# Table des matières

# Chapitre 1

# Présentation générale

## 1.1 Introduction

Après les premiers balbutiements des années 70 , les systèmes dits "d'aide à la démonstration" ont eu, ces dernières années un développement rapide. Ainsi, de nombreux systèmes ont vu le jour, citons par exemple LCF, HOL, Coq, Lego, Nupr, Alf, Isabelle ou PVS.

Le système Coq sur lequel nous avons principalement travaillé, a maintenant atteint un niveau qui permet d'envisager la vérification mécanique de preuves relativement complexes. Un exemple récent développé à Sophia-Antipolis a permis de prouver la correction d'un algorithme de calcul formel sur les polynômes. 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 conclusions majeures 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 réuni au même endroit, les propriétés topologiques à un autre endroit, etc. L'organisation des théories peut influer de façon importante sur la vitesse de développement des preuves et leur réutilisabilité, deux facteurs importants pour que le système de preuve soit praticable en milieu industriel.

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 donc 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 porter d'une modification, pour éviter que l'utilisateur préfère construire une théorie bancale plutôt que de faire l'effort de la nettoyer.

Dans cette thèse nous présentons donc un panorama d'outils d'aide au développement et à la maintenance de théories.



Dans une première partie, nous considérerons les problèmes à leurs niveaux "microscopiques", c'est-à-dire que l'on étudiera les problèmes liés à la construction d'une preuve et que l'on présentera quelques outils permettant d'assister l'utilisateur dans cette tâche [Pon97]. Le travail effectué s'articule principalement autour de trois aspects: la navigation dans les preuves dirigées par les buts, le retour en arrière logique, l'isolation de fragments de preuve et lemmes.

Dans une seconde partie nous considérons le niveau "macroscopique" c'est a dire que nous étudions les problèmes liés à la construction, la représentation et la maintenance de théories [PBR98]. On retrouvera ici de nombreuses similarités avec les concepts et les outils de génie logiciel mais ils prendront parfois un 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.

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

## 1.2 De la programmation à la preuve

Les difficultés inhérentes aux gros développements sur ordinateurs sont bien sûr d'abord apparue dans les activités de programmations. 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 notion de bibliothèque, de modularité, de compilation séparé, d'objets ...

2. Concevoir des outils pour manipuler les langages existents: Cela conduit aux environnements de programmation, aux outils d'aide à gestion de version, aux outils de transformation de programmes ...

En fait ces deux démarches sont bien souvent complémentaires car les outils développés dans la seconde optique amélioreront bien souvent le confort d'utilisation des langages conçus dans la première optique.

### 1.2.1 Structuration du code

### 1.2.2 Interfaces et Environnement

Le Synthesizer Generator[RT88] et le système Centaur [INR94, JR92] sont des outils génériques pour créer à partir d'une spécification formelle d'un langage de programmation (syntaxe et sémantique) des environnement 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.

### 1.2.3 Compréhension, analyse, transformation ...

#### 1.2.3.0.1 Compréhension

**1.2.3.0.2 Visualisation** [LAD+94] Visualizing Program Dependencies: An Experimental Study [KS81] Methods for Visual Understanding of Hierarchical System Structures

#### 1.2.3.0.3 Transformation

##### 1.2.3.0.3.1 Analyse de flot de donnée

##### 1.2.3.0.3.2 Trafola, Trlf [Rou]

### 1.2.4 Prouver c'est Programmer

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 [Paulin-Mohring89]. Ainsi, dès lors que les développements de preuve deviennent importants, on est confronté à des problèmes similaires à ceux que l'on rencontre en programmation. Des solutions similaires peuvent donc être envisagées. Ainsi les notions de bibliothèques, de compilation séparée existent ou sont à l'étude dans les divers langages de preuve. L'adaptation des concepts d'environnement de développement aux assistants de preuve, dont Bertot, Thery et Khan [TBK92] ont mis en évidence l'intérêt, a ouvert de nouveaux horizons.



## 1.3 Présentation rapide de quelques systèmes de preuve

Sans revenir aux pionniers que furent Automath[dB68] et LCF[GMW79, ?], nous présentons ci dessous quelques caractéristiques des principaux prouvers que nous avons étudiées (parfois très sommairement). Des travaux de comparaisons ont par ailleurs été mené citons sans prétendre être exhaustif, le travial de Zammit [Zam97] pour Coq et HOL, de Griffioen et Huisman [WG98] pour PVS et Isabelle/HOL. Dans [Kam] F Kammüller a par ailleurs comparé IMPS, PVS et Isabelle en s'interressant surtout à leur traitement de la modularité. Enfin, [ABD96] compare HOL et Alf.

La plupart des systèmes que nous décrivons ci-dessous sont des héritiers de LCF même si ils n'en ont pas tous gardé le style. Ils implémentent les notions de tactique et de tactiquelle[1].

En logique formelle, on peut procéder déductivement (par chaînage avant), c'est a dire en énumérant tous les théorèmes, mais on risque alors d'attendre longtemps avant que cela ne produise le théorème qui nous intéresse. La plupart des systèmes de preuve fonctionnent donc suivant une démarche inductive (par chaînage arrière). On commence par un but, constitué du théorème que l'on veut prouver et on le transforme récursivement en des buts plus simples. Les fonctions qui réduisent un but en un sous-but sont appeler des tactiques. Dans un tel cadre, une preuve complète est un arbre dont les feuilles sont des axiomes ou des théoremes connus. Les tactiquelle sont des operateurs qui permettent de combiner les tactiques.

#### 1.3.0.1 HOL et Isabelle

HOL[GM93] est basé sur la logique (classique) d'ordre supérieure. Le meta langage est ML et les tactique et tactiquelles sont des fonctions ML. Une preuve peut se faire par chaînage avant mais se fait généralement par chaînage arrière.

En HOL, les théories sont des objets de première classes. On commence une session en ouvrant une théorie.

Il n'existe pas d'object preuve bien que depuis [Won95] on puisse sauvegarder une preuve comme une combinaison des huit règles de base (ou d'un sur-ensemble de ces huit règles).

On compte au moins quatre interfaces pour HOL88 et HOL92: Un environement Emacs du Windley [Win90] et dans lequel Curzon implemente la notion de virtual theories[Cur95c].

Un environnement graphique sous X qui permet de représenter graphiquement les arbre de preuves implementé par Schubert et Biggs [SB94]

Un environnement structuré base sur Centaur, CHOL du a Thery [Thé93] qui à été le véritable point de départ des expériences de Bertot Thery et Khan[BT98] sur l'adap-

1. ce néologisme est calqué sur le terme fonctionnelle

tation des concepts, outils et méthodes des environnements de programmation aux environnements de preuve. Enfin Syme a développé un environnement basé sur tclTk[Ous94], tkHoh[Sym95]

Sigmalons aussi que les problèmes de maintenance de preuve en HOL on été discuté par Curzon dans [Cur95a, Cur95b].

Isabelle [Pau90, PN90, Pau93a, Pau93b, Pau94] est un théorème prouveur générique. De nouvelles logiques sont introduite en spécifiant leur syntaxe et leurs règles d'inférence. Les procédures de preuve sont exprimé par des tactique et des tactiquelles. De nombreuse logiques du premier ordre (FOL,ZF,LCF etc.) ou d'ordre supérieure avec en particulier HOL y sont implementées.

Du cotes des interface, on peut cité les experience de Bertot, Thery et Khan utilisant Centaur ou les interface XIsabelle

#### 1.3.0.2 Lego

Le système Lego [LPT89, LP92, Pol94] implémente différents systèmes de type (LF, CC, GCC, UTT) et surtout ECC [Luo90] . Il est très proche de Coq d'abort puisse que ECC et CCI sont très proche mais aussi dans sont mode de fonctionnement. Ainsi la plupart des choses que nous dirons sur coq restent valable en Lego.

Du coté des interfaces, on compte un mode Emacs dans lequel certain outils[BSS] d'abord développé dans le cadre d'un environnement structure comme la preuve par sélection, ont été adaptés. Nous pouvons aussi mentionner une expérimentation[Val93] basé sur Centaur (avec notamment la construction interactive de l'arbre de preuve en utilisant le serveur de graphe [Hor92, Léo93] qui fonctionne avec Centaur)

#### 1.3.0.3 Nuprl

Le système Nuprl[C$^+$86, CH90]est basé sur la théorie des types de Martin Lof (prédicatif et constructif). Il dispose d'une interface graphique et d'un puissant langage de tactique à la LCF.

#### 1.3.0.4 Alf

Le système Alf[MN94, Mag95] est aussi basé sur la théorie des types de Martin Lof avec substitutions explicites. Il se caractérise par le fait que l'on manipule (édite) directement *lambda*-terme à l'aide de la souris dans un environnement graphique basé sur Emacs (on n'a pas de script de preuve à moins d'enregistrer les opérations associées aux clics souris). 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. (## ATTENTION plus LCF like)



#### 1.3.0.5 PVS

PVS[SOR93] est basé sur une logique classique d'ordre supérieure avec fonction, ensemble, enregistrement, n-uplet, sous-typage, type dépendant, et théories paramétrées.

#### 1.3.0.6 IMPS

La logique de IMPS[FGT90, FGT92a, FGT93a, FGT93b, FGNT94] (appelée LUTIN) est une version non constructive de la théorie des types Elle autorise la définition de fonctions partielles et de sous types.

Une des caracteristique du système est la notion de "little theories " [FGT92b].

On a une interface assez conséquente construite sous Emacs.

#### 1.3.0.7 KIV[Rei92, Rei95] (Karlsruhe Interactive Verifier)

Dans la tradition LCF, la construction de la preuve se fait par application de tactiques qui reduisent un but en un nombre donné de sous but. Ce système se caractérise par un haut degré d'automatisation: la sélection des tactiques se fait le plus souvent par heuristique. Lorsque toute les heuristiques sélectionnées échouent l'intervention de l'utilisateur est requise dans le choix des tactiques ou de nouvelles heuristiques, la décision de revenir en arrière ou d'ajouter des lemme intermédiaire.

Le système comporte par ailleurs une interface graphique et une organisation hiérarchique des projets, que l'on peut visualiser en utilisant de logiciel de dessin de graphe daVinci.

#### 1.3.0.8 Miscellaneous

Jape[BS], Eudillos[SMO92] sont avant tout pédagogiques.

Les démonstrateurs conçus par Amy Felty dans son travail de thèse [Fel89, Fel93] implementent la logique intuitioniste du premier ordre dans une version mixte du calcul des séquents et de la déduction naturelle. Implementé en Lambda-Prolog ces travaux ont servi de base à ses expérimentations sur la manipulation de preuve, les problemes de "undoing" et de correction et de reutilisation de preuve [FH94b, FH94a].

#### 1.3.0.9 B

La méthode B[Abr96] développée par J.R. Abrial est basé sur la théorie des ensemble. Elle fonctionne par raffinement de spécifications exprimées sous forme de machines abstraites

Il existe plusieurs implementations de ce système comme l'atelier B. On y dispose d'un prouveur chargé de vérifier les obligations de preuve générées lors des étapes de raffinement.

#### 1.3.0.10 Le Larch Prouveur

Le Larch prouveur LP[GHG+93, ME93]ăest base sur une logique du première ordre "multisorté"

Citons enfin Omega[HKK+94] et Ergo[UW94] dont nous mentionnerons les interfaces.

Parmi ces différents prouveurs nous avons choisi le système Coq pour mener nos expérimentations. Il permet d'allier (ou de concilier ?) les avantages de la théorie des types d'ordre supérieure (en particulier l'existence d'objets preuve sous forme de $\lambda - terme$) et des procédures de preuve a 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 demarche de preuve tout en permettant d'intégrer des composantes de preuve automatique. Ce point de vue hybride fournit des contraintes nouvelles pour certains problèmes (comme le problème du retour arrière logique ou la modification de théorie). Nous en donnons maintenant une présentation plus détaillé.

## 1.4 Remarques générales sur ce document

La variété des applications abordées dans cette thèse s'appuie sur quelques lignes directrices. La notion de dépendance y joue un rôle central.

# Chapitre 2

# Présentation détailléede Coq et de l'environnement CtCoq

### 2.0.1 Présentation de Coq

L'assistant à la démonstration **Coq** est développé depuis 1984 à l'INRIA et depuis 1989 à l'ENS Lyon. 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 le décrivons maintenant en détail.

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*). 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.

**2.0.1.0.1 Terme de preuve versus script de preuve** Une formule (ie. une spécification) est représentée par un type, une preuve de cette formule (ie. le programme qui la réalise) est un $\lambda$-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 $\lambda$-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)
```

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 donnée consiste juste à le "type checker".

Comme on le constate en regardant le terme précédant, une preuve sous forme de *lambda*-terme est assez peu lisible pour le commun des mortels. Un axe de recherche en plein développement [CKT95], consiste à essayer de traduire ce $\lambda$-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 construite par raffinements successifs d'un but initial à l'aide de ce qu'on appelle des *tactiques*. Ces tactiques sont des règles d'inférence du *Calcul des Constructions Inductives*, appliquées de bas en haut, des règles moins primitive ou des programmes fonctionnels. Elle peuvent être composé entre elle par ce que l'on appelle des tactiquelles[1].

Par exemple, la preuve du lemme précédent peut être construite en appliquant successivement les tactiques de la Figure 2.1.

| | |
|---|---|
| 1 | Intros n m p. |
| 2 | Elim n. |
| - | (* cas de base *) |
| 3 | Simpl. |
| 4 | Apply refl_equal. |
| | |
| - | (* Ind *) |
| 5 | Intros. |
| 6 | Simpl. |
| 7 | Rewrite ->H. |
| 8 | Apply refl_equal. |

FIG. 2.1 – *Script de la preuve de l'associativité à gauche de l'addition*

1. ce néologisme est calqué sur le terme fonctionelle

De façon interne le système manipule un $\lambda$-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). Elle représentent les sous termes de preuve restant à construire. A chacunes d'elles est donc associé un sous-but restant à prouver (on parlera aussi de sous-but ouvert). Ici on 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 l'opérateur de récursion 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 representation interne le $\lambda$-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.

La preuve (complète ou non) est donc 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-but 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 plus loin qu'elles peuvent aussi être utilisées au niveau de l'utilisateur notamment pour lui permettent retarder l'instantiation d'un témoin lors de l'élimination d'un quantificateur existentiel.



Dans notre exemple, l'utilisateur est donc 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. 2.2 – *Une étape de construction de la preuve*

**2.0.1.0.2 Structure du script** Nous avons donné figure 2.1 le script "dans l'ordre". Dans ce script en effet, à chaque étapes, c'est le premier but ouvert que propose le système qui est attaqué. Mais on peut construire la preuve dans un ordre quelconque. On précise alors le sous but que sur lequel on veut travailler grâce à son indice liste des sous buts ouverts. On précède la tactique utilisé par cet indice comme cela est illustré sur la figure 2.3

| | |
|---|---|
| 1 | Intros n m p. |
| 2 | Elim n. |
| 3 | 2:Intros. |
| 4 | 1:Simpl. |
| 5 | 2:Simpl |
| 6 | 2:Rewrite ->H. |
| 7 | 1:Apply refl_equal. |
| 8 | 1:Apply refl_equal. |

FIG. 2.3 – *Autre script de preuve de l'associativité à gauche de l'addition*

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

Ainsi les scripts de preuve que nous venons de montrer (figure 2.1 et 2.3) correspondent tous deux à l'arbre de la figure 3

Cette structure arborescente est d'ailleurs également maintenue, le temps de la preuve, 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.

Au cours du développement d'une preuve, la stratégie de démonstration se reflète donc dans la construction d'un arbre de preuve dont les noeuds sont les tactiques que l'on applique. Pour aider à s'orienter dans la structure de la preuve que l'on construit, nous

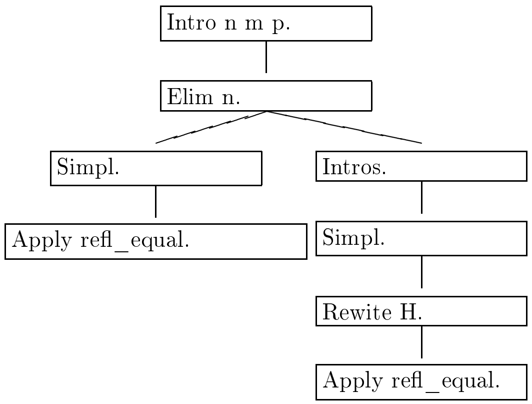



FIG. 2.4 – *l'arbre de preuve de l'associativité à gauche de l'addition*

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.

### 2.0.2 Présentation de l'environnement de preuve Ct-coq

Nous présentons ici l'interface CtCoq dans laquelle certain de nos outils ont été implémentés.

CtCoq[BBC+96] 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 [BT98, TBK92]

#### 2.0.2.1 "Point de vue utilisateur"

.

L'environement a 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 structurés*: l'environnement fournit les moyens d' éditer structurellement formules et commandes, des sous-expressions entières peuvent être réduite en un seule 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. 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 simple sélection à la souris.

– l'interface graphique et le système de preuve tournent comme deux processus indépendants.

[BKT94]

ICI UNE BELLE PHOTO............

#### 2.0.2.2 Point de vue du concepteur

Toute l'interface est basée sur l'environnement Centaur. Coq et centaur sont deux procesus séparés, il y par ailleurs une autre procesus Coq qui à été spécialise pour le parsing.

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[JB93a]. On dispose d'un langage pour manipuler les arbres de syntaxe, le VTP (Virtual Tree Processor) est une bibliothèque de fonction lisp.

La syntaxe abstraite peut être décompilée via différent pretty printer. Ces différent pretty printer sont écrit en utilisant un langage de boite PPML[JB93b]

L'ensemble du systeme est ecrit en Lisp.

Première partie

# Etude locale des théorèmes

# Chapitre 3

# Compréhension et modification de preuve

## 3.1 Compréhension

### 3.1.1 Visualisation et Navigation

Comme nous l'avons mentionné précédemment le script de preuve a naturellement une structure arborescente qu'il importe de mettre en valeur. Pour cela, on a principalement deux possibilités, une méthode graphique qui consiste à afficher l'arbre de preuve a l'aide d'un programme de dessin d'arbre. L'autre possibilité consiste à mettre en valeur la structure dans la représentation textuelle du script. Ces deux méthodes connaissent de nombreuses variantes et optimisations.

Les outils de navigation dans la représentation textuelle que nous décrivons plus loin font maintenant partie de la distribution standard de CtCoq.

#### 3.1.1.1 Visualisation graphique de l'arbre de preuve

Nous avons mené quelques expériences de visualisation de l'arbre de preuve. Dans un premier temps cela c'est fait dans le système Centaur en utilisant le serveur de graphe standard de Centaur [Hor92, Léo93].

Puis nous avons interfacer directement Coq (donc en pratique ML) avec un logiciel de visualisation de graphes: daVinci[MF96, Frö97] développé à l'université de Brême. Nous donnons ici un exemple de visualisation graphique de l'arbre de preuve dans le prototype d'interface graphique[1]

`..............................` `FAIRE UNE BELLE  PHOTO D'ECRAN`

1. l'implementation de l'interface Ml/daVinci est décrite en annexe



On y visualise la structure de l'arbre de tactiques. Il peut aussi être intéressant de visualiser l'arbre des sous-buts correspondant.

`..............................` `FAIRE UNE BELLE  PHOTO D'ECRAN`

Dans un environnement de preuve, il est d'autre part important que les noeuds de l'arbre et les lignes du script soient reliés. La sélection d'un noeud entraînant la sélection de la tactique correspondante. Cela devra être mis en oeuvre lors de l'intégration de ce prototype dans l'environnement Centaur.

**3.1.1.1.1 Ce qui se fait ailleurs** Une expérience avait avait été esquissée pour Lego en utilisant Centaur et le serveur de graphe de Centaur.

L'interface standard (Emacs) de PVS [SOR93, RSC95, ORS92, ORR+96] utilise TCL/TK[Ous94] pour visualiser les arbres de preuve.

Les différentes interfaces graphiques d'HOL permettent de visualiser l'arbre de preuve.

Omega dispose d'une interface graphique distribuée [SHB+98]

Et Kiv aussi. ###

Tous représentent l'arbre complet avec un noeud pour chaque étape de la preuve.

**3.1.1.1.2 Problèmes et solutions** Le problème majeur de cette représentation est la croissance de l'arbre de preuve. On peut assez facilement réduire la croissance verticale en regroupant "les enfants uniques" dans le même noeud que leurs parents, la représentation graphique ne sert alors qu'à mettre en lumière les embranchements de l'arbre de preuve. On dit que le l'on represente l'arbre de choix.

Par contre il est beaucoup moins évident de limiter la croissance horizontale de l'arbre. Cette limitation apparait clairement, par exemple sur les preuves de sémantique des languages qui ont pour caracteristique de faire un usage important de definitions inductives et de générer un grand nombre de cas.

Pour valider l'approche consistant à ne mettre en valeur que les points de choix, on s'est livré à quelques statistiques sur les scripts de preuve. Nous en donnons ici les resultats

`Graphe rapport nombre de noeuds unaires / nombre de noeuds totaux`

Il apparait que ce rapport est proche de ....

#### 3.1.1.2 Visualisation textuelle

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

**3.1.1.2.1 Indentation** Intuitivement pour représente la structure d'arbre sur le texte correspondant, il suffit de l'indenter en décalant chaque ligne du script en fonction de sa profondeur dans l'arbre de preuve. Nous expliquons maintenant pourquoi en pratique les choses ne sont pas si simples.

**3.1.1.2.1.1 Problème du peigne.** Comme nous l'avons mentionné dans notre présentation, l'utilisateur ne construit pas forcement son script linéairement. En variant régulièrement les sous-buts qu'il attaque il peut obtenir un script de la forme suivant:

`EXEMPLE`

Un 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 ci-dessous:

`EXEMPLE`

Donc dès que le script de preuve ne correspond pas à un parcours de l'arbre de preuve en profondeurs d'abord et de la gauche vers la droite, l'indentation interactive du script devient sans intérêt si l'on n'en modifie pas la structure au préalable. Modifier la structure du script à la volé ne semble pas non plus une solution viable. D'une part cela modifie constamment la vision que l'utilisateur à de son script supprimant toute trace de la stratégie suivie, d'autre part pour cela oblige a de nombreux parcours du script pour insérer chaque nouvelle commande en bonne position.

Dans le cas d'un éditeur structure comme Ctcoq ces parcours du script sont des parcours de l'arbre de syntaxe abstraite (AST).

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

Dés lors nous avons préféré ne pas modifier la structure du script en cours de développement mais l'instrumenter pour permettre d'y simuler la navigation dans l'arbre de preuve.

#### 3.1.1.2.2 Navigation

**3.1.1.2.2.1 Exemple de navigation dans un script de preuve** Sur notre exemple figure 3.1, on pourra en sélectionnant la tactique `Apply refl_equal` à la ligne `7`, retrouver quelle tactique a généré le but qui est abordé par la tactique sélectionnée (c'est-à-dire, de retrouver son **père** dans l'arbre de preuve). 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 ses `fils` ce qui nous donnera la tactique `Simpl` de la ligne `5`. On



```
1   Intros n m p.
2   Elim n.
3   2:Intros.
4   1:Simpl.
5   2:Simpl
6   2:Rewrite ->H.
7   1:Apply refl_equal.
8   1:Apply refl_equal.
```

FIG. 3.1 – *Un script de preuve de l'associativité à gauche de l'addition*

peut enfin demander à voir toute la dépendance 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 combinaison de touches qui correspondent aux différentes actions possibles: retrouver le père, les enfants, toute une descendance, les frères etc ... On montre ici quelques exemples de navigation dans un script de preuve joué dans l'environnement CtCoq:

`DE BELLE COPIES D'ECRAN`

De plus dès qu'une tactique du script est sélectionnée, il est possible de demander l'affichage du sous but qu'elle a attaqué.

`Exemple`

La conjugaison de ces deux possibilité améliore réellement la compréhension d'un preuve en cours

De plus nous fournissons la possibilité, fois la preuve fini de réorganiser le script. Ce que l'on a appelé "le problème du peigne disparaît" disparaît et le script peut alors être indenté. Comme dans le cas de la visualisation graphique, Nous en limiterons la croissance horizontale tout en mettant en évidence les embranchement 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 croit pas trop vite en largeur. Les scripts des figure 2.1 et 3.1 se reecrivent tout les deux comme illustré sur la figure 3.2.

### 3.1.2 La relation tactique/sous-but

Dès que l'on a une interface graphique et 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.

```
1    Intros n m p.
2    Elim n.
3        Simpl.
4        Apply refl_equal.

5        Intros.
6        Simpl.
7        Rewrite ->H.
8        Apply refl_equal.
```

FIG. 3.2 – *Script "remis en forme"*

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

Dans l'exemple ci-dessous[2] (figure 3.5), il y a deux sous-buts, on a sélectionné le second sous but dans la fenêtre d'état(fenêtre inférieur) et la sélection de la fenêtre d'édition (fenêtre supérieur) a été déplacée sur la tactique `Split` qui a generé ce second sous-but.

## 3.2 Mécanisme de Undoing

Le système de retour en arrière (Undo) de Coq n'utilise que les dépendances historiques entre les tactiques. 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 preuves logiquement indépendants de la tactique à effacer.

Ainsi si l'on désire effacer la tactique `Simpl` à la ligne 4 du script de la figure 3.1 on depilera 4 fois l'état courant (`Undo 4`)et on ne gardera que le script suivant:

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 coup 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 donc implémenté un mécanisme de "undoing" 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 rejouée immédiatement. Nous appelons "Undo-redo" la commande implémentant ce mécanisme.

Après application de la commande "Undo-redo" sur la ligne `4` du script de la figure 3.1 on obtient le script suivant

2. L'affichage des sous-buts dans la version courant de CtCoq n'est plus celui de cette figure mais cette fonctionnalité n'y a pas encore était portée



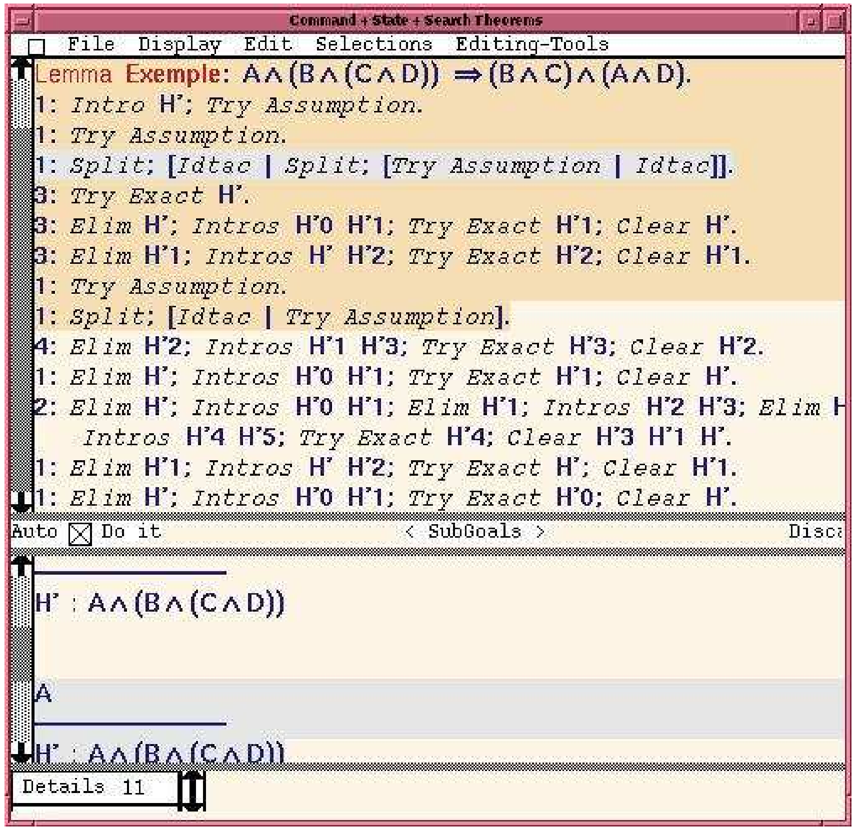



FIG. 3.3 – *Relation entre les fenêtres d'édition et d'état*

Le script reste cohérent. Tous les numéros de lignes (indiquant à quel sous-but elles s'appliquent) de la partie du script conservée 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 numéros des buts sont néanmoins remis à jour pour permettre à l'utilisateur qui le voudrait de rejouer immédiatement cette partie de preuve.

### 3.2.1 La commande "Undo-Redo"

Comme nous venons de la dire, le problème du "Undo-redo" est de conserver la cohérence du script. En effet si l'on se contente d'effacer du script l'ensemble des tactiques correspondants à la branche de l'arbre de preuve que l'on desire couper. Le script restant peut devenir incohérent et ne pourra alors pas etre rejoué lors d'une prochaine session de

(

| | |
|---|---|
| 1 | Intros n m p. |
| 2 | Elim n. |
| 3 | 2:Intros. |

FIG. 3.4 – *Apres un undo classique*

| | |
|---|---|
| 1 | Intros n m p. |
| 2 | Elim n. |
| 3 | 2:Intros. |
| 4 | 2:Simpl |
| 5 | 2:Rewrite ->H. |
| 6 | 2:Apply refl_equal. |
| ———————————— | |
| 7 | 1:Simpl. |
| 8 | 1:Apply refl_equal. |

FIG. 3.5 – *Un script après application de "Undo-Redo"*

preuve.

```
TO do
ddddd
```

FIG. 3.6 – *Un script devenu incoherent*

**3.2.1.0.3 Exemple** Pour garder la coherence, il faut éventuellement modifier les numéros indiquant à quel sous-but s'applique chaque tactique.

En pratique on a deux problèmes distinct garder la cohérence du script restant pour qu'il puisse être rejoué dans une prochaine session et restituer la cohérence de la partie effacé afin de pouvoir éventuellement la rejouer dans la session courante.

Nous introduisons deux notions:

– Le rang d'une tactique est sa position dans le script. Cela correspond a l'ordre dans lequel ont été entrées les tactiques. $rang(t1) < rang(t2)$ ssi $t1$ a été joué avant $t2$.

– Le chemin d'une tactique correspond a sa position dans l'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. Les chemin seront donc simplement des suite d'entiers. On utilisera deux ordre sur l'ensemble des chemin:

1. Un ordre total, l'ordre lexicographique:

$$() < (1) < (1;1) < (1;1;1) < \ldots < (1;1;2) < (1;1;2;1) < \ldots < (1;2) \ldots (1;3) < \ldots (2) \ldots$$



2. Un ordre partiel, l'ordre préfixe:

$$() < (1) < 1;1) < (1;1;1) < 1;1;1;2) \ldots$$

Dans le pseudo code qui suit, ces deux relations sont notées respectivement $>$ et `est-prefixe-de`

#### 3.2.1.1 Garder la cohérence du script restant malgré les lignes qui ont été supprimées

:

Après avoir récupéré la tactique sélectionnée et calculé sa liste de dépendances, on va effacer toutes les tactiques de cette liste de la zone de script que l'on a déjà jouée; à chaque fois il faudra modifier les numéros de but dans la partie restante pour qu'elle reste valide. En pratique on commence par modifier ces numéros sans rien effacer puis on déplace en bloc ce qui doit l'être.

Remarquons d'abord que toute les tactiques qui, dans le script sont antérieures à celle que l'on va effacer (ie., qui ont un rang inférieur) ne sont pas affectées. Celles qui lui sont inférieure (pour l'ordre lexicographique), ne le sont pas non plus.

Nous parcourons donc le script à partir de la ligne à effacer et pour toutes les lignes dont le chemin dans l'arbre de preuve est supérieur (pour l'ordre préfixe), à celui de la ligne de départ, on met à jour les numéros de débuts de lignes.

Voici le pseudo code de la fonction qui met à jour les numéros de but pour les tactiques qui ne sont pas effacées. Elle prend la tactique `T` qui correspond à la tactique où l'on veut faire le `Undo-Redo`

```
Mise-a-jour-des-numeros-de-but-de-la-partie-conservée T
------------------------------------------------------
  decalage := nombre-de-sous-but T
  t        := t-initial
 TANT QUE t Dans script-joué  FAIRE
     SI  rang (t) > rang (T)
       ALORS  SI chemin (t) > chemin (T)
                ALORS SI  (chemin T) est-prefixe-de (chemin t)
                        ALORS (decalage := decalage + (nombre-de-sous-but t) -1)
                               (suivant t)
                        SINON (numero t) <- (numero t) - decalage
                              (suivant t)
                SINON (suivant t)
       SINON (suivant t)
 FIN TANT QUE
```

Enfin, on supprime la tactique sélectionnée et toutes les dépendances, pour les déplacer à la fin de la partie du script qui a déjà été jouée .

#### 3.2.1.2 Pouvoir rejouer la partie effacée

:

Il s'agit de modifier les numéros des tactiques déplacées afin qu'elles puissent être rejouées sans modifications. Pour cela on commence par calculer le numéro de but de la tactique racine du sous-arbre que l'on a déplacé. L'idée est de parcourir la partie de script qui a été conservée et pour chaque tactique dont le chemin est inférieur à celui de notre tactique racine, on fait la différence entre le nombre de sous-buts qu'elle génère et le nombre de ses fils présents dans le script dont le chemin est inferieur a celui de la tactique racine. La somme de ces différences nous donne le nombre de sous-buts ouverts dans la partie de l'arbre de preuve qui est inférieur à la tactique que l'on a effacée. Voilà le pseudo code de la fonction qui met à jour le numéro de but de la tactique `T` sélectionnée pour l'effacement.

```
Coherence-de-la-racine T
------------------------
  difference := 0
  t          := t_initial
 TANT QUE t Dans script-conservé  FAIRE
      SI chemin (t) < chemin (T)
         ALORS (difference := difference +
             ((nombre-de-sous-but t) -
             (nombre-de-fils-de-chemin-inf-a-racine-present-dans-le-script-joué))
             (suivant t)
         SINON (suivant t)
  FIN TANT QUE
(numero T) <- difference
```

Pour finir on met à jour les numéros des buts dans le sous-arbre effacé; pour cela on parcourt la partie de script qui vient d'être effacée et pour chaque tactique, si c'est un $i^{eme}$ fils de la tactique précédente, on augmente le numéro de la tactique précèdente de $(i-1)$, sinon on l'augmente du nombre de buts qu'elle a génèrés moins un, majoré du nombre de sous-but non résolu qui se trouvent entre nos deux tactiques dans l'arbre de preuve. Voici enfin, le pseudo code de la fonction qui met à jour les numéros de but sur le sous-arbre de la tactique `T` sélectionnée pour l'effacement.



```
Coherence-dependance-de-la-tactique-selectionne T
------------------------------------------------
  numero := (numero T)
  t         := (suivant T)
 TANT QUE t Dans script-deplacé  FAIRE
     t':= (precedent t)
     SI (chemin (t) = chemin (t')@i)               ;; i.e. si t est un fils de t'
                ALORS (numero := numero + (i - 1))
                      (numero t) <- numero
                      (suivant t)
                SINON (numero := numero +
                                 (nombre-de-but-de t') +
                                 ((dernier (chemin t))-
                                  (n-iem (longeure (chemin t)) (chemin t')))
                                 - 1
                      (numero t) <- numero
                      (suivant t)
 FIN TANT QUE
```

Les mises à jours sur le script etant achevées, on modifie la structure interne du prouveur, pour mettre système et le script en conformité. .

La figure suivante montre l'utilisation du Undo-redo dans l'environement Ct-Coq

## 3.3 Aperçu de l'implémentation

### 3.3.1 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 tous une structure arborescente. Le manipulateur d'arbres (Virtual Tree Processor) que nous utilisons fournit une structure d'arbre doublement líes, de telle sorte que l'accès au père d'un terme donné se fait en temps constant. Les noeuds 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 avec le chemin auquel elle correspond dans l'arbre de preuve (l'annotation étant une liste d'entiers), avec le nombre de but(s) qu'elle a généré(s), par une indication de son niveau de profondeur dans la preuve, et par un nom de thèoreme qui indique à quelle preuve se rattache cette ligne de script.

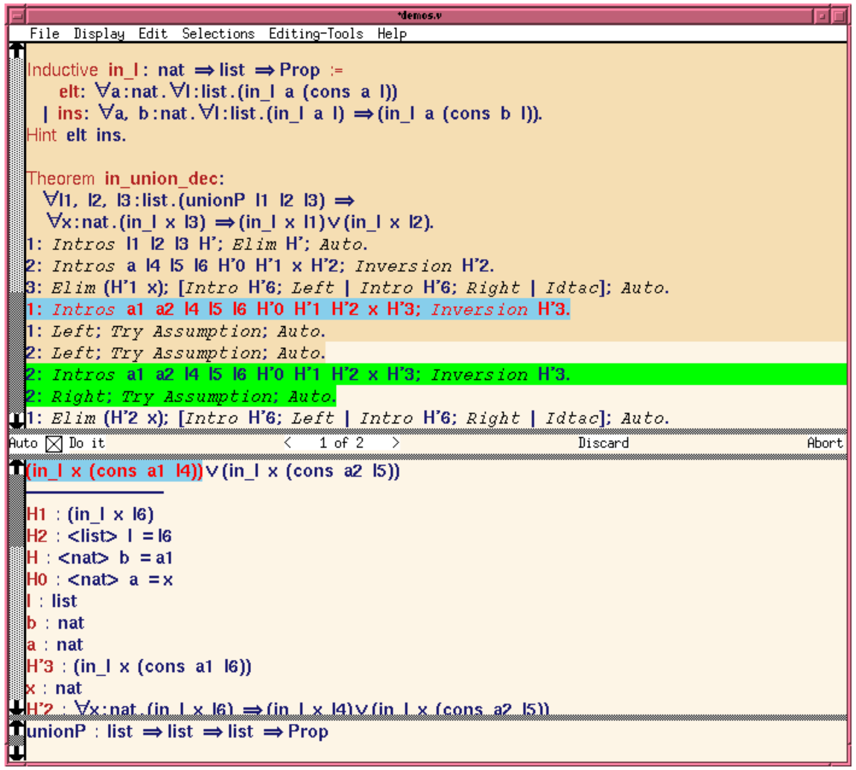



FIG. 3.7 – *L'environement CtCoq après un Undo-Redo*

Certaines opérations comme rechercher la tactique associée à un chemin donné, ont une complexité temporelle en O(n). 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 donc une table de hachage (chemin,tactique) pour chacune des preuves du script. Ces tables sont elles même conservées dans une autre table utilisant comme clef les noms de théorème. Cette table globale est elle même une annotation du script complet.

### 3.3.2 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 donné, nous avons ajouter au système coq une commande pour effacer la sous preuve d'un sous-but référencer 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 correspondants 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 but générés, (c'est le nombre de sous-buts reçus) et au nom du théorème.

### 3.3.3 Gestions 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 preuve 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 possibilités d'ouvrir et de fermer chacune des fenêtre séparément. Quand on ferme une fenêtre, il est très important de récupérer la mémoire correspondant au 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.

## 3.4 Undo logique 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 undoing et essayons d' esquisser une solution. Mais voyons d'abord succinctement l'intérêt des variables existentielles et comment on les manipule en Coq

### 3.4.1 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 $X$.

Cet exemple peut être traduit en Coq:

```
Lemma equation : {n:nat  |(eq nat (plus n (S(S O))) (S(S(S(S(S O)))))) }.

 EApply exist.
(Generalize plus_n_Sm ; Intros H; Rewrite <- H; Rewrite <- H).
(Generalize plus_n_O ; Intros H0; Rewrite <- H0).
(Apply eq_S; Apply eq_S).
 Trivial.
Defined.

(* notons que l'on peut récupérer le témoin *)
Lemma solve :(A:Set)(P:A->Prop)(sig A P)->A.
  Intros A P H0;Inversion H0;Trivial.
  Defined.

Syntactic Definition Solve := (solve ? ?).
(*
Coq <  Compute (Solve equation).
     = (S (S (S O)))
     : nat
*)
```

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

### 3.4.2 Manipulation des variables existentielles en Coq

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

- `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 ident` (essaie d'instancier la n-ième variable existentielle avec `ident`)



- `NormEvars` (remplace les variables contraintes par leurs valeurs)

Dans la section suivante on va s'intéresser à l'interaction entre le mécanisme de Undo logique décrit précédemment et la présence de méta-variables. On verra que pour fonctionner en leur présence le mécanisme de Undo doit être modifié.

Il nous faut par ailleurs écrire le bout de code ML permettant de désinstancier une variable existentielle.

### 3.4.3 Interaction avec le Undo

En présence de variables existentielles les différentes branches de l'arbre de preuve ne sont plus forcément indépendantes. Si le script de preuve est "explicite", c'est-à-dire si les tactiques instanciant des variables existentielles se comportent de la même façon que ces variables soient instanciées ou non, le "Undo logique" fonctionnent toujours. Considerons le lemme suivant:

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

Fig. 3.8 – *lemme1*

Nous en donnons un script de preuve explicite figure 3.9.

```
Intros.                                         Intros
EApply trans_equal.                             EApply trans_equal.

               (* conduit a deux sous-buts *)
               (* ...
                   H : a=b
                   H0 : b=c
                   H1 : a=d
                   ============================
                      a=?

                    sub-goal 2 is:
                       ?=c
                    *)
(* on attaque d'abord le second *)         (* on attaque d'abord le premier *)
2:EExact H0. (* contrainte ?=b *)  ------>> EExact H0.    (*contrainte ?=b *)
EExact H.                                       EExact H.
```

Fig. 3.9 – *Une preuve du lemme1 par un script explicite*

La seconde branche issue de l'application de la transitivité peut être effacée, le reste du script reste valable, l'instanciation ou la désinstanciation de la variable existentielle n'en modifie pas le comportement.

Par contre dès que le comportement n'est plus explicite, le "Undo logique" ne fonctionn e plus. Comme dans cet autre script de preuve du lemme de la figure 3.8 Ici le try EExact

```
Intros.                                     Intros.
EApply trans_equal.                         EApply trans_equal.
EExact H0.                         ---->>   Try EExact H1.(* instancie ?=d *)
                                                       (* et resoud le sous-but *)
Try EExact H1.   (* ne fait rien *)         EExact H. (* obsolète *)
EExact H.                                   EExact H0.(* unifiaction impposible *)
```

FIG. 3.10 – *Une preuve du lemme1 par un script non explicite*

ne fait rien car `EExact H1` échoue à cause des contraintes introduites à la ligne précédent. Si on efface cette ligne `try EExact` réussi, resoud le premier sous-but et introduit d'autres contraintes (`?=d`) et le reste du script n'est donc plus valide.

La situation est la même lorsque l'on utilise des tactiques automatiques dont le comportement dépend des contraintes sur les variables existentielles. Dans cet autre script de preuve du lemme de la figure 3.8

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

Si on efface la deuxième branche et que l'on veut la rejouer. Soit le script suivant,

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

On n'arrive plus dans le même état. Car avec la nouvelle contrainte introduite par le premier `EAuto` oblige à des étapes de réécriture que le second `EAuto` ne sait pas faire.

## 3.5 Solution du problème

On doit ajouter 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.



### 3.5.1 Principe de base

On maintient l'information qui indique à quel endroit dans le script les variables ont été instanciées. Ainsi on peut ainsi couper une branche donnée (en utilisant les dépendances logiques) et minimiser les modifications à apporter au reste du script pour qu'il reste cohérent avec les contraintes liées aux variable existentielles. Dès que l'on efface une tactique contenant des meta-variables instanciées, les branches logiquement indépendantes mais situées plus bas dans le script et contenant des tactiques agissant sur des sous-buts contenant les mêmes variables existentielles que celles instanciées dans la branche effacée doivent être précédées d'une tactique forçant les instanciations. Si de telles branches n'existent pas il faut désinstancier les variables existentielles qui ont été instanciées dans la sous branche effacée.

### 3.5.2 Instrumentation

En plus de l'instrumentation nécessaire au "Undo logique", chaque tactique doit donc être annotée par la liste des variables existentielles instanciées qu'elle contient et la liste de celles qu'elle instancie.

### 3.5.3 Fonctionnement

Lors du undo,

- La tactique à effacer est sélectionnée
- Ses dépendances logiques sont calculées
- On commence à effacer les dépendance logiques. Pour chaque tactique à effacer:
  - si elle comporte une instanciation d'une (ou plusieurs) des variables existentielles présentes dans la racine, la première tactique situé plus bas dans le script et logiquement indépendante doit être précédée d'une instanciation de la variable. Ainsi si l'on rejoue ce script les contraintes seront les mêmes que précédemment. Puis on l'efface et on passe à la suivante.
  - S'il n'y a pas d'instanciation on l'efface et on passe à la suivante.
- On termine en remettant à jour les préfixes des tactiques comme dans le "Undo logique" précédemment décrit.

### 3.5.4 Conclusion sur les varaibles existentielles

En pratique, l'utilisation des variables existentielles dans un système comme Coq pose de nombreux autres problèmes. Leur présence allié aux types dépendant complique sensiblement le mécanisme de l'unification. Leur emploi semble encore peut répandu: une rapide analyse des "contrib" de la distribution standard révèle que `EAuto` apparaît 2 fois et `EApply` 13 fois sur un total de 118106 lignes de code. Les problèmes liés aux variables existentielles ne constituent donc pas réellement un obstacle à l'utilisation de nos outils dans l'état actuel du système Coq.

## 3.6 Autres études du mécanisme de Undoing

Le problème du "Undo" a déjà été étudié par Lena Magnusson[Mag93] dans le prouver 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 "Undo local". On n'est donc pas confronté au problème de tactique dont le comportement change en fonction des contraintes.

Amy Felty et Douglas Howe[FH94c, FH94d]ont aussi etudié le problème dans le cadre du prouveurs d'Amy. Il propose aussi de pouvoir 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.

Citons aussi les mécanismes de révision de preuve décrits par Andrew Stevens [ste96] et sur lesquels nous reviendrons dans le prochain chapitre.

Nous allons maintenant aborder des mécanismes plus fin de manipulation de preuve et notamment la lemmification et la factorisation de preuve.

# Chapitre 4

# 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 (typiquement CtCoq permet d'avoir plusieurs fenêtre de preuve pour prouver plusieurs résultats "en parallèle") mais cela nécessite de savoir a priori quelles hypothèses du contexte locale vont être utiles à sa preuve et doivent être abstraites pour construire l'énoncé. La solution que nous avons choisit consiste à prouver le résultat dans le contexte local, à repérer les hypothèses de ce contexte qui ont été réellement utilisées, à les abstraire pour construire l'énoncé du théorème et finalement à extraire la partie de script correspondant à sa preuve.

## 4.1 Lemmification

### 4.1.1 Exemple

Imaginons qu'ayant définie l'addition 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 donc le script suivant (les commentaires rappellent le chemin dans l'arbre de preuve)

```
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 prouver des résultat intéressant par eux même et qui pourront être utiles dans d'autres preuves. Il décide donc d'en faire des lemmes à part entière. Pour cela il utilise une fonction de lemmification qui a pour arguments un chemin dans l'arbre de preuve et un nom. Cette fonction génère le lemme correspondant au sous but associé à ce chemin.

Ainsi il dans cet exemple il pourra géné

```
# 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.
```

Des lors anticipant sur la notion de factorisation 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 recherche du sous-but intéressant et de son chemin dans l'arbre de preuve se fera par de simples clics souris. Les lemmes générés seront insérés automatiquement dans le script



### 4.1.2 Principe de fonctionnement

Le problème est don de dégager du script de la preuve `P` d'un lemme `L` donné, la sous preuve `SP` d'un sous but `SG` pour en faire un lemme à part entière `NL`. `NL` est alors prouvé automatiquement en reutilisant `SP`.

**Le principe est le suivant:**

1. Retrouver dans le script, la tactique `T` ayant attaqué le sous-but `SG`.
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 `P` le sous terme `ST` correspondant au script de preuve SSG du sous but `SG`.
4. Parcourir `ST` pour savoir quelles sont parmi les hypothèses de `BH`, celles qui ont été réellement utilisées soient $BU_1 : T_1 \ldots BU_m : T_m$.
5. Créer le Lemme `NL`:

   ```
   Lemma NL := (BU1:T1)...(BUm:Tm)(SG).
   ```

   On crée le script de preuve correspondant:

   ```
   Intros BU1,...,BUm.
   SSG
   ```

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

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

Dans l'environnement CtCoq, les outils de navigations du chapitre précédent permettent de naviguer dans le script et de retrouver simplement le sous-lemme que l'on veut extraire.

## 4.2 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-preuves et l'utiliserait ensuite dans tous les sous-buts ou cela est nécessaire.

### 4.2.1 Exemple

On considère le lemme trivial ci-dessous:

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

qui peut être prouvé par le script suivant (dont on numérote les lignes pour pouvoir y faire référence)

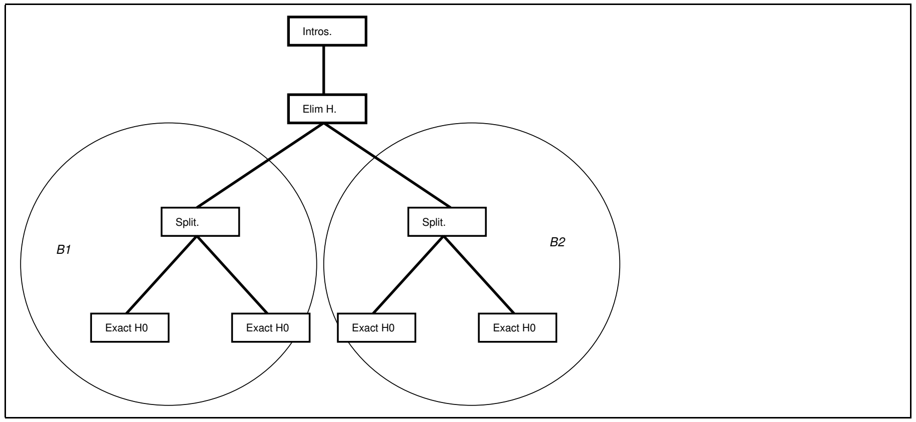
```
1        Intros.
2        Elim H.
3        Split.
4        Exact H0.

5        Exact H0.

6        Split.
7        Exact H0.

8        Exact H0.
```

Et qui correspond au $\lambda$-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`.

```
3: Split
4: Exact H0

5: 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.

Il faudra donc prévoir de décaler les indices sur les noms de variables en fonction du contexte. Ici, comme il n'y a plus d'hypothèse `H` dans le contexte, il faut décaler tous les $H_i$ en $H_{i-1}$ soit le script suivant :

```
3: Split
4: Exact H

5: Exact H
```

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

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

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

intermédiaire < Exact [H0:A](conj A A H0 H0).
Subtree proved!

intermédiaire < Show Proof.
LOC:
Subgoals
Proof: [H0:A]<(A),(A)>{H0,H0}
```

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 on ne veux pas surcharger l'environnement, en ajoutant un lemme comme `intermédiaire` on pourra 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:

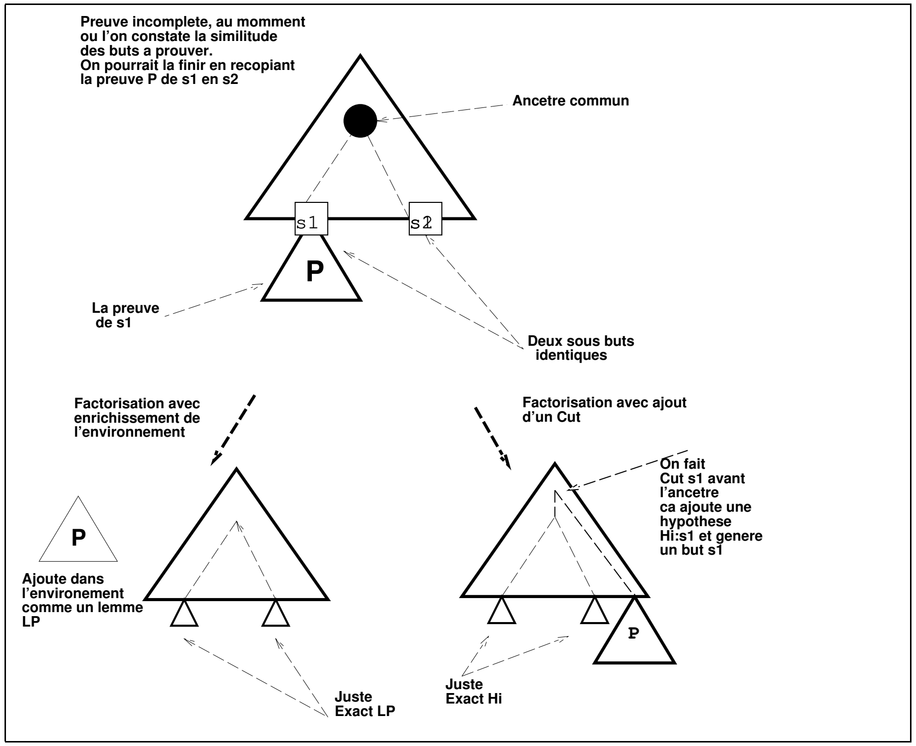
```
Intros.
Cut A->A/\A.
Intros.
Elim H.
Exact H0.

Exact H0.

Split.     (*                                *)
Exact H0   (*                                *)
           (* ou Exact [H0:A](conj A A H0 H0). *)
Exact H0   (*                                *)
```

Le principe est le suivant:

1. .

2. .

**Illustration du principe de factorisation:**




### 4.2.2 La tactique Similar

La tactique `Similar` vise comme son nom l'indique à exploiter les similarités de certaines sous preuve. Pour une question de coût, ces similarités qui se limitent pour l'instant à une convertibilité ne sont détectées systématiquement, c'est a l'utilisateur de se dire "tient 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 quelle hypothèses ont été nécessaires à sa preuve. Si ces hypothèses sont satisfaites dans le contexte courant, on peut réutiliser la preuve qui est alors mise en facteur.

`Exemple`

### 4.2.3 Conclusions et perspectives

## Deuxième partie

# Etude global des théorèmes

# Chapitre 5

# Introduction à la gestion des théories

## 5.1 Qu'est ce qu'une théorie

La notion de théories est très variable selon les systèmes. Elle dépend autant de leur état de développement que de la logique sur laquelle ils reposent.

#### 5.1.0.0.1 Librairie

#### 5.1.0.0.2 Théorie

**5.1.0.0.3 Module** (### refaire rapidement un état de l'art citer les théories d'HOL, l'affichage du graphe de dépendance, les modules en Isabelle et la perspective de module en Coq issue des travaux de Judicaël Courant [Cou97] ainsi que les théories de PVS et les "little théorie" de IMPS)

Dans la suite nous appellerons une théorie un fichier (ou un ensemble de fichiers) compilable séparément. Pour Coq une théorie est donc un fichier ".v" compilable en un ".vo".

## 5.2 Dépendances entre 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 consistance 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*[Fel79] 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 déclarés 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* qui parcourt les sources et utilise les `#include` pour calculer les dépendances. Dans le monde des langages fonctionnels, comme par exemple *Caml*[WL94, Ler95] on dispose d'outils comme *camldep*, ou dans le monde des système de preuves comme *Coq* on a l'outil *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 syntaxiques des fichiers sources.

Les outils travaillant à une granularité plus fine que celle des fichiers sont moins classiques. Dans le cadre des langages fonctionnels signalons [BS91] qui décrit un environnement de développement pour *Standard ML*[AM91] sous *GNU Emacs* dans lequel il maintient les dépendances entre fonctions et les utilise pour simuler une liaison dynamique alors que la liaison de *SML* est statique. Ainsi lorqu'une fonction est modifiée toutes les fonctions qui l'utilisent feront référence à la nouvelle version de la fonction modifiée.

Dans la fin de ce chapitre nous donnons quelques exemples de graphes de dépendances et d'outils de visualisation des dépendances entre théories. Dans le chapitre 6 nous nous intéressons au calcul des dépendances entre théorèmes et définitions dans les systèmes d'aide à la preuve. L'objectif est de savoir lorsqu'un théorème, une preuve ou une définition est modifié, quels sont ceux qui doivent être vérifiés (dans le cas des outils d'aide a la preuve,on dit qu'ils doivent être rejoués). On développe ensuite au chapitre 7.1 un outil interactif d'aide à la propagation des modifications. Notons que cela est aussi valable pour 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.

### 5.2.1 Utilisation des Dépendances entre théories

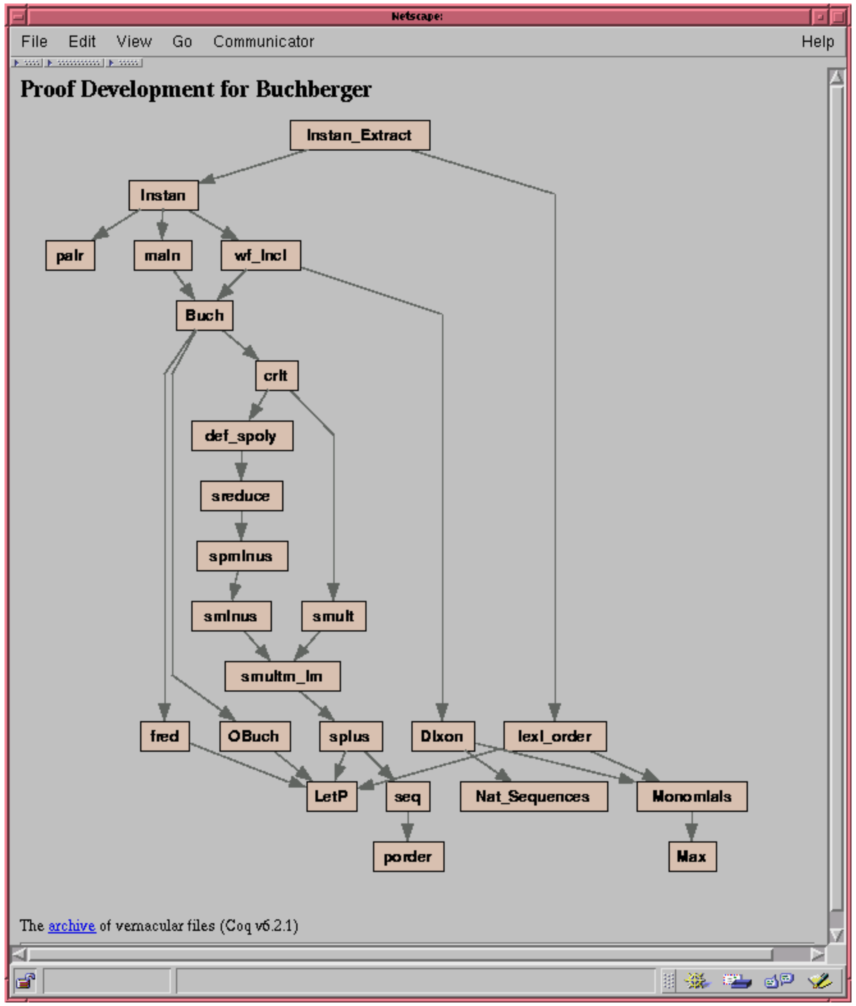
### 5.2.2 Compilation séparé et cohérence

L'outil `Coqdep`

### 5.2.3 Visualisation et génération de document utilisable par un broswer web

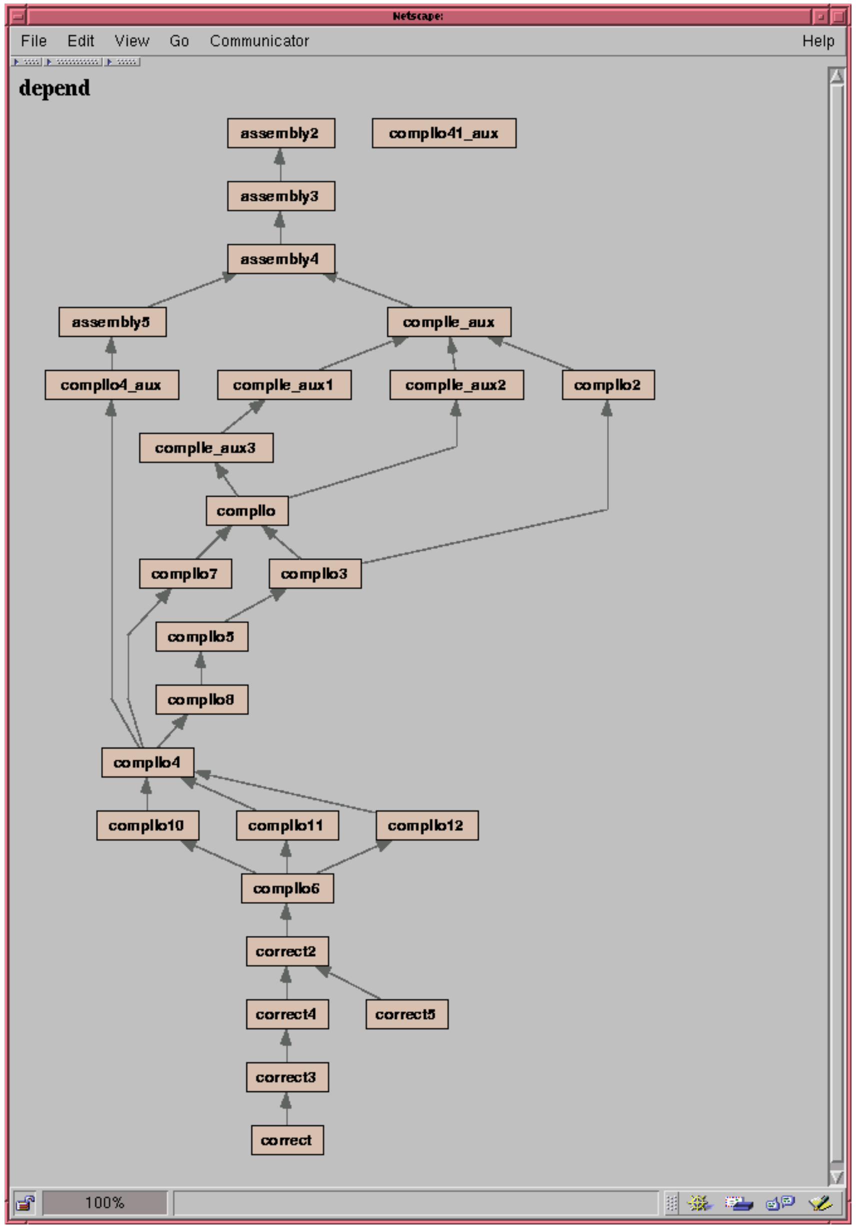
Nous donnons ici deux exemples de visualisation de l'organisation d'un gros développement.

D'abord un developement dû à Laurent Théry [Thé97] conduisant à une preuve de l'algorithme de Buchberger pour le calcul de bases de Gröbner:




Ce dernier exemple visualise les relations entre les différents fichiers d'une preuve de compilateur faites par Yves Bertot[Ber98].






Le graphe est visualisé en utilisant un fichier html contenant un .gif "clickable".Le tout est généré automatiquement à partir d'un fichier de dépendance au format Make, en utilisant `dependtohml` [PP98].

Les liens correspondants aux noeuds terminaux peuvents permettre d'accéder aux fichiers correspondant ou visualiser sur un nouveau graphe les dépendances entre les différents objets de ce fichier.

Cela nous amène naturellement au sujet du chapitre 6, les dépendances entre objets d'une théorie mais il nous faut mentionner avant un dernier outils de visualisation qui permet de tracer les graphes de "coercion".

En effet en mathématique on a naturellement une hiérarchie des théories, ainsi un anneau est aussi un groupe qui est lui même aussi un monoide etc. Cela pose le problème du sous typage. Dans le système Coq, on peut simuler le sous typage via les "coercion" implicites[?].

Il nous a paru utile de permettre de visualiser graphiquement ce graphe. L'exemple suivant montre le cas de la théorie des ...

`TO PUT`

FIG. 5.1 – *Le graph de coercion de la théorie ....*

# Chapitre 6

# Calcul et application des dépendances entre objets d'une théorie

## 6.1 Introduction

## 6.2 Calcul des dépendances entre objets d'une théorie

.

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 (* fichier .v *)
2. Sur le terme de preuve (* fichier .vo *)

Il faut ici signaler que lorsque qu'une preuve est finie, la seule chose qui est conservé par le système est le terme de preuve. De même les scripts écrits par l'utilisateur sont conservés dans les fichier (suffixés par `.v`) tandis que la version compilées de ces fichiers (suffixés par `.vo`) ne contient que les termes de preuves.

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 ou tout les fichiers du `Prelude` de coq[1] ont été

1. Le Prelude est un ensemble de fichiers chargés par défaut et qui contiennent les définitions de base notamment la définition de l'egalité, des entiers et quelques unes de leurs propriétés de base.



chargés et où on a défini les listes `list`, la concatenation sur les listes (`app`) et les deux théorèmes suivant:

```
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)).
```

**6.2.0.0.4 Calcul des dépendances sur le Script** On peut ennoncer 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 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 concatenation 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 concatenation et la symétrie de l'égalité. En particulier 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. Donc notre théorème peut potentiellement dépendre de tout ce qui a été précédemment défini.

**6.2.0.0.5 Calcul des dépendances sur le terme de preuve** Nous considerons maintenant le terme de preuve de l'associativité à gauche de la concaténation de liste. C'est le terme associé a l'identificateur `ass_app` soit :

```
ass_app =
[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 qui apparaissent dans le terme et sont les suivants:

- `list`, qui correspond à la définition inductive des listes

– `app`, qui correspond à la définition de la fonction de concatenation des listes

– `sym_equal`, qui correspond à la symétrie de l'égalité

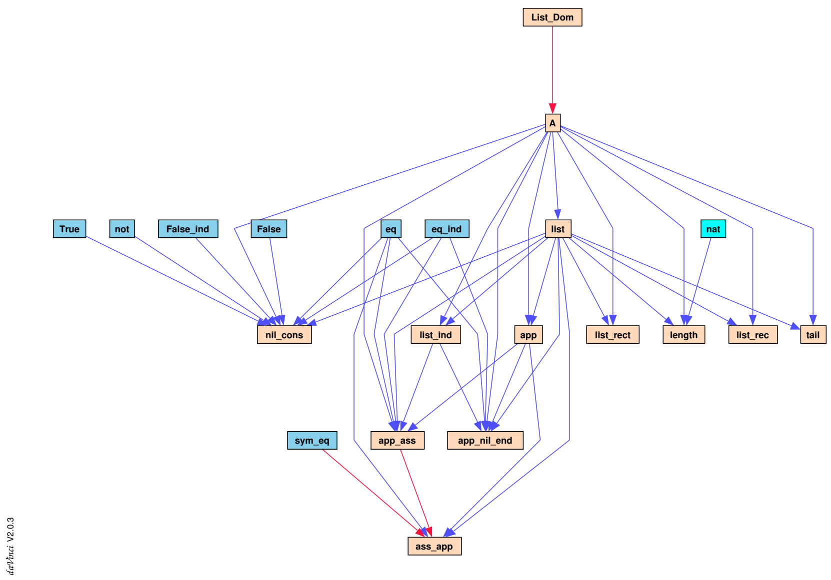
– app_ass, qui correspond à l'associativité à droite de la concaténation.

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

Appliquant récursivement le calcul précédent à tout les termes de l'environnement, on peut construire le graphe de dépendance d'une théorie. La figure ci-dessous montre le graphe de dépendance d'une infime partie de la théorie des listes.






## 6.3 Le calcul des dependance et la construction des graphes

### 6.3.1 Principe de base

Comme nous venons de le voir notre calcul de déependance va se faire sur le terme de preuve. Au cours d'une session de preuve, ces termes sont ajouté dans un environnement au fur et à mesure de leur définition.

L'environnement a donc grossièrement une structure de listes. Un objet est accessible dans l'environnement grâce à un chemin d'accès, le `section_path`. Ce chemin d'accès est construit à partir du nom de l'objet (pour une description plus fine on se reportera au manuel de référence).

Pour calculer les dépendances d'un objet on construit donc 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 dépends.

On aura deux modes de calcul des dépendances, en interactif ou en batch.

### 6.3.2 Mode interactif

En mode interactif dès qu'un identificateur est ajouté à l'environnement on calculera ces dépendances. On construit donc le graphe au fur et à mesure du développement. On pourra visualiser ces dépendances graphiquement ou par une navigation dans le script de preuve.

Cela permettra par ailleurs de manipuler les objets de l'environnement en cours de développement avec des application comme le **Reset Logique** que nous décrivons au pargraphe 6.4.0.0.9..

### 6.3.3 Mode batch

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

On peut 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 ajouter directement dans l'environnement. On parcourt alors cet environnement calculant pour chaque objet ses dépendances.

### 6.3.4 Construction et manipulation de la structure de graphe

Pour les applications de visualisation, on peut se contenter de générer un fichier au Mme format que les fichier de dependance lisible par Make. Par exemple pour le fichier Liste.v de la bibliotheque standard on aurait:

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

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

Les suffixes `.fct` sont là pour pouvoir récupérer le parseur fait pour les fichiers mais sont appelé à disparaître

FIG. 6.1 – *Dépendence entre objets de la théorie des List au format Make*

Pour les applications plus compliquées il est préférable de travailler avec une structure de graphe en Caml. Nous avons utilisé une représentation classique sous forme de listes d'adjacence.

Voyons maintenant quelques utilisations possibles de ce graphe de dépendance.



## 6.4 Quelques exemples d'applications utilisant le graphe de dépendance

```
ENCORE UN BÔ GRAPHE !
```

FIG. 6.2 – *Visualisation du graphe de dependance*

#### 6.4.0.0.6 Exemple 1: Visualisation

**6.4.0.0.7 Exemple 2: Épuration de théories** Nous commencerons par une comparaison avec la notion de coupe de programme (slicing) [Wei81] issue des méthodes d'analyse de flot en génie logiciel.

Le slicing peut se resumer 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 tous 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 théoreme 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 théoremes, définitions etc. dont dépend (transitivement) le théorème que l'on a choisit. On obtient la "théorie minimale" sur laquelle repose ce théorème, autrement dit une coupe (slice) de la théorie de départ par rapport au théorème etudié.

Notons que la notion de minimalité est relative au développement initial. Le seul ordre dont on dispose naturellement sur "l'ensemble des script" 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 script incomparable pour cette relation pouvant mener à la preuve d'un même théorème.

Avec la notion de théorie minimale, apparaît le premier "problème". Il est liés au fait que bien que calculant les dependance sur le terme de preuve, c'est le script que nous manipulons et que nous voulons être capable de rejouer.

Et comme nous le montrons au chapitre 8, calculer l'ensemble des objets dont dépend un théorème donné nous permet d'affirmer qu'il existe une preuve de ce théorème n'utilisant que ces objets (c'est le terme de preuve que nous avons construit). Mais cela ne permet pas d'affirmer que le script obtenu par coupe de la théorie de départ pourra être rejoué.

#### 6.4.0.0.8 Exemple 3: Maintenance de théorie

**6.4.0.0.8.1 Propagation de dépendance** Pour continuer notre parallèle avec les notions de l'analyse de flot on s'est ensuite intéressé à ce qui apparaît parfois sous le nom de "ripple effect analysis"[YCM78]. 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 affectés 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 modifié pour rester valide. On veut aussi lui fournir des outils pour l'assister dans la propagation des modifiactions.

Intuitivement il semble seul les théorèmes qui dépendent (transitivement) du théorème modifié puissent être affecté. Néanmoins nous montrons plus loin que les tactiques implémentant des procédures de décision automatiques (`Auto, Trivial...`) introduisent des liens plus complexes entre les objets et compliquent sensiblement les choses. Ce problème est étudié en détail au chapitre 9.

**6.4.0.0.8.2 Déplacement de code et réorganisation de théories.** Une variante du problème de propagation des dépendances est la réorganisation de théories. Cela consiste à déplacer des fragments de code d'une théorie à une autre.

Pourquoi ?

Les problèmes rencontrés en pratique sont aussi liés au fait que les procédures de décisions automatiques sont sensibles au contexte dans lequel elle sont utilisées.

On peut enfin signaler ici que l'environnement Emacs pour HOL et la notion de "virtual theories" implémenté par Curzon [Cur95c] permettent une réorganisation interactive du code Un "censeur" se chargeant de vérifier le fondement du résultat prouvé par rapport à sa théorie.

**6.4.0.0.9 Exemple 4: Reset Logique** Le Reset Logique est un exemple intéressant d'utilisation des dépendances en mode interactif. C'est le pendant du "Undo-Redö 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 on été definis après lui seront aussi effacés même s'ils ne dépendent pas de lui.

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.



Comme dans le cas du "Undo Logique" les objets effacés seront réécris 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 deplacement des théorèmes entraîne en effet un changement du contexte dans lequel il seront rejoués et cela peut changer le comportement des procédures de décision automatiques. Nous etudions cela en détail au chapitre 9.

Les problèmes rencontrés dans la mise en oeuvre de ces quelques applications et que nous détaillerons au chapitres 8 et 9, nous on conduit à distinguer plusieurs catégories de dépendances.

## 6.5 Catégorie de dépendance

### 6.5.1 Les dépendances chronologiques

Les dépendances chronologiques sont basée sur l'ordre d'apparition des théorèmes et définitions et sur les liens entre les différents fichiers introduit par des commandes comme `Require, load ...` Comme nous l'avons vu au chapitre **??** ces dépendance sont déjà plus ou moins utilisées par les outils d'aide à la réalisation des `Makefile` comme `coqdev` (qui recense les dependances entre fichiers en filtrant certain mots clefs).

En ce qui concerne la maintenance de théorie, ces dépendance devront être prise en compte en cas de modification de théoreme. En effet, du fait des tactiques de decision automatique comme `Auto`, si nous modifions un théorème `T1`, il se peut qu'un théorème `T2` qui était jusque lá indépendant de `T1`, se mette à l'utiliser. Cet exemple est étudié plus en détail au capitre 9.

**6.5.1.0.10 Exemple** Nous constatons qu'en pratique c'est essentiellement pour les problèmes liés au comportement des diverses tactiques automatiques que les dépendances historiques seront essentielles. Ces tactiques automatiques utilisent toutes pour leur recherche de solution une base de données de théorèmes et définitions que nous appellerons "la liste des Hint". Cette liste peut être étendue dynamiquement pendant une session.

L'utilisation dans un script de couple (Hint/Auto) ou (Immediat/Trivial) consiste donc à introduire un context variable ou une notion d'état dans la démarche de preuve.

Le problème est que les tactique d'automatisation sont dépendantes de ce context et toute modification de ce contexte peut entraîner une modification de leur comportement.

Là encore, dans le développement d'outils de maintenance, on contournera le problème en restreignant le contexte (c'est-à-dire la liste des Hint); en pratique on remplacera dans le script les `Auto` par des `Auto with T1 ... Tn` ou `Auto without T1`.

### 6.5.2 Les dépendances logiques

Les dépendances logiques sont celles que nous avons manipulé jusqu'à présent. Elles représentent le fait que la définition d'un objet utilise un autre objet. Remarquons que si un objet A dépend logiquement de B, B doit avoir été défini précédemment et donc A dépend donc aussi historiquement de B. Elle correspond donc à un "affinement" de la notion de dependance historique.

Nous pouvons distinger deux types de dépendances logiques:

1. Les dépendances opaques

   Les dépendances opaques ne prennent en compte que le type des objects, c'est à dire qu'un terme `T1` dépend de manière opaque d' un terme `T2` si il l'utilise mais sans accéder à la structure du terme.

   **6.5.2.0.11 Exemple**

   Du point de vue de la maintenance, cela a pour conséquence 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 du 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.

#### 6.5.2.1 Opaque versus transparent

**6.5.2.1.1 Définitions** Généralement en mathématiques, les théorèmes dépendent les uns des autres par leurs énoncés en 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 leur 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 differents (ils ont des constructeurs dont les noms et les types diffèrent)

Quand la définition d'un objet n'utilise que le type d'un autre objet on parlera de dépendance opaque, si elle utilise en utilise le terme on parlera de dépendance transparente.



Mais il n'est pas toujours évident de déterminer la nature d'une dépendance. Illustrons cela par quelques exemples

`EXEMPLE`

**6.5.2.1.2 Une approximation de la nature des dépendances** 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'utilise 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 à priori si les objets qui l'utilisent, utilisent ou non son terme. Les opérations de réduction des fonctions ($\iota-reduction$) ne laissent pas de trace dans le terme obtenu. On ne peut donc savoir avec certitude la nature des dépendances. Mais une approximation sans risque est de conciderer 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.

On peut traquer dans le script les tactiques qui accède à la structure d'un terme `Simpl, Change,...` mais ce n'est pas très fiable ne serait-ce que parce que l'utilisateur peux définir ces propres tactiques.

L'autre solution est plus coûteuse, elle consiste à rendre le terme opaque et à rejouer le script, chaque fois qu'une preuve échoue, on sait que la dépendance est transparente et on met la définition en axiome pour pouvoir continuer à rejouer le développement. Toutes les fois que le script repasse sans problèmes on sait que la dépendance est opaque.

Ce calcul plus fin sera utilisé dans les outils de réutilisation et de généralisation que nous étudions au chapitre 10

Il est évident que les dépendances opaques sont les plus robustes au changement et que l'utilisateur doit être encouragé à opacifier au maximum ces définitions.

# Chapitre 7

# Une méthode interactive de propagation des modifications



## 7.1 Graphe de dépendances et ordre des modifications

Pour étudier les dépendances entre différents objets d'une théorie mathématique on maintient un graphe de dépendance entre ces objets. Chaque noeud de ce graphe étant une référence à un objet de la théorie (par exemple un nom de théorème) et l'ensemble de ses fils étant une liste de références aux objets qui en dépendent (par exemple une liste de noms de théorèmes utilisant le théorème père).

Lorsque l'on modifie un des objets, il faut pour que la théorie reste valable, modifier en conséquence les objets qui en dépendent.

Un graphe de dépendance est donc un graphe acyclique orientés (DAG). Dans un DAG, la relation binaire du graphe définit un ordre partiel.( elle est transitive et reflexive ). On notera $A \geq_t B$ si $A = B$ ou s'il existe un chemin d'origine $A$ se terminant en $B$. On peut trouver un ordre total $<$, compatible avec cet ordre partiel, Autrement dit on peut exhiber une liste $(S_1 \ldots S_n)$ croissante pour la relation $<$, de tous les sommets du DAG telle que si l'on a $S_j \geq_t S_i$ alors $j \geq i$. Trouver un tel ordre s'appelle faire un tri topologique. Cela permet d'obtenir un ordre de modification des théorèmes. Ainsi sur le graphe de la figure 7.1, on peut obtenir [C; M; N; B; F; E; A; D; G; H; I; J; L; K]. Mais cet ordre n'est pas

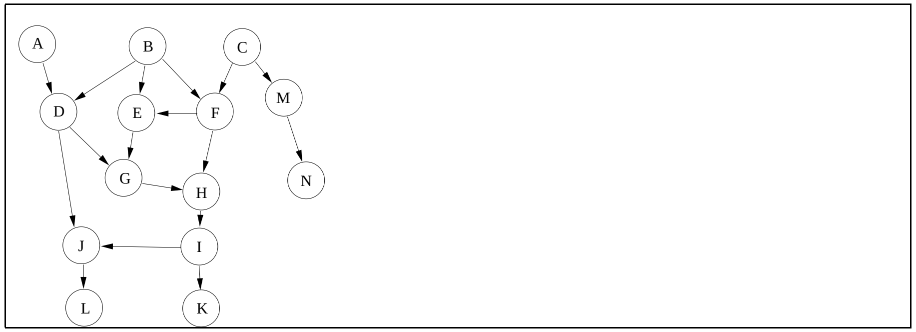



FIG. 7.1 – *Le graphe de départ*

unique et [A; B; C; D; F; E; G; H; I; J; K; L; M; N] convient tout autant. 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 noeud qu'il veut modifier à l'étape suivante et de connaître l'état exact de la propagation des modifications. On étudiera ensuite la situation ou au cours de la propagation d'un modification, l'utilisateur décide de modifier un autre théorème que ceux qui lui sont normalement proposé

## 7.2 Méode génerale

### 7.2.1 Principe

Les noeuds seront annotés par un statut qui peut prendre 4 valeurs:

NORMAL,MODIFIE,DOUTEUX,AMODIFIER

et une date (qui dans l'algorithme est un entier mais sera bien évidemment la date fournie par le système au moment où a lieu la modification).

Le statut `NORMAL` représente les noeuds en l'absence de modification, le statut `MODIFIE` les noeuds qui ont été modifiés, le statut `DOUTEUX` les noeuds qui sont douteux , c'est-à-dire ceux qui dépendent d'un noeud modifié, enfin les statuts `AMODIFIER` représentent les noeuds qui ne dépendent directement que de noeuds modifiés et doivent donc être vérifiés à l'étape courante.

Au départ tous les noeuds sont annotés par `NORMAL` et la date 0. A chaque fois qu'un noeud prend le statut `MODIFIE` sa date augmente de 1.
L'algorithme est le suivant:

1. On modifie `n` noeuds incomparables (La restriction de choix à des noeuds incomparable nous permettra d'avoir la propriété 3 qui affirme que chaque noeud et vérifié est modifié au plus une fois) pour $\geq t$ ;ils sont annotés à MODIFIE

2. Tous leurs descendants dans le graphe sont annotés à `DOUTEUX`

3. Parmi leurs descendants directs, ceux qui n'ont pas de parents `DOUTEUX` sont annotés à `AMODIFIER` et sont proposés à l'utilisateur qui choisit lequel il veut attaquer.

4. Après modification, le noeud choisi à l'étape précédente est annoté à `MODIFIE` et sa date est mise-à-jour, tandis que dans ses descendants directs ceux qui n'ont pas de parent `DOUTEUX`, sont annotés à `AMODIFIER` et pourront donc être attaqués à l'étape suivante.

5. Tant qu'il reste des noeuds à `AMODIFIER` qui peuvent être attaqués on reprend à l'étape précédente.

6. Quant la propagation est terminée, tous les noeuds ayant le statut `MODIFIE` peuvent être remis à `NORMAL`



### 7.2.2 exemple

Au départ fig 7.1 tous les noeuds ont un statut `NORMAL` (en blanc).

Après modification des noeuds `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 `AMODIFIER`(en blanc cerclé de noir) et peuvent être modifiés à leur tour pour maintenir la cohérence. L'utilisateur doit alors décider si il poursuit

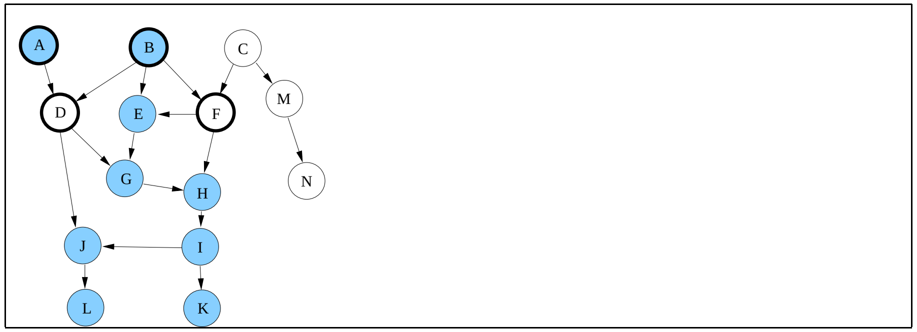



FIG. 7.2 – *Début de traitement après modification de A et B*

en modifiant le noeud `D` ou le noeud `F`. Ici il choisit alors de modifier le noeud `D`, qui prend donc le statut `MODIFIER`. Ses descendants qui n'ont plus de parents `DOUTEUX` deviennent `AMODIFIER` 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.

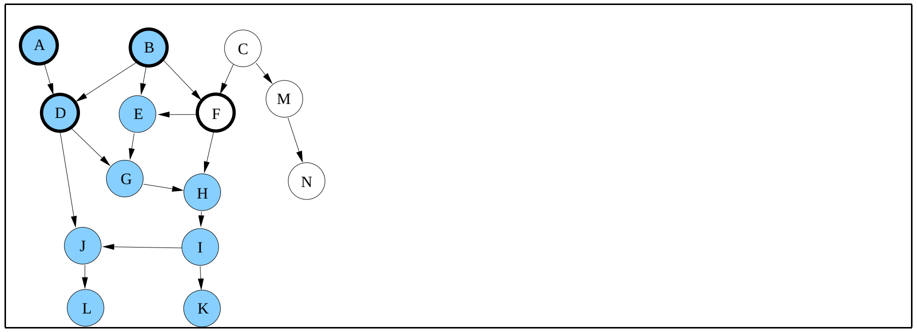



FIG. 7.3 – *Après avoir propagé la modification au noeud D*

Le seul noeud modifiable (`AMODIFIER`) 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 `AMODIFIER`.

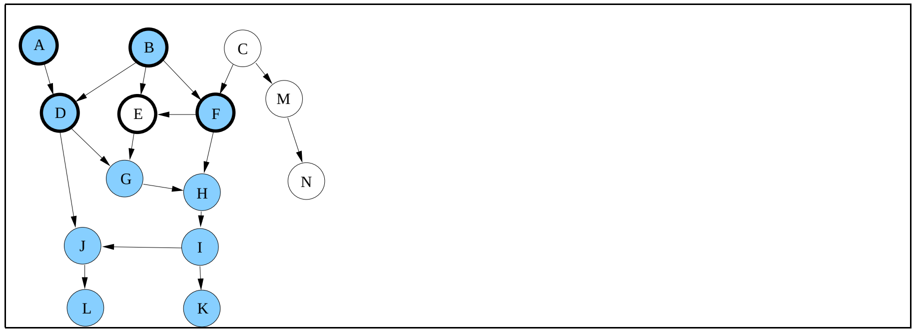



FIG. 7.4 – *Après avoir propagé la modification au noeud F*

On attaque ensuite le noeud `E` qui devient `MODIFIE` tandis que `G` qui n'a plus de parents `DOUTEUX` devient `AMODIFIER`.

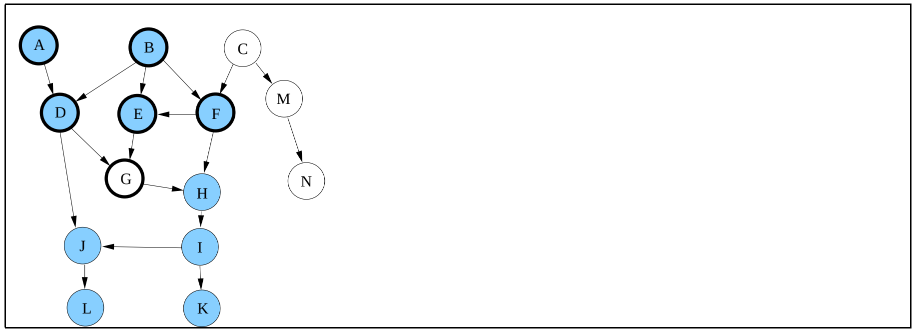



FIG. 7.5 – *Après avoir propagé la modification au noeud E*

On continue en attaquant `G` qui devient `MODIFIE` tandis que son fils `H` qui n'a plus de parents `DOUTEUX` devient `AMODIFIER`.

### 7.2.3 Propriétés

1. Seuls les noeuds qui dépendent effectivement des noeuds de départ sont considérés
2. Aucun noeud n'est modifié plus d'une fois
3. Tout noeud modifié, n'est effectivement modifié qu'après les noeuds dont il dépend ie:
$$date(A[i]) \ < \ date(A[j]) \quad si \quad A[i] \geq_t A[j]$$



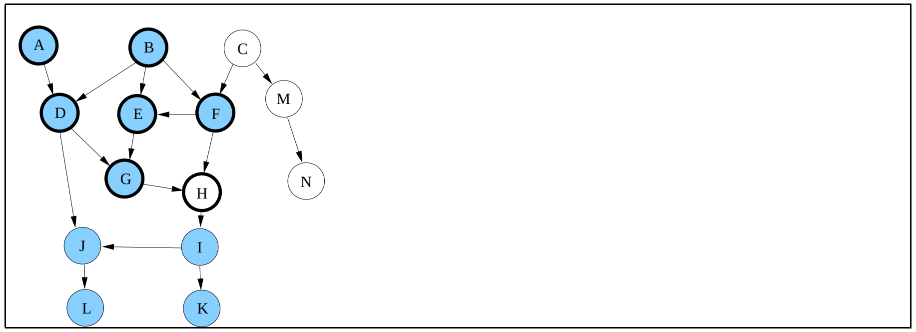



FIG. 7.6 – *Apres avoir propagé la modification au noeud G*

## 7.3 Spécificités des dépendances entre théorèmes

### 7.3.1 Opacité et transparence

Un théorème ou une définition `T1` dépend d'un autre théorème ou définition `T0` si ce dernier apparaît dans son énoncé ou dans sa preuve. Mais T1 peut faire référence ou ne pas faire référence à la structure du terme T0. Ainsi, lorsque qu'un théorème ou une définition n'utilise que le type(ie. l'énoncé) d'un autre théorème ou définition, on parle de dépendance `Opaque`, si il utilise la structure du terme(ie. sa preuve) on parle de dépendance transparente.

Généralement une référence à une fonction est une dépendance transparente Par exemple, pour prouver le lemme suivant sur l'addition,

```
Lemma plus_n_O : (n:nat) n=(plus n O).
```

il faudra connaître la structure de `plus` et pas seulement sont type `nat->nat->nat` qui est le même que celui de `mult`ou `puissance` pour lequel le theorem est faux.

Par contre quand une preuve utilise un théorème précédemment démontrer ce qui est généralement intéressant c'est l'ennoncer de ce théorème (son type) et non sa preuve (le terme de preuve), la dépendance est opaque.

Les dépendances opaques permettent de limiter la propagation des modifications. Et on peut Optimiser notre algorithme pour tenir compte de cela.

### 7.3.2 Adaptation de la méthode précédente

Le principe reste le même, et les premières étapes sont identiques

1. On modifie n noeuds incomparables pour $\geq t$ ;ils sont annotés à MODIFIE
2. Tous leurs descendants dans le graphe sont annotés `DOUTEUX`
3. Parmi leurs descendants directs, ceux qui n'ont pas de parents `DOUTEUX` sont annotés a `AMODIFIER` et sont proposés à l'utilisateur qui choisit lequel il veut attaquer.
4. (a) Si le noeud choisi à l'étape précédente est modifié, il est comme précédemment annoté à MODIFIE et sa date est mise à jour, tandis que dans ses descendants directs ceux qui n'ont pas de parent DOUTEUX, sont annotés à `AMODIFIER` et pourront donc être attaqués à l'étape suivante.
   (b) si le noeud choisi à l'étape précédente n'est pas modifié il est annoté à NORMAL et l'on n'a pas besoin de regarder les noeuds qui ne dépendent que de lui. Tous ces noeuds vont prendre le statut `NORMAL`. Tandis que parmi tous ces autres descendants, ceux qui ont des parents à `DOUTEUX` ou à `AMODIFIER` gardent le statut `DOUTEUX`, ceux qui n'en ont pas mais ont des parents à `MODIFIE` deviennent `AMODIFIER`, les autres qui n'ont que des parents `NORMAL` deviennent `NORMAL`. A chaque fois qu'un noeud est remis a `NORMAL` sa date est mise à jour.
5. Tant qu'ils reste des noeuds à `AMODIFIER` qui peuvent être attaqués on reprend à l'étape précédente.
6. Quant la propagation est terminée, tous les noeuds ayant le statut `MODIFIE` peuvent être remis à `NORMAL`

Les propriétés de l'algorithme de la section précédente sont toutes conservées

### 7.3.3 Exemple

Si après l'étape de la figure 7.6, on attaque `H` mais qu'il apparaît que `H` n'a pas à être modifié, il prend alors le statut Normal. Donc, tous les noeuds qui ne dépendent que de `H` n'ont plus à être considérés, et prennent donc le statut `Normal`. Mais le noeud `J` dépend aussi de `D` qui a le statut `MODIFIE` donc `J` prend le statut `AMODIFIER` et la propagation peut continuer sur cette branche.

## 7.4 Principe d'utilisation

### 7.4.1 Principe de base

1. Modifier l'axiomatique,



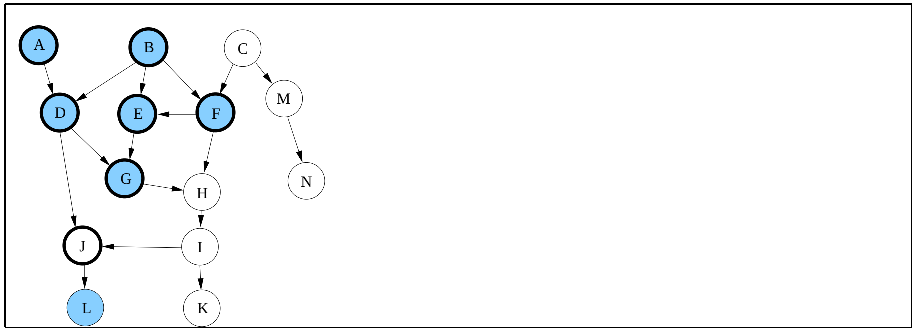



FIG. 7.7 – *Arrêt de la propagation des modifications sur les noeuds qui ne dépendent que de H*

2. Utiliser l'algorithme de propagation pour calculer quels noeud peuvent être modifiés.
3. Donner le choix a l'utilisateur (avec ou sans visualisation des théorème concernés.
4. Se positionner dans l'éditeur sur le théorème choisi
5. Reprendre à l'étape 2

### 7.4.2 Stockage en cour de modification

La Reprise 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 tache de reprise.

Le mieux est de stocker une représentation du graphe avec les noeuds marqués, ce qui permet de reprendre à l'étape 3 lorsque l'on reprend le traitement.

### 7.4.3 Gérer les conflits

Que fait t'on si en cours de propagation de modification l'utilisateur veut effectuer de nouvelle modification.

On peut vérifier de leur indépendance avec les modification pré cédentes, auquel cas il n'y a pas de problèmes on fait un merge des deux graphes.

Mais que faire si les nouvelle modification interviennent avec les anciennes??

## 7.5 Perspective

Des outils de gestion des dépendances dans les systèmes d'aide à la preuve semblent indispensable à la maintenance de grosses preuves. Et peu disposent de mécanismes plus performant que la génération de fichier de description de dépendances entre fichiers sources lisibles par *Make*. Dans *Hol*[GM93], on dispose de la notion de théorie et où un fichier peut contenir plusieurs théories, et le système donne accès aux dépendances entre théories. Mais il n'a rien pour les dépendances entre les théorèmes eux-même. Nous avons commencé a développer un prototype pour gérer et manipuler ces dépendances entre théorèmes dans le système *Coq*. Ce système est basé sur une théorie typée le *Calcul des Constructions Inductives*. L'énoncé d'un théorème est donc un type et une preuve un terme ayant ce type. La construction de la preuve se fait interactivement par un langage de commande (les tactiques) qui permettent de raffiner le but. La liste des commandes qui permettent de construire le terme de preuve par raffinement successif s'appel un script de preuve. En première approximation les dépendances peuvent être calculées en parcourant le terme et en récupérant la liste des constantes qui y apparaissent. On peut alors utiliser l'algorithme de la section précédente pour aider l'utilisateur dans ces tâches de modification et de maintenance. C'est-à-dire qu'ayant modifié un théorème ou une définition dans le script, on aide l'utilisateur à trouver quelles autres parties du script doivent être modifiées pour pouvoir être rejouées, et à le faire dans l'ordre voulu. En fait les dépendances visibles sur le terme de preuve sont insuffisantes, car le script peut contenir des tactiques automatiques qui implémentent des heuristiques de recherche. Ces tactiques peuvent se comporter très différemment après modification d'un théorème et entraîner l'apparition de dépendances qui n'étaient jusqu'alors pas visibles. Une solution à ce problème en tenant compte des dépendances historiques entre les théorèmes (c'est-à-dire de l'ordre dans lequel ils ont été introduits) et en utilisant les spécificités comportementales de chacune des heuristiques, est à l'étude.. Le problème de la gestion des conflit reste ouvert.

# Chapitre 8

# Problèmes liés à la normalisation et à l'élimination des coupures

## 8.1 Script de preuve et objet preuve

Nous Avons vu lors de la construction d'une preuve, l'utilisateur envoie ses tactiques au système qui les utilise pour construire un terme de preuve.

Dans notre calcul de dépendance, nous avons conjecturéăque le terme de preuve contenait plus d'information que le script.

Or dans nos première expérience sur les coupes de théorie,il est apparu que les script des théories minimale que nous avions calculé ne pouvaient pas forcement être rejoué sans problème. Certains objet manquaient. Dés lors une question s'imposait:

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

Nous allons voir que la réponse est non et que cela est liés à l'elination des coupures pour reprendre le langage du calcul des sequents ou à la normalisation du $\lambda$-terme si l'on préfère la terminologie du $\lambda$ calcul.

### 8.1.1 Insuffisance de l'information contenu dans le terme de preuve

On peut constater que lorsque le script contient des tactiques comme `Cut` (ou des tactiques composées à partir de `cut` comme `LApply`), le $\lambda$-terme stocker par le système ne



contient pas forcément tous les identificateurs qui apparaissent dans le script de preuve.

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

mais après que la preuve ai été sauvegardée, le $\lambda$-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
```

En effet, le système garde le terme de preuve sous sa forme normale, et la réduction sous forme normale peut faire disparaître des identificateurs du $\lambda$-terme, comme dans l'exemple précédent.

On peut même rencontre le cas ou un indentificatuer n'apparaît ni dans le script ni dans le terme normalisé mais et utilise dans la preuve comme dans l'exemple ci dessous:

```
(*
Nous illustrons le fait qu'un identificateur peut n'apparaître ni
dans le terme ni dans le script ni dans le terme normalisé mais
apparaître dans le terme non normalisé et donc être nécessaire pour que
le script repasse *)

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

```
Hint titi.

Lemma toto :(A:Prop)A->A.
Cut (x:nat)(x=O)->(pred (S x))=O.
Auto.
Auto.


(*
toto < Show Proof.
LOC:
Subgoals
Proof: ([_:(x:nat)x=O->(pred (S x))=O][A:Prop][H0:A]H0 [x:nat][H:x=O]H)
*)

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

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 corresponde à cette démonstration. Cela revient à éliminer les coupures.

Un réarrangement automatique du script peut être envisagé dans certains cas particuliers mais cela est loin d'être simple et peut de plus entraîner de grosses modifications qui peuvent le rendre incompréhensible à son concepteur.

En pratique il y a surtout un cas ou il est intéressant d'introduire des coupures sans être sur que les résultat introduit dans le contexte facilite la preuve en cours. C'est en liaison avec les tactique automatiques. L'idée est d'enrichir suffisamment le contexte pour que la procédure de décision puisse prouver le résultat courant.

Quand le besoin s'en fera sentir nous contournerons nous le problème en calculant les dépendances sur les termes non $\beta$-réduit.

# Chapitre 9

# Problème lies aux procédures de décisions automatiques

## 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 ou sont prouver les autre théorème. Si les modifications ou déplacement de code ont été faites de manière cohérente avec le graphe de dépendance, cela peut sembler anodin puisque nous savons grâce au calcul du graphe de dépendance comment les objets de notre environnement sont reliés les uns aux autres. Mais en pratique lorsque l'on veut rejouer un script de preuve, de tel modifications du contexte sont pas sans conséquence. En effet les procédure de décision automatique peuvent devenir plus puissante (On sait qu'elle ne devienne pas moins puissante car tout les lemme qu'elle on utilisé avant la modification apparaisse dans le grave de dépendance et on donc été conservé dans le nouveau contexte).

Illustrons le par un exemple:

Nous nous focalisons maintenant sur les problèmes que cela pose dans la propagation de modification mais des difficulté semblable surgisse pour nos outils de déplacement de code et de réorganisation de théorie.

**Le nouveau script est il toujours un sous ensemble de l'ancien?**

**L'ancien script est il suffisant pour adapter le nouveau?** Ou dit autrement, toute tactique est elle une fonction décroissante du pouvoir des procédures de décision automatique (C'est-à-dire grossièrement pour Coq, de la liste des Hint). Nous donnons ici un contre exemple en utilisant conjointement la tactique `Auto` et la tacticale `Orelse`.



## 9.2 Contre exemple

```
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.
Hint th1.
Parameter th : (B H1 H2 H3 ∧ H3 H3 ∧ H3).
Parameters th2 : H2; th4 : H4.

Goal (A (B H1 H2 H3 ∧ H3 H3 ∧ H3) (H4 ∧ H4) ∧ (H4 ∧ H4) True).
1: Split with; Auto;
   (Split with; Auto; Apply λx, y : Prop . λh : x ∧ y . h) Orelse Apply th.
Two sub goals: H4/\H4 ** I **
1: Split with; Apply th4.
1: Split with; Apply th4.


With Hint on th2 and th4 in (** I**) the two
subgoals become H3/\H3.
Qed.
```

Ce contre exemple est une instanciation du schéma plus général suivant:

`le retrouver ou le refaire !!!!!!!!!!`

En Pratique la tacticale `Orelse` semble néanmoins fort peut utilisé (71 fois sur les quelques 118037 ligne des "contrib" de la distribution standard de Coq soit 0,06

Hormis ces cas pathologiques l'utilisation de l'ancien script pour automatiser la modification du nouveau. Les outils de déplacement et de modification de l'arbre de preuve développer dans la première partie de cette thèse pourrons être utilisés. Et même dans les cas pathologique il pourront servir à isoler la partie de preuve à refaire.

## 9.3 Mise en oeuvre de la récupération de script

Nous avons donné dans un précédent chapitre un algorithme interactif aidant a propager les modification du code a partir du graphe de dépendances. En pratique chaque modification change le contexte Donc tout les théorème qui seront joué après un théorème modifié est susceptible de changement

## 9.4 Une mise en oeuvre pratique de la modification de théorie

Dans toute cette section on s'intéresse au "cas simple" sans `Orelse`.

En pratique on pourra faire tourner deux Coq en parallèle. Dans la première on rejouera l'ancienne version de notre code, Dans la seconde on jouera la version en cour de modification. On pourra aussi jouer précédemment l'ancienne version dans un mode "debug" en sauvegardant toute les information utiles sur la structure des preuves.

Après avoir modifier le script de la théorie nous commencerons à le rejouer en propageant les modifications. Pour chaque théorème, le graphe de dépendance nous indiquera s'il dépend d'un théorème modifier dépendance, et doit donc être modifié. Le système nous positionnera alors sur le théorème à modifier dans la fenêtre d'édition. Il essayera de rejouer les autre théorème "théoriquement" indépendant dans le contexte ainsi créé. Si un de ces résultat logiquement indépendant 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 puissance et quelles sont les branches de l'ancien arbre de preuve qui sont désormais résolue automatiquement. Après avoir supprimer les lignes de script correspondant a ces branche désormais inutile et décaler les indices des lignes restantes le script sera rejoué sans encombre. Et passera au théorème suivant.

En fait si l'on veut que les branche devenue obsolette soit éliminer automatiquement, on aura souvent besoin d'un arbre de preuve plus fin que celui décrit dans la première partie de cette thèse. En effet nous considerions un ensemble de tactique composées par l'intermédiaire de la tactiquelle `then` comme un seul noeud, alors que cela peu être vu comme un sous arbres. Ainsi un noeud comme :

```
1:Intros z H1 H2; Elim (H z H2);Auto; Intros E;Rewrite E;Auto.
```

correpond à l'arbre suivant:

```
              Intros z H1 H2.
              (Elim H x H2)
     Auto        Auto          Auto
   intros E                  intros E
   Rewrite E                 Rewrite E
    Auto                       Auto
```

# 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 $\mathbf{N}$ 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 nouvelle 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 typé.

La base théorique de la généralisation est l'anti-unification [Hue75, Pfe91] (`# Ca suppose deux terme a antiunifier...et ca les papiers sont vraiment hard?`)

D'autres approches ont néanmoins été proposées [KW94] qui propose d'abstraire les termes fonctionnels et c'est plus dans ce sens que nous nous orientons dans ce travail.



## 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 l'addition des entiers naturels:

```
Lemma plus_permute : (n,m,p:nat) ((plus n (plus m p))=(plus m (plus n p))).
Proof.
Intros; Rewrite (plus_assoc_l m n p); Rewrite (plus_sym m n); Auto.
Qed.
```

On peut démontrer la même propriété pour la multiplication.

```
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.
```

Intuitivement cette propriété restera vrai pour toute fonction associative et commutative sur les entiers. Regardons le terme de preuve et essayons de l'utiliser pour fournir la preuve d'une généralisation de notre théorème.

```
Proof: [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))
```

On constate en effet que les seules propriétés de la multiplication qui soient utilisées sont la symétrie et l'associativité à gauche qui correspondent aux identificateurs `mult_sym` et `mult_assoc_l`.

On recherche donc leurs énoncés (c'est-à-dire le type associé à ces identificateurs):

```
generalize_permute < Check mult_sym.
mult_sym
     : (n,m:nat)(mult n m)=(mult m n)

generalize_permute < Check mult_assoc_l.
mult_assoc_l
     : (n,m,p:nat)(mult n (mult m p))=(mult (mult n m) p)
```

Il reste à abstraire dans le théorème le nom de la fonction et ses propriétés, il vient :

```
Lemma generalized_permute :
(n,m,p:nat)
(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))
((f n (f m p))=(f m (f n p))).
```

que l'on prouve en réutilisant le terme de preuve précédent:

```
Intros.
Exact
        (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))
```

On a maintenant le terme de preuve suivant:

```
Proof: [n,m,p:nat]
        [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)]
           (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))
```

### 10.2.1 Où l'on abstrait encore !!!

Mais il semble que cette propriété reste vraie pour toute fonction `f:E->E` associative et commutative, quel que soit l'ensemble `E`. Essayons de continuer notre généralisation:

```
Lemma more_generalized_permute :
(E:Set)
(n,m,p:E)
(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))
((f n (f m p))=(f m (f n p))).
```





Que l'on prouve en réutilisant le terme de preuve précédent:

```
Do 7 Intro.
Exact
        (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))
```

D'ou le terme de preuve complet:

```
Proof: [E:Set]
        [n,m,p:E]
         [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)]
            (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))
```

### 10.2.2 Où l'on pousse à l'extrême (c'est-à-dire un peu trop loin!)

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).`
Le théorème reste donc vrai pour toute relation `R` vérifiant ce principe d'où la généralisation suivante:

```
Lemma  more2_generalized_permute :
(E:Set)
(n,m,p:E)
(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)))
(R E(f n (f m p)) (f m (f n p))).
```

que l'on prouve en réutilisant le terme de preuve précédent:

```
Do 9 Intro.
Exact
        (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)).

more2_generalized_permute < Show Proof.
LOC:
Subgoals
Proof: [E:Set]
        [n,m,p:E]
         [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))]
              (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))
```

### 10.2.3 l'instanciation

On vérifie que l'on peut instancier cette généralisation:

```
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!
```

## 10.3 Robustesse et problèmes

### 10.3.1 Problèmes généraux

En fait, on ne peut garantir que notre généralisation d'un théorème `T` fonctionnera si sa preuve ne fait référence qu'aux types des identificateurs généralisés et n'utilise pas leurs structures internes. On dit alors que la dépendance vis-a-vis de ces termes est opaque.



Malheureusement pour prouver les propriétés de bases des fonctions on a souvent 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 définie par de telles règles s'appelle la $\iota$-réduction.

Kolbe et Walter [KW94] ont etudié la réutilisation de telles preuves, mais cela semble se limiter à de petits exemples et semble 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 => O
(S p) =>  m  + ( p * m))
```

mais pas si on la définit par le système

```
 O => O
(S p) =>( m * p) + m)
```

qui est pourtant équivalent.

### 10.3.2 Autres problèmes !

Quand on veut abstraire un identificateur associé à un type inductif, plusieurs cas sont possibles. Soit il est utilisé de "façons muette" dans le terme où on l'abstrait, c'est-à-dire que l'on ne s'intéresse pas à la structure de données qu'il définit (c'est le cas de `nat` dans notre exemple) et l'abstraction peut se faire sans risque. Soit le terme où l'on fait l'abstraction contient un sous-terme correspondant à un raisonnement sur la structure inductive (Induction, raisonnement par cas . . . ) et l'abstraction devient beaucoup plus compliquée.

Dans le cas d'un raisonnement par induction, on va retrouver dans le terme de preuve des identificateurs correspondant aux récurseurs associés aux types inductifs. Ces identificateurs peuvent a leur tour être abstraits (comme on le voit dans notre exemple) mais cela ne semble pas très interressant d'un point de vue pratique.

Les constructions primitives de coq sont basées sur `Fix` et `Case` sur lesquels sont construits les recurseurs.

Si l'on fait un raisonnement par cas, et non par induction, ce que l'on retrouvera dans le terme n'est pas un récurseur mais une définition fonctionnelle par cas.

Ainsi sur le `lemme 1`, on utilise un Case

```
Lemma l1 :(n:nat) (n=n).
```

```
Intros.
Case n;Auto.
```

Il vient le terme de preuve suivant dans lequel on ne peut pas abstraire `n`

```
l1 =
[n:nat]
 <[n0:nat]n0=n0>Case n of
    (refl_equal nat O)
    [n0:nat](refl_equal nat (S n0))
    end
     : (n:nat)n=n
```

Si on fait la preuve par induction

```
lemma l1':(n:nat) (n=n).
Induction n;Auto.
```

il vient

```
l1' =
Subgoals
Proof: [n:nat]
        (nat_ind [n0:nat]n0=n0 (refl_equal nat O)
          [n0:nat][_:n0=n0](refl_equal nat (S n0)) n)
```

dans lequels on peut abstraire n:

```
Lemma l2: (E:Set)(O1:E)(S1:E->E)
(E_ind:( P:E->Prop)(P O1)->((n:E)(P n)->(P (S1 n)))->(n:E)(P n))
(n:E)
        (n=n).

Do 4 Intro.
Exact
       [n:E]
        (E_ind [n0:E]n0=n0 (refl_equal E O1)
          [n0:E][_:n0=n0](refl_equal E (S1 n0)) n)
```

En fait l'abstraction sur un type inductif ne semble avoir un intérêt pratique que dans le cas où il correspond à "un type muet". Dans les autres cas l'abstraction correspondrait à un "sur-typage", notion qui en coq peut être manipulée aux moyen des fonctions de coercions d'A Saïbi.



## 10.4 Idée de l'implémentation

- On fournit le nom des fonctions ou termes à abstraire. Dans l'environnement `Ctcoq`, cela peut se faire par sélection souris.
- On recherche son type `T`
- On récupère la liste des identificateurs qui apparaissent dans le terme.
- On recherche le type qui leur est associé
- On ne garde que ceux dont le type contient le ou les identificateurs correspondant aux termes à abstraire.
- On associe à chacun des identificateurs précédemment sélectionnés un nouvel identificateur $f_i$ et un type équivalent dans lequel les $f_i$ on été substitué aux noms des identificateur correspondant.
- On abstrait les fonctions $f_i$ de type $T_i$ dans l'énoncé du théorème.
- Pour chaque propriété $p_i$ d'une fonction abstraite $f_i$, on crée une nouvelle propriété $f_iP_i$ dont le type correspond au type de $p_i$ dans lequel on a substituer les $f_i$ aux identificateurs abstraits.
- On abstrait les propriétés crées à l'étape précédente.
- On substitue dans le terme de preuve les $f_i$ et les $f_iP_i$ aux noms des fonctions et propriétés abstraites.

## 10.5 Restriction dans la forme des théories

Pour pouvoir utiliser dans un développement les notions de généralisation que nous venons d'esquisser, ce développement devra respecter un certain nombre de règles. En fait on dans l'état actuel on peut disposer d'un outils de généralisation de preuve pour un développement fait de façons entièrement équationelle en utilisant seulement la réécriture.

## 10.6 Génération de théorie abstraite

L'idée principale est d'éviter de repartir de zéro lorsque l'on va développer des théories abstraites comme par exemple la théories de groupes. En effet dans les développement existant de nombreuse preuves n'utilise que n'utilise que quelques proprietes des objets sur lesquelles elles travaillent.

On peut concevoir de founir un mode debug dans leqel chaque propriete utilise dans les preuve sur des fonction sera sera generaliser et stocker.

Lorsque par la suite on aura à définir une structure, on donnera ses propriétés abstraite et en comparant avec notre stockage on pourra récupérer gratuitement dans les développements existants tout les théorèmes qui sont valable pour notre structure.

#### 10.6.0.0.3 Exemple

## 10.7 conclusion et perspectives

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.). La generalisation devrai permettre de recurer en les genéralisant les preuve qui ont été faite pour des instance de ces theorie comme les entiers ou les polynômes.

Pour les développements moins directement lies aux structures mathématiques, mais plus orientés vers les preuves de spécifications logiciels, la généralisation pourra aider à organiser et structurer un gros projet, en centralisant certains raisonement fondamentaux. Cela aidera a réduire la taille des développents, et en facilitera la compréhension, la maintenance et la réutilisation

# Chapitre 11

# Conclusion

Nous avons montrer l'importance des activités de maintenance pour les systèmes de preuve. Nous nous somme focalisé sur la maintenance des scripts de preuve. La notion de script de preuve et les notions connexes de tactique et de tactiquelle issues de LCF sont en effet communes à de nombreux assistants de preuves. Nous avons fournis des outils assistant l'utilisateur dans la maintenance et la transformation de ces scripts.

Ces outils s'appuient sur la notion fondamentale de dépendances.

Nous avons discerné plusieurs niveaux de dépendance: entre théories (fichiers), entre objets d'une théorie, et entre différentes parties d'une preuve.

Toute ces notions sont assez génériques néanmoins la manière dont les dépendance sont calculées est relativement dépendante du système utilisé. Dans les système 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 basés sur une logique classique, pourra par exemple utiliser la sauvegarde sous forme de règles primitives.

Les problèmes que nous avons mentionnés à propos des procédures de décisions automatiques se retrouvent sous des formes différentes dans la plupart des systèmes.

# Annexes

# Annexe A

# Une interface utilisant daVinci et Caml

## A.1 Interfacer daVinci et Caml

### A.1.1 Une Idée de l'implementation

**A.1.1.0.4 Introduction** On décrit ici une interface *Ocaml* [WL94] au logiciel de visualisation de graphe *daVinci* [Frö97, MF96] développé à Université de Brême.

Le but 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 noeuds/arcs, modification des caractéristiques d'un noeud/arcs, vue multiple, navigation ...).

La structure interne des graphes de *daVinci* n'étant pas aisée à manipuler, on maintient notre propre structure de graphe en *ML* (par exemple sous forme de liste d'adjacences) qui est compilé vers celle du visualisateur quand le besoin s'en fait sentir. Sur cette base on peut écrire en *ML* à peu près toutes les fonctionnalitées voulues. Nous illustrons l'utilisation de ces outils au travers de divers petits exemples puis par la réalisation d'une interface graphique interactive pour l'assistant de preuve Coq (qui est écrit en *Ocaml*).

**A.1.1.0.5 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 [Ous94]). L'affichage peut être optimisé en tentant de minimiser le nombre de croisements des arrêtes. L'algorithme utilisé est une variante de celui de Sugiyama [SM91, KS81].

**A.1.1.0.6 Le type abstrait graphe** Un graphe sera représenté en *Ocaml* par une liste de couples `(sommet,liste de fils)`. En pratique, pour optimiser certaines manipulations on purra utiliser une structure doublement liée `(sommet,(liste de fils,liste de parents))`. On pourrait aussi utiliser une structure matricielle.



**A.1.1.0.7 Communication daVinci-Ocaml** La connection entre ocaml et daVinci se fait par pipe. La figure ci dessous, issue de la documentation en ligne de *daVinci* en illustre le principe.

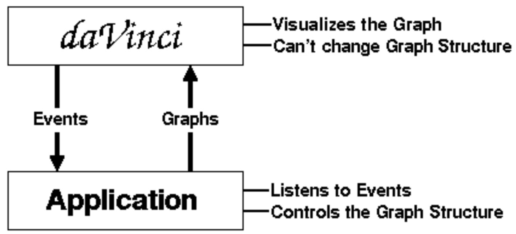



FIG. A.1 – *Communication* `daVinci-Ocaml`

Elle se réalise via deux pipes UNIX: un pour envoyer les commandes aux API de *daVinci* , un pour recevoir les réponses. Cote caml nous gérons la connections en utilisant les possibilités de la bibliothèque Unix [Ler92].

**A.1.1.0.8 L'envoi de message à daVinci** L'envoie 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'ecrivant sur le bon canal. Puis on utilise cette fonction pour implementer toute les fonctionnalités des API de *davinci*.

**A.1.1.0.9 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 a la sélection d'un noeud).

Cette vérification se fait grâce a 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"])` signal que le noeud identifier par "4" a été selectioné.

Nous faisons donc l'analyse syntaxique de ces chaînes pour en récupère le message et les éventuels arguments.

**A.1.1.0.10 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 entré au clavier et des messages reçus du visualisateur.

Le module permetant d'utiliser le package (`Reaction`) est donnée en annexe

**A.1.1.0.11 Utilisation du package** Pour utiliser ce package caml de manipulation de graphe, il faut d'abord 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. Elle doit ressembler a cela:

```
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 () ;;
```



**A.1.1.0.12 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 noeud crée un nouveau noeud ; `AddEdge` qui nous met dans un mode ou l'on peut ajouter des arêtes, chaque "click impair" selectionant un noeud source et chaque "click pair" le noeud cible ; `CutSubgraph` qui permet de couper le sous graphe issu du noeud que l'on sélectionne. Enfin `NoInteraction` qui nous met dans un mode ou toutes les sélections sont inactives. La fonction de réaction `reactionInterface` gère les réactions en fonction du mode dans lequel on se trouve. La fonction `reactionText` ne fait rien.

(Les sources se trouvent dans le module Interface)

**A.1.1.0.13 Affichage de dépendances entre fichier**

## A.2 Une interface graphique pour *Coq*

On présente maintenant un prototype très rudimentaire d'interface graphique pour le système *Coq*. La plupart des fonctionnalités que nous implémentons sont liées á la notion de dépendance. Nous n'entrerons pas néanmoins dans les difficultés théoriques de la notion de dépendance dans les systèmes de preuve. Pour une étude plus complète on pourra consulter [Pon97, PBR98]

### A.2.1 Affichage et manipulation des Arbre de preuve

Nous court-circuitons le "top-level" de *Coq*, la fonction de traitement du texte entré au clavier, reconnaît et traite les commande *Coq* . On considère qu'une commande est complète dés qu'un point (le terminateur de commande Coq) est entré. On transmet alors la commande au parseur puis à l'évaluateur de *Coq* qui répond en affichant le résultat de la commande. 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 noeud 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 noeud correspondant.

### A.2.2 Affichage du graphe de coercion

Le sous typage n'existe pas en Coq. Pour remédier à cet inconvénient, on dispose d'un mécanisme de coercion implicite[]. Le système maintient un graphe des coercions.

Lorsque l'on développe de grosses théories notamment en Algèbre, il peut devenir difficile savoir de quelle coercion on dispose. On se propose ici de visualiser le graphe de coercion. 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.

### A.2.3 Affichage de l'environnement

Grossièrement, l'environnement est une liste d'identificateurs auquel on peut associer les $\lambda$-termes qui leur correspondent. Comme le script d'un preuve l'environnement est classiquement la liste des objets définis. Un calcul des dépendances entre ces objets permet de munir cet environnent d'une structure de arborescente A partir d'un terme nous calculons donc,par une simple analyse syntaxique, de quel identificateur il dépend. Nous pouvons alors maintenir un graphe de dépendance entre les identificateurs. Le graphe peut alors être visualisé. L'interface graphique peut aussi être utilisé pour "nettoyer l'environnement".

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 objet qui on été définie après lui. Nous nous proposons d'utiliser la structure de graphe de l'environnement pour fournir un outils de nettoyage moins brutal permettant d'effacer un objet et tous ce qui en dépendant sans toucher au reste de l'environnement.

On se propose de gérer les dépendances de la session courante. On maintiendra deux niveaux de visualisation. Dans le premier, tous les objets introduits depuis le début de la session seront représente ainsi que tous les objets "extérieurs" qu'ils utilisent. Les objets "extérieurs" sont ceux qui sont définis dans d'autre fichiers qui preuves être importés dans le développement courant. Des que le développement devient trop général , le niveau de visualisation devient inutilisable. On regroupera donc tout les objet d'un même fichier en un seul objet.

`picture........`

Dans l'implemementation pour *Coq*, les chose se complique un peu a cause de mécanisme de section. Il est en effet impossible d'effacer un objet d'un section qui a était fermé, et on est obligé d'effacer toute la section. On vera donc pour certaines applucation tout les objects d'une section comme un seul et meme object. (* a verifier c'est ce que je comprend de leur code mais ca n'as pas l'aire vrai en pratique *)

### A.2.4 Conclusion

Notre package offre la possibilité de développer rapidement en *Ocaml* des prototypes d'interface nécessitant la visualisation de graphe.

# Bibliographie


[ABD96] Sten Agerholm, Ilya Beylin, and Peter Dybjer. A comparison of hol and alf formalizations of a categorical coherence theorem. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 17–32, Murray Hill, NJ, USA, August 1996. Springer.

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third International Symposium on Programming Language Implementatio n and Logic Programming*, New York, August 1991. Springer-Verlag.

[BBC+96] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. *User Guide to the CtCoq Proof Environment*. INRIA, Feb 1996.

[Ber98] Yves Bertot. A proved compiler for imperative language, 1998. draft.

[BKT94] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, April 1994. Springer-Verlag LNCS 789.

[BS] Richard Bornat and Bernard Sufrin. Jape - a framework for building interactive proof editors. Available at http://www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.shtml.

[BS91] Urban Boquist and Jan Sparud. Sml-make: Incremental software development in standard ml under gnu emacs. Masters thesis, Department of Computer Science, Chalmers University of Technology, June 1991.

[BSS] Yves Bertot, Thomas Schreiber, and Dilip Sequeira. Proof by pointing in a weakly structured context. ftp://ftp.dcs.ed.ac.uk/pub/lego/pbp/main.ps.





[BT98] Yves Bertot and Laurent Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 22, 1998.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CH90] Robert Constable and Douglas Howe. NuPrl as a general logic. In P. Odifreddi, editor, *Logic and Computation*. Academic Press, 1990.

[CKT95] Y. Coscoy, G. Kahn, and L. Théry. Extracting text from proof. In M. Dezani and G. Plotkin, editors, *Proceedings of the TLCA 95 Int. Conference on Typed Lambda Calculi and Applications*, volume 902. Springer-Verlag LNCS, April 1995.

[Cou97] Judicael Courant. A module calculus for pure type systems. In R. Hindley, editor, *Proceedings fo the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Nancy, France, April 1997. Springer-Verlag LNCS.

[Cur95a] P. Curzon. Tracking design changes with formal machine-checked proof. *j-COMP-J*, 38(2):91–100,???? 1995.

[Cur95b] Paul Curzon. The importance of proof maintenance and reengineering. In Jim Alves-Foss, editor, *International Workshop on Higher Order Logic Theorem Proving and Its Applications: B-Track: Short Presentations*, pages 17–32, 1995.

[Cur95c] Paul Curzon. Virtual theories. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 138–153. Springer-Verlag, 1995.

[dB68] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *spe*, 9(4):255–65, April 1979.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, 1989. (Available as Technical Report MS-CIS-89-53).

[Fel93] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, 1993.

[FGNT94] W. M. Farmer, J. D. Guttman, M. E. Nadel, and F. J. Thayer. Proof script pragmatics in IMPS. In A. Bundy, editor, *Automated Deduction—CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 356–370. Springer-Verlag, 1994.

[FGT90] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System (system abstract). In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 653–654. Springer-Verlag, 1990.

[FGT92a] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: System description. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 701–705. Springer-Verlag, 1992.

[FGT92b] W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

[FGT93a] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

[FGT93b] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730-1420, USA, November 1993.

[FH94a] A. Felty and D. Howe. Generalization and reuse of tactic proofs. *Lecture Notes in Computer Science*, 822:1–??, 1994.

[FH94b] A. Felty and D. Howe. Tactic theorem proving with refinement-tree proofs and metavariables. *Lecture Notes in Computer Science*, 814:605–??, 1994.

[FH94c] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 1–15, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.

[FH94d] Amy Felty and Douglas Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 605–619, Nancy, France, June 1994. Springer-Verlag LNAI 596.



[Frö97] M. Fröhlich. *Incremental Graphlayout in the Visualization System daVinci (in german language)*. PhD thesis, Department of Computer Science; University of Bremen, November 1997.

[GHG+93] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and 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.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Number 78 in Lecture Notes in Computer Science. Springer Verlag, 1979.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP: A proof development environment. In Alan Bundy, editor, *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.

[Hor92] A. Le Hors. Graph: A directed graph displaying server, GIPE 2 Esprit project, 4th review report, workpackage 4, 1992.

[Hue75] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, June 1975.

[INR94] INRIA. *Centaur 2.0*, 1994.

[JB93a] Ian Jacobs and Janet Bertot, editors. *Centaur 1.2*, chapter The Metal Manual. Inria Sophia-Antipolis, 1993.

[JB93b] Ian Jacobs and Janet Bertot, editors. *Centaur 1.2*, chapter The PPML Manual. Inria Sophia–Antipolis, 1993.

[JR92] Ian Jacobs and Laurence Rideau. A centaur tutorial. Technical Report 140, INRIA, July 1992.

[Kam] Florian Kammüller. Comparison of imps, pvs and larch with respect to theory treatment and modularization. http://www.cl.cam.ac.uk/users/fk203/papers.

[KS81] M. Toda K. Sugiyama, S. Tagawa. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2):109 – 125, July/August 1981.

[KW94] T. Kolbe and C. Walther. Reusing proofs. In *Proc. of the 11th ECAI*, pages 80–84, Amsterdam, The Netherlands, 1994.

[LAD+94] Panagiotis K. Linos, Philippe Aubet, Laurent Dumas, Yann Hellebоid, Patricia Lejeune, and Philippe Tulula. Visualizing program dependencies: An experimental study. *Software—Practice and Experience*, 24(4):387–403, April 1994.

[Ler92] Xavier Leroy. Programmation du système unix en caml light. Rapport technique 147, INRIA, 1992.

[Ler95] Xavier Leroy. The caml light system (release 0.7). Projet Cristal, INRIA, 1995.

[Léo93] Jean-Michel Léon. Displaying graphs with centaur, bull research france, Janvier 1993.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. How to use LEGO. Technical Report LFCS-TN-27, University of Edinburgh, Edinburgh, Scotland, oct 1989.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, jun 1990.

[Mag93] Lena Magnusson. Refinement and local undo in the interactive proof editor ALF. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 191–208, Nijmegen, The Netherlands, 1993.

[Mag95] 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, January 1995.

[ME93] Ursula Martin and Jeannette M. Wing (Eds.). *First International Workshop on Larch, Dedham 1992*. Workshops in Computing. Springer-Verlag, 1993. ISBN 3-540-19804-0/ISBN 0-387-19804-0.

[MF96] M. Werner M. Fröhlich. davinci v2.0 online documentation, 1996. Available at http://www.informatik.uni-bremen.de/~davinci/doc_V2.0.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.



[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Ous94] John K. Ouserhout. *Tcl and the Tk Toolkit*. Professional Computing series. Addison-Wesley, 1994.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau93a] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pau93b] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[PAULIN-MOHRING89] C. PAULIN-MOHRING. *Extraction de programmes dans le calcul des construction*. These de Doctorat, Université Paris 7, 1989.

[PBR98] Olivier Pons, Yves Bertot, and Laurence Rideau. Notions of dependency in proof assistants. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1998"*, Sophia-Antipolis, France, 1998. Available at `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, January 1990.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Pon97] Olivier Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997. Available at http://www.inria.fr/croap/-events/uitp97-papers.html.

[PP98] Loic Pottier and Olivier Pons. dependtohtml :creating hypertext graphical representation of directed graphs, 1998. Available at http://www.inria.fr/croap/personnel/Loic.Pottier/Dependtohtml-/README.html.

[Rei92] W. Reif. The kiv system: Systematic construction of verified software. In D. Kapur, editor, *Automated Deduction: CADE-11 - Proc. of the 11th International Conference on Automated Deduction*, pages 753–757. Springer, Berlin, Heidelberg, 1992.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[Rou] Christophe Roudet. Trfl.

[RSC95] John Rushby and David W. J. Stringer-Calvert. A less elementary tutorial for the PVS specification and verification system. Technical Report SRI-CSL-95-10, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995.

[RT88] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: a system for constructing language based editors*. Springer Verlag, 1988. (third edition).

[SB94] Tom Schubert and John Biggs. A tree-based, graphical interface for large proof development. In Thomas F. Melham and Juanito Camilleri, editors, *Proceedings of the 7th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'94)*, September 1994.

[SHB+98] Jörg Siekman, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, and Volker Sorge. A distributed graphical user interface for the interactive proof system. In *Proceedings of the International Workshop "User Interfaces for Theorem Provers 1998 (UITP'98)"*, Eindhoven, Netherlands, 1998.

[SM91] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21(4):876 – 892, July/August 1991.



[SMO92] Hajime Sawamura, Toshiro Minami, and Kyoko Ohashi. Euodhilos: A general reasoning system for a variety of logics. In *Proceedings of the International Confrence on Logic Programming and Automated Reasoning (LPAR 92),St. Petersburg, Russia*, volume 624. SpringerVerlag LNCS, 1992.

[SOR93] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[ste96] Andrew stevens. Toward mechanisd revision of proofs and theories. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1996"*, University of York, 1996.

[Sym95] D. Syme. A new interface for HOL – ideas, issues and implementaion. In E. T. Schbert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[TBK92] Laurent Théry, Yves Bertot, and Gilles Kahn. Real Theorem Provers Deserve Real User-Interfaces. *Software Engineering Notes*, 17(5), 1992. Proceedings of the 5th Symposium on Software Development Environments.

[Thé97] Laurent Théry. Proving and computing: a certified version of the buchberger's algorithm. Technical Report 3275, INRIA, Oct 1997.

[Thé93] Laurent Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic theorem proving and its applications*, volume 780 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1993.

[UW94] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, February 1994.

[Val93] Pierre Valentin. Legolang: Une interface graphique pour le systeme de preuve lego. Master's thesis, Universite de Nice, 1993. Also available by anonymous ftp to barar.inria.fr:pub/croap/BraTypes:interf-lego.ps.gz.

[Wei81] Mark Weiser. Program slicing. In *icse5*, March 1981.

[WG98] M. Huisman W.O.D. Griffioen. A comparison of pvs and isabelle/hol. Technical Report CSI-R9810, Computing Science Institute of the University of Nijmegen, 1998.

[Win90] Phillip J. Windley. Using hol inside emacs. Technical Report CS-90-01, Department of Computer Science University of Idaho, December 90.

[WL94] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, ISBN 2-7296-0493-6, 1994.

[Won95] W. Wong. Recording and checking hol proofs. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 353–368. Springer, Berlin,, 1995.

[YCM78] S. S. Yau, J. S. Collofello, and T. M. MacGregor. Ripple effect analysis for software maintenance. In *Proc. Second International Computer Software and Applications Conference, (COMPSAC '78)*, pages 60–65, Nov 1978.

[Zam97] V. Zammit. A comparative study of Coq and HOL. In E. L. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 323–337, Murray Hill, NJ, USA, August 1997. Springer.