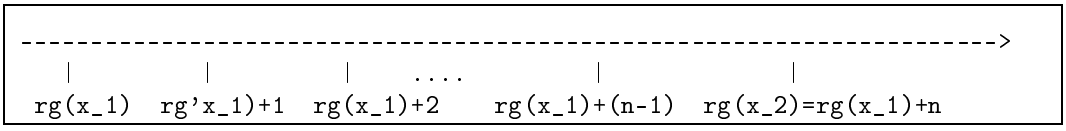
Trivialement x_1 appartient à cet ensemble mais n'appartient pas à l'ensemble

$$\{c_i \in \mathcal{T}' | c \prec_p c_i \wedge rg(c_i) > rg(x_1)\}$$

et donc $rg'(x_1) \neq rg'(x_2)$ ce qui contredit l'hypothèse (H1).

2^o Si $c \not\prec_p x_1 \wedge c \not\prec_p x_2$ On a

$rg(x_1) - Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) < rg(x_1)\}) =$
$rg(x_2) - Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) < rg(x_2)\})$ (1)
Sans perdre de généralité on peut supposer $rg(x_1) < rg(x_2)$
Ainsi $rg(x_2) = rg(x_1) + n$ avec $n \geq 1$ et $n \in N$.
De (1) il vient, $rg(x_2) - rg(x_1) =$
$Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) < rg(x_2)\}) -$
$Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(x_1) \leq rg(d_i) < rg(x_2)\})$
d'où $n = Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(x_1) \leq rg(d_i) < rg(x_2)\})$ (2)



Ainsi

$$A = (\{d_i \in \mathcal{T}' | rg(x_1) \leq rg(d_i) < rg(x_2)\}) \quad (3)$$

et $Card(A) = n$
mais en ajoutant la condition $c \prec_p d_i$ dans la définition de A, on supprime x (car $c \not\prec_p x$) donc

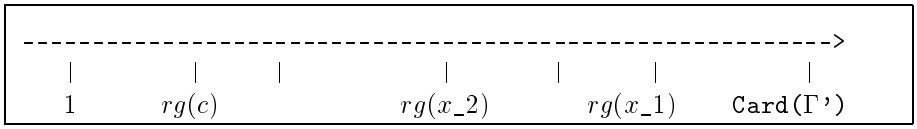
$$Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(x_1) \leq rg(d_i) < rg(x_2)\}) \leq n - 1$$

et par (2) on a $n \leq n - 1$ ce qui est absurde.

3^o Si $c \not\prec_p x_1 \wedge c \prec_p x_2$
On a
$rg(x_1) - Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) > rg(x_1)\}) =$
$Card(\mathcal{T}') - Card(\{d_i \in \mathcal{T}', rg(d_i) > rg(x_2)\})$ (1)

– Si $rg(x_1) > rg(x_2)$

On pose $rg(x_1) = rg(x_2) + n \quad (1 \leq n)$
$Card(\mathcal{T}' - rg(x_1) = Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) > rg(x_2)\}) -$
$Card(\{i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) < rg(x_1)\})$ (1)
On a:



On note

$$A_y = Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) > rg(x_2)\})$$

et

$$A_y = Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) < rg(x_1)\})$$

or $A_y - A_x = Card(\{i \in \mathcal{T}' | c \prec_p d_i, rg(x_1) \leq rg(d_i)\}) -$
$Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) \leq rg(x_2)\})$
On note

$$B_x = Card(\{i \in \mathcal{T}' | c \prec_p d_i, rg(x_1) \leq rg(d_i)\})$$

et

$$B_y = Card(\{d_i \in \mathcal{T}' | c \prec_p d_i, rg(d_i) \leq rg(x_2)\})$$

On a

$$O \leq B_x \leq Card(\mathcal{T}') - rg(x_1)$$

$$1 \leq B_y \leq rg(x_2) - rg(c)$$

$$-rg(x_2) + rg(c) \leq -B_y \leq -1$$

ainsi

$$-rg(x_2) + rg(c) \leq B_x - B_y \leq Card(\mathcal{T}') - rg(x_1) - 1$$

or

$$Card(\mathcal{T}') - rg(x_1) = A_x - A_y = B_x - B_y$$

d'où

$$Card(\mathcal{T}') - rg(x_1) \leq Card(\mathcal{T}') - rg(x_1) - 1$$

ce qui est absurde.

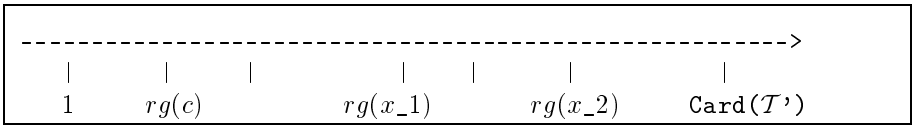
– $\boxed{\text{si } rg(x_1) < rg(x_2)}$

On a

$$rg(x_2) = rg(x_1) + n \quad (1 \leq n)$$

Comme $c \prec_p x_2$ on a $rg(c) < rg(x_2)$ Il y a donc deux sous-cas selon les positions respectives de $rg(c)$ et $rg(x_1)$.

– $\boxed{rg(c) < rg(x_1)}$



En reprenant les notations A_x et A_y introduite pour le cas précédent, on a toujours

$$Card(\mathcal{T}') - rg(x_1) = A_y - A_x$$

$$0 \leq A_y \leq Card(\mathcal{T}') - rg(x_2)$$

$$0 \leq A_x$$

$$-A_x \leq 0$$

$$A_x - A_y \leq Card(\mathcal{T}') - rg(x_2)$$

d'où

$$Card(\mathcal{T}') - rg(x_1) \leq Card(\mathcal{T}') - rg(x_1) - n$$

ce qui entraîne

$$0 \leq -n$$

et donc

$$n \leq 0$$

ce qui est impossible.

– $\boxed{rg(x_1) < rg(c)}$

On a toujours

$$Card(\mathcal{T}') - rg(x_1) = A_y - A_x$$

$$0 \leq A_y \leq Card(\mathcal{T}') - rg(x_2)$$

$$A_x = 0$$

on a donc

$$Card(\mathcal{T}') - rg(x_1) \leq Card(\mathcal{T}') - rg(x_2)$$

ce qui entraîne

$$n \leq 0$$

ce qui est impossible.

◇

Bibliographie

[1] Jean-Raymond Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] Sten Agerholm, Ilya Beylin, et Peter Dybjer. « A Comparison of HOL and Alf Formalizations of a Categorical Coherence Theorem ». Dans J. von Wright, J. Grundy, et J. Harrison, éditeurs, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 de *Lecture Notes in Computer Science*, pages 17–32, Murray Hill, NJ, USA, août 1996. Springer.

[3] Alfred V. Aho, John E. Hopcroft, et Jeffrey D. Ullman. *Structures de données et algorithmes.* InterEditions,Paris, 1989.

[4] Penny Anderson. « *Program Derivation by Proof Transformation* ». PhD thesis, Carnegie Mellon University, octobre 1993. Egalement paru comme Rapport Technique CMU-CS-93-206.

[5] Penny Anderson. « Representing Proof Transformations for Program Optimization ». Dans *Proceedings of the 12th International Conference on Automated Deduction*, pages 575–589, Nancy, France, juin 1994. Springer-Verlag LNAI 814.

[6] Andrew W. Appel et David B. MacQueen. « Standard ML of New Jersey ». Dans Martin Wirsing, éditeur, *Third International Symposium on Programming Language Implementatio n and Logic Programming*, New York, août 1991. Springer-Verlag.

[7] David R. Aspinall. « Isabelle Modules: a New Theory Mechanism for Isabelle ». Undergraduate dissertation, University of Cambridge, 1991.

[8] David R. Aspinall, Healfdene Goguen, Thomas Kleymann, et Dilip Sequeira. « Proof General 2.1 », mars 1999. Accessible à l'URL: http://www.dcs.ed.ac.uk/home/proofgen/ProofGeneral/doc/ProofGeneral_toc.html.

[9] Owen Astrachan. « METEOR: Exploring Model Elimination Theorem Proving ». *Journal of Automated Reasoning*, 13:283–296, 1994.

[10] Bernhard Beckert et Joachim Posegga. « leanTAP: Lean Tableau-Based Theorem Proving. Extended Abstract ». Dans A. Bundy, éditeur, *Proceedings, 12th International Conference on Automated Deduction (CADE), Nancy, France*, LNCS 814, pages 793–797. Springer, 1994.

[11] Bernhard Beckert et Joachim Posegga. « leanTAP: Lean Tableau-based Deduction ». *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[12] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, et Francis Montagnac. « *User Guide to the CtCoq Proof Environment* ». INRIA, février 1996.

[13] Yves Bertot. « Direct Manipulation of Algebraic Formulae in Interactive Proof Systems ». Dans *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, septembre 1997.

[14] Yves Bertot. « A proved compiler for imperative language », 1998. (version préliminaire).

[15] Yves Bertot, Gilles Kahn, et Laurent Théry. « Proof by Pointing ». Dans *International Symposium on Theoretical Aspects of Computer Science*, 1994.

[16] Yves Bertot, Thomas Schreiber, et Dilip Sequeira. « Proof by pointing in a weakly structured context ». Accessible par FTP: ftp://ftp.dcs.ed.ac.uk/pub/lego/pbp/main.ps.

[17] Yves Bertot et Laurent Théry. « A Generic Approach to Building User Interfaces for Theorem Provers ». *Journal of Symbolic Computation*, 22, 1998.

[18] Paul E. Black et Phillip J. Windley. « Automatically Synthesized Term Denotation Predicates: A Proof Aid ». Dans Schubert et al. [120], pages 46–57.

[19] Urban Boquist et Jan Sparud. « SML-Make: Incremental software development in Standard ML under GNU Emacs ». Masters thesis, Department of Computer Science, Chalmers University of Technology, juin 1991.

[20] Richard Bornat et Bernard Sufrin. « Jape — A Framework for building Interactive Proof Editors ». Accessible à l'URL: http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.shtml.

[21] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, et Valérie Pascual. « Centaur: the system ». *Software Engineering Notes*, 13(5), 1988. Third Symposium on Software Development Environments. Egalement paru comme Rapport Technique INRIA no. 777.

[22] Richard J. Boulton. « *Efficiency in a Fully-Expansive Theorem Prover* ». Rapport Technique number 337, University of Cambridge Computer Laboratory, mai 1994.

[23] Robert S. Boyer et J S. Moore. *A Computational Logic.* Academic Press, New York, NY, 1979.

[24] Robert S. Boyer et J S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[25] Robert Constable et Douglas Howe. NuPrl as a General Logic. Dans P. Odifreddi, éditeur, *Logic and Computation.* Academic Press, 1990.

[26] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, Walter R. Cleaveland, James F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, et Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[27] Yann Coscoy, Gilles Kahn, et Laurent Théry. « Extracting Text from Proof ». Dans M. Dezani et G. Plotkin, éditeurs, *Proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902. Springer-Verlag LNCS, avril 1995.

[28] Judicaël Courant. « A Module Calculus for Pure Type Systems ». Dans R. Hindley, éditeur, *Proceedings fo the Third International Conference on Typed Lambda*

Calculus and Applications (TLCA'97), Nancy, France, avril 1997. Springer-Verlag LNCS.

[29] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, et Mandayam Srivas. « A Tutorial Introduction to PVS ». Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, avril 1995.

[30] Paul Curzon. « The Importance of Proof Maintenance and Reengineering ». Dans Jim Alves-Foss, éditeur, *International Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track: Short Presentations*, pages 17–32, 1995.

[31] Paul Curzon. « Tracking design changes with formal machine-checked proof ». *j-COMP-J*, 38(2):91–100, 1995.

[32] Paul Curzon. « Virtual Theories ». Dans E. Thomas Schubert, Phillip J. Windley, et James Alves-Foss, éditeurs, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 de *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1995.

[33] Nikolas G. de Bruijn. « The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions ». Dans M. Laudet, éditeur, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, décembre 1968. Springer-Verlag LNM 125.

[34] Nikolas G. de Bruijn. A Survey of the Project AUTOMATH. Dans J.P. Seldin et J.R. Hindley, éditeurs, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[35] Anne-Marie Déry et Laurence Rideau. « *Distributed programming environments: an example of a message protocol* ». Rapport Technique Inria no. 165, 1994.

[36] K. A. Eastaughffe. « Support for Interactive Theorem Proving: Some Design Principles and Their Application ». Dans *Proceedings of UITP'98, User Interfaces for Theorems Provers Workshop, Eindhoven*, 1998.

[37] K. A. Eastaughffe et M. A. Ozols. « A Proof Tree Package for Isabelle ». http://www.cl.cam.ac.uk/users/kae24/.

[38] William M. Farmer, Joshua D. Guttman, Mark E. Nadel, et F. Javier Thayer. « Proof Script Pragmatics in imps ». Dans A. Bundy, éditeur, *Automated Deduction—CADE-12*, volume 814 de *Lecture Notes in Computer Science*, pages 356–370. Springer-Verlag, 1994.

[39] William M. Farmer, Joshua D. Guttman, et F. Javier Thayer. « imps: An Interactive Mathematical Proof System (system abstract) ». Dans M. E. Stickel, éditeur, *10th International Conference on Automated Deduction*, volume 449 de *Lecture Notes in Computer Science*, pages 653–654. Springer-Verlag, 1990.

[40] William M. Farmer, Joshua D. Guttman, et F. Javier Thayer. « imps: System Description ». Dans D. Kapur, éditeur, *Automated Deduction—CADE-11*, volume 607 de *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[41] William M. Farmer, Joshua D. Guttman, et F. Javier Thayer. « Little Theories ». Dans D. Kapur, éditeur, *Automated Deduction—CADE-11*, volume 607 de *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[42] William M. Farmer, Joshua D. Guttman, et F. Javier Thayer. « imps: An Interactive Mathematical Proof System ». *Journal of Automated Reasoning*, 11:213–248, 1993.

[43] William M. Farmer, Joshua D. Guttman, et F. Javier Thayer. « The imps User's Manual ». Rapport Technique M-93B138, The mitre Corporation, 202 Burlington Road, Bedford, MA 01730-1420, USA, novembre 1993.

[44] Stuart I. Feldman. « Make—A Program for Maintaining Computer Programs ». *spe*, 9(4):255–65, avril 1979.

[45] Amy Felty. « *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language* ». PhD thesis, University of Pennsylvania, 1989. (Accessible comme Rapport Technique MS-CIS-89-53).

[46] Amy Felty. « Implementing tactics and tacticals in a higher-order logic programming language ». *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[47] Amy Felty et Douglas Howe. « Generalization and Reuse of Tactic Proofs ». Dans Frank Pfenning, éditeur, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 1–15, Kiev, Ukraine, juillet 1994. Springer-Verlag LNAI 822.

[48] Amy Felty et Douglas Howe. « Tactic Theorem Proving with Refinement-Tree Proofs and Metavariables ». Dans Alan Bundy, éditeur, *Proceedings of the 12th International Conference on Automated Deduction*, pages 605–619, Nancy, France, juin 1994. Springer-Verlag LNAI 596.

[49] Amy Felty et Douglas J. Howe. « Generalization and Reuse of Tactic Proofs ». *Lecture Notes in Computer Science*, 822:1–15, 1994.

[50] Amy Felty et Douglas J. Howe. « Tactic theorem proving with refinement-tree proofs and metavariables ». *Lecture Notes in Computer Science*, 814:605–619, 1994.

[51] Michael Fröhlich. « *Incremental Graphlayout in the Visualization System daVinci (in german language)* ». PhD thesis, Department of Computer Science; University of Bremen, novembre 1997.

[52] Michael Fröhlich et Mattias Werner. « daVinci V2.0 Online Documentation », 1996. Accessible à l'URL: http://www.informatik.uni-bremen.de/davinci/doc_V2.0.

[53] Harald Ganzinger, Robert Nieuwenhuis, et Pilar Nivela. « The Saturate System ». ftp://ftp.mpi-sb.mpg.de/pub/SATURATE/doc/Saturate/Saturate.html.

[54] Christoph Goller, Reinhold Letz, Klaus Mayr, et Johannes Schumann. « SETHEO V3.2: Recent Developments – System Abstract – ». Dans Alan Bundy, éditeur, *Automated Deduction – CADE-12*, volume 814 de *Lecture Notes in Artificial Intelligence*, pages 778–782, Nancy, France, juin/juillet 1994. Springer.

[55] Michael J.C. Gordon et Thomas.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[56] Michael J.C. Gordon, Robin Milner, et Christopher P. Wadsworth. *Edinburgh LCF.* Numéro 78 dans Lecture Notes in Computer Science. Springer Verlag, 1979.

[57] David Griffioen et Marieke Huisman. « A Comparison of PVS and Isabelle/HOL ». Dans Jim Grundy et Malcolm Newey, éditeurs, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 de *Lecture Notes in Computer Science*, pages 123–142, Canberra, Australia, septembre 1998. Springer-Verlag.

[58] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, et J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and

Monographs in Computer Science. Springer-Verlag, 1993. ISBN 0-387-94006-5/ISBN 3-540-94006-5.

[59] John Harrison. « Formalized Mathematics ». Rapport Technique 36, Turku Centre for Computer Science (TUCS), Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland, 1996. Accessible à l'URL: http://www.cl.cam.ac.uk/users/jrh/papers/form-math3.html.

[60] Robert W. Hasker et Uday S. Reddy. « Generalization at Higher Types ». Dans D. Miller, éditeur, *Proceedings of the Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, Pennsylvania, juillet 1992. University of Pennsylvania. Accessible comme Rapport Technique MS-CIS-92-86.

[61] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, et Jörg Siekmann. « Ω-MKRP: A proof development environment ». Dans Alan Bundy, éditeur, *Automated Deduction — CADE-12*, Proceedings of the 12th International Conference on Automated Deduction, pages 788–792, Nancy, France, 1994. Springer-Verlag, Berlin, Germany. LNAI 814.

[62] Ian Jacobs et Janet Bertot, éditeurs. « *Centaur 1.2* », Chapitre The PPML Manual. Inria Sophia–Antipolis, 1993.

[63] Ian Jacobs et Janet Bertot, éditeurs. « *Centaur 1.2* », Chapitre The Metal Manual. Inria Sophia-Antipolis, 1993.

[64] Ian Jacobs et Janet Bertot, éditeurs. « *Centaur 1.2* », Chapitre The Virtual Tree Processor. Inria Sophia-Antipolis, 1993.

[65] Ian Jacobs et Laurence Rideau. « A Centaur Tutorial ». Rapport Technique INRIA no 140, juillet 1992.

[66] Florian Kammüller. « Comparison of IMPS, PVS and Larch with respect to theory treatment and modularization ». http://www.cl.cam.ac.uk/users/fk203/papers.

[67] Thomas Kolbe et Christoph Walther. « Reusing Proofs ». Dans *Proc. of the 11th ECAI*, pages 80–84, Amsterdam, The Netherlands, 1994.

[68] Arnaud Le Hors. « Graph: A Directed Graph Displaying Server, GIPE 2 ESPRIT project, 4th Review Report, Workpackage 4 », 1992.

[69] Projet LEMME. « PCOQ ». Communication personnelle.

[70] Xavier Leroy. « Programmation du système Unix en Caml Light ». Rapport Technique INRIA no. 147, 1992.

[71] Xavier Leroy. « The Caml Light system (release 0.7) ». Projet Cristal, INRIA, 1995.

[72] Reinhold Letz, Johann M. P. Schumann, Stephan Bayerl, et Wolfgang Bibel. « SETHEO: A High-Performance Theorem Prover ». *Journal of Automated Reasoning*, 8:183–212, 1992.

[73] Panagiotis K. Linos, Philippe Aubet, Laurent Dumas, Yann Helleboid, Patricia Lejeune, et Philippe Tulula. « Visualizing Program Dependencies: An Experimental Study ». *Software—Practice and Experience*, 24(4):387–403, avril 1994.

[74] Jean-Michel Léon. « Displaying graphs with Centaur, Bull Research France », Janvier 1993.

[75] Zhaohui Luo. « *An Extended Calculus of Constructions* ». PhD thesis, Department of Computer Science, University of Edinburgh, juin 1990.

[76] Zhaohui Luo et Robert Pollack. « The LEGO Proof Development System: A User's Manual ». Rapport Technique ECS-LFCS-92-211, University of Edinburgh, mai 1992.

[77] Zhaohui Luo, Robert Pollack, et Paul Taylor. « How to use LEGO ». Rapport Technique LFCS-TN-27, University of Edinburgh, Edinburgh, Scotland, octobre 1989.

[78] Donald Mackenzie. « The Automation of Proof: A Historical and Sociological Exploration ». *IEEE Annals of the History of Computing*, 17(3):7–29, 1995.

[79] Lena Magnusson. « Refinement and local undo in the interactive proof editor ALF ». Dans *Proceedings of the Workshop on Types for Proofs and Programs*, pages 191–208, Nijmegen, The Netherlands, 1993.

[80] Lena Magnusson. « *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution* ». PhD thesis, Chalmers University of Technology and Göteborg University, janvier 1995.

[81] Lena Magnusson et Bengt Nordström. The ALF Proof Editor and Its Proof Engine. Dans Henk Barendregt et Tobias Nipkow, éditeurs, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[82] Ursula Martin et Jeannette M. Wing, éditeurs. *First International Workshop on Larch, Dedham 1992*, Workshops in Computing. Springer-Verlag, 1993. ISBN 3-540-19804-0/ISBN 0-387-19804-0.

[83] Per Martin-Löf. Constructive Mathematics and Computer Programming. Dans J. L. Cohen, J. Łoś, H. Pfeiffer, et K.-D. Podewski, éditeurs, *Proceedings 6th Intl. Congress on Logic, Methodology and Philosophy of Science, Hannover, FRG, 22–29 août 1979*, volume 104 de *slfm*, pages 153–175. North Holland, Amsterdam, 1982.

[84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 de *Studies in Proof Theory: Lecture Notes*. Bibliopolis, Napoli, 1984.

[85] William W McCune. « *otter 3.0 Reference manual and guide* ». Argonne national laboratory, 9700 south cass avenue Argonne, Illinois 60439-4801, janvier 1997.

[86] Nicholas Merriam et Michael Harrisson. « What is Wrong with GUIs for Theorem Provers? ». Dans *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997. Accessible à l'URL: http://www.inria.fr-/croap/events/uitp97-papers.html.

[87] Robin Milner, Mads Tofte, Robert Harper, et Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[88] Allen Newell et J. C. Shaw. « Programming the logic theory machine ». Dans *Proceedings of the 1957 Western Joint Computer Conference*, pages 230–240. IRE, 1957.

[89] Allen Newell, J. C. Shaw, et H. A. Simon. « Empirical explorations with the logic theory machine ». *Proceedings of the Western Joint Computer Conference*, 15:218–239, 1957.

[90] Tobias Nipkow et David von Oheimb. « Java$_{\ell ight}$ is Type-Safe — Definitely ». Dans *Proc. 25th ACM Symp. Principles of Programming Languages*, pages p. 161–170. ACM Press, New York, 1998.

[91] John K. Ouserhout. *Tcl and the Tk Toolkit.* Professional Computing series. Addison-Wesley, 1994.

[92] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, et M.K. Srivas. « PVS: Combining Specification, Proof Checking, and Model Checking ». Dans Rajeev Alur et Thomas A. Henzinger, éditeurs, *Computer-Aided Verification, CAV '96*, volume 1102 de *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, juillet/août 1996. Springer-Verlag.

[93] Sam Owre, John M. Rushby, et Natarajan Shankar. « PVS: A Prototype Verification System ». Dans Deepak Kapur, éditeur, *11th International Conference on Automated Deduction (CADE)*, volume 607 de *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, juin 1992. Springer-Verlag.

[94] Maris A. Ozols, Katherine A. Eastaughffe, et Tony Cant. « XIsabelle: A System Description ». Dans William McCune, éditeur, *Proceedings of CADE-14 (14th International Conference on Automated Deduction)*, volume 1249 de *Lecture Notes in Computer Science*, pages 366–389, Townsville, Australia, juillet 1997. Springer-Verlag.

[95] Christine Paulin-Mohring. *Extraction de programmes dans le calcul des construction.* These de Doctorat, Université Paris 7, 1989.

[96] Christine Paulin-Mohring. Inductive Définition in the Système Coq - Rules and Properties. Dans *Typed Lambda Calculi and Application - également paru en rapport de recherche 92-49.* LIP-ENS Lyon, Décembre 1992.

[97] Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. Dans P. Odifreddi, éditeur, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[98] Lawrence C. Paulson. « Introduction to Isabelle ». Rapport Technique 280, University of Cambridge, Computer Laboratory, 1993.

[99] Lawrence C. Paulson. « The Isabelle Reference Manual ». Rapport Technique 283, University of Cambridge, Computer Laboratory, 1993.

[100] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828, 1994.

[101] Lawrence C. Paulson. *ML for the Working Programmer (2nd Edition).* Cambridge University Press, 1996.

[102] Lawrence C. Paulson et Tobias Nipkow. « Isabelle Tutorial and User's Manual ». Rapport Technique 189, University of Cambridge, Computer Laboratory, janvier 1990.

[103] Frank Pfenning. « Logic Programming in the LF Logical Framework ». Dans Gérard Huet et Gordon Plotkin, éditeurs, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[104] Frank Pfenning. « Unification and Anti-Unification in the Calculus of Constructions ». Dans *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, juillet 1991.

[105] Gordon D. Plotkin. « A note on inductive generalization ». Dans B. Meltzer et D. Michie, éditeurs, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.

[106] Robert Pollack. « *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions* ». PhD thesis, University of Edinburgh, 1994.

[107] Olivier Pons. « Undoing and Managing a proof ». Dans *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997. Accessible à l'URL: http://www.inria.fr/croap/events/uitp97-papers.html.

[108] Olivier Pons, Yves Bertot, et Laurence Rideau. « Notions of dependency in proof assistants ». Dans *Electronic Proceedings of "User Interfaces for Theorem Provers 1998"*, Sophia-Antipolis, France, 1998. Accessible à l'URL: `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[109] Loic Pottier. Accessible à l'URL: http://www.inria.fr/croap/personnel/Loic.Pottier/Coq/MATH/index.html.

[110] Loic Pottier et Olivier Pons. « dependtohtml:Creating hypertext graphical representation of directed graphs », 1998. Accessible à l'URL: http://www.inria.fr/croap/personnel/Loic.Pottier/Dependtohtml/README.html.

[111] Ralf Reetz. « Improving the usability of higher order logic theorem prover by graph visualisation ».

[112] Wolfgang Reif. The KIV System: Systematic Construction of Verified Software. Dans D. Kapur, éditeur, *Automated Deduction: CADE-11 - Proc. of the 11th International Conference on Automated Deduction*, pages 753–757. Springer, Berlin, Heidelberg, 1992.

[113] Wolfgang Reif. The KIV-approach to Software Verification. Dans M. Broy et S. Jähnichen, éditeurs, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[114] Wolfgang Reif et Kurt Stenzel. « Reuse of Proofs in Software Verification ». Rapport Technique, Karlsruhe, 1992.

[115] Wolfgang Reif et Kurt Stenzel. « Reuse of Proofs in Software Verification ». *Foundations of Software Technology and Theoretical Computer Science*, 13, 1993.

[116] Thomas Reps et Tim Teitelbaum. *The Synthesizer Generator: a system for constructing language based editors*. Springer Verlag, 1988. (troisième édition).

[117] John Rushby et David W. J. Stringer-Calvert. « A Less Elementary Tutorial for the PVS Specification and Verification System ». Rapport Technique SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, juin 1995.

[118] Amokrane Saïbi. « Typing algorithm in type theory with inheritance ». Dans *Proc. of Principles of Programming Languages*, Paris, France, janvier 1997. ACM Press.

[119] Amokrane Saïbi. « *Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories* ». Thèse de Doctorat, Université Paris 6, 1999.

[120] E. Thomas Schubert, Phillip J. Windley, et James Alves-Foss, éditeurs. *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 de *Lecture Notes in Computer Science*, Aspen Grove, UT, USA, septembre 1995. Springer-Verlag.

[121] Tom Schubert et John Biggs. « A Tree-Based, Graphical Interface for Large Proof Development ». Dans Thomas F. Melham et Juanito Camilleri, éditeurs, *Proceedings of the 7th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'94)*, septembre 1994.

[122] Johann Schumann et Ortrun Ibens. « *SETHEO V3.3 Reference Manual (Draft)* ». Institut für Informatik, TU München, 1997.

[123] Natarajan Shankar, Sam Owre, et John M. Rushby. « *PVS Tutorial* ». Computer Science Laboratory, SRI International, Menlo Park, CA, février 1993.

[124] Jörg Siekman, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, et Volker Sorge. « A Distributed Graphical User Interface for the Interactive Proof System ». Dans *Proceedings of the International Workshop "User Interfaces for Theorem Provers 1998 (UITP'98)"*, Eindhoven, Netherlands, 1998.

[125] Michael Siff et Thomas Reps. « Program Generalization for Software Reuse: From C to C++ ». Dans *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21_6 de *ACM Software Engineering Notes*, pages 135–146, New York, octobre 16–18 1996. ACM Press.

[126] Konrad Slind. « An Implementation of Higher Order Logic ». Rapport Technique 91-419-03, Computer Science Department, University of Calgary, janvier 1991.

[127] Konrad Slind. « *HOL98 User's Manual* ». Cambridge University Computer Laboratory, avril 1998. Preliminary Draft.

[128] Andrew Stevens. « Toward Mechanised Revision of Proofs and theories ». Dans *Electronic Proceedings of "User Interfaces for Theorem Provers 1996"*, University of York, 1996.

[129] Kozo Sugiyama et Kazuo Misue. « Visualization of Structural Information: Automatic Drawing of Compound Digraphs ». *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21(4):876 – 892, juillet/août 1991.

[130] Kozo Sugiyama, Shojiro Tagawa, et Mitsuhiko Toda. « Methods for Visual Understanding of Hierarchical System Structures ». *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109 – 125, juillet/août 1981.

[131] Christian B. Suttner et Geoff Sutcliffe. « The design of the CADE-13 ATP system competition ». *Lecture Notes in Computer Science*, 1104:146–??, 1996.

[132] Donald Syme. « A New Interface for HOL – Ideas, Issues and Implementaion ». Dans E. T. Schubert, P. J. Windley, et J. Alves-Foss, éditeurs, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 de *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[133] Tanel Tammet. « *Gandalf* ». Department of Computing Science, University of Goteborg / Chalmers University of Technology, S-412 96 Goteborg Sweden, octobre 1997.

[134] Laurent Théry. « A proof development system for the HOL theorem prover ». Dans *Higher Order Logic theorem proving and its applications*, volume 780 de *Lecture Notes in Computer Science*. Springer-Verlag, août 1993.

[135] Laurent Théry, Yves Bertot, et Gilles Kahn. « Real Theorem Provers Deserve Real User-Interfaces ». *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.

[136] Laurent Théry. « Proving and Computing: a certified version of the Buchberger's algorithm ». Rapport technique INRIA no. 3275, octobre 1997.

[137] Mark UTTING et Keith WHITWELL. « Ergo User Manual ». Rapport Technique 93-19, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, février 1994.

[138] Pierre VALENTIN. « Legolang: Une interface graphique pour le système de preuve lego ». Rapport de DEA, Université de Nice, 1993.

[139] Jeffrey Scott VITTER. « US&R: a New Framework for Redoing ». *j-IEEE-SOFTWARE*, 1(4):39–52, octobre 1984.

[140] Pierre WEIS et Xavier LEROY. *Le langage Caml*. InterEditions, ISBN 2-7296-0493-6, 1994.

[141] Mark WEISER. « Program slicing ». Dans *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, mars 1981.

[142] Benjamin WERNER. *Une Théorie des Construction Inductives.* Thèse de Doctorat, Université Paris 7, 1994.

[143] Alfred North WHITEHEAD et Bertrand RUSSELL. *Principia Mathematica*. Cambridge University Press, Cambridge, 1910.

[144] Phillip J. WINDLEY. « Using HOL inside EMACS ». Rapport Technique CS-90-01, Department of Computer Science University of Idaho, décembre 90.

[145] Wai WONG. Recording and Checking HOL Proofs. Dans E. T. SCHUBERT, P. J. WINDLEY, et J. ALVES-FOSS, éditeurs, *Higher Order Logic Theorem Proving and Its Applications*, pages 353–368. Springer, Berlin,, 1995.

[146] Wai WONG. « A Proof Checker for HOL ». Rapport Technique 389, University of Cambridge Computer Laboratory, mars 1996.

[147] Stephen S. YAU, James S. COLLOFELLO, et Terry M. MACGREGOR. « Ripple Effect Analysis for Software Maintenance ». Dans *Proc. Second International Computer Software and Applications Conference, (COMPSAC '78)*, pages 60–65, novembre 1978.

[148] Vincent ZAMMIT. « A Comparative Study of Coq and HOL ». Dans Elsa GUNTER et Amy FELTY, éditeurs, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 de *Lecture Notes in Computer Science*, pages 323–337, Murray Hill, NJ, USA, août 1997. Springer.