Deuxième partie

Etude global des théorèmes

BROUILLON BROUILLON BROUILLON BROUILLON B

BROUILLON BROUILLON BROUILLON BROUILLON B

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

BROUILLON BROUILLON BROUILLON BROUILLON B

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

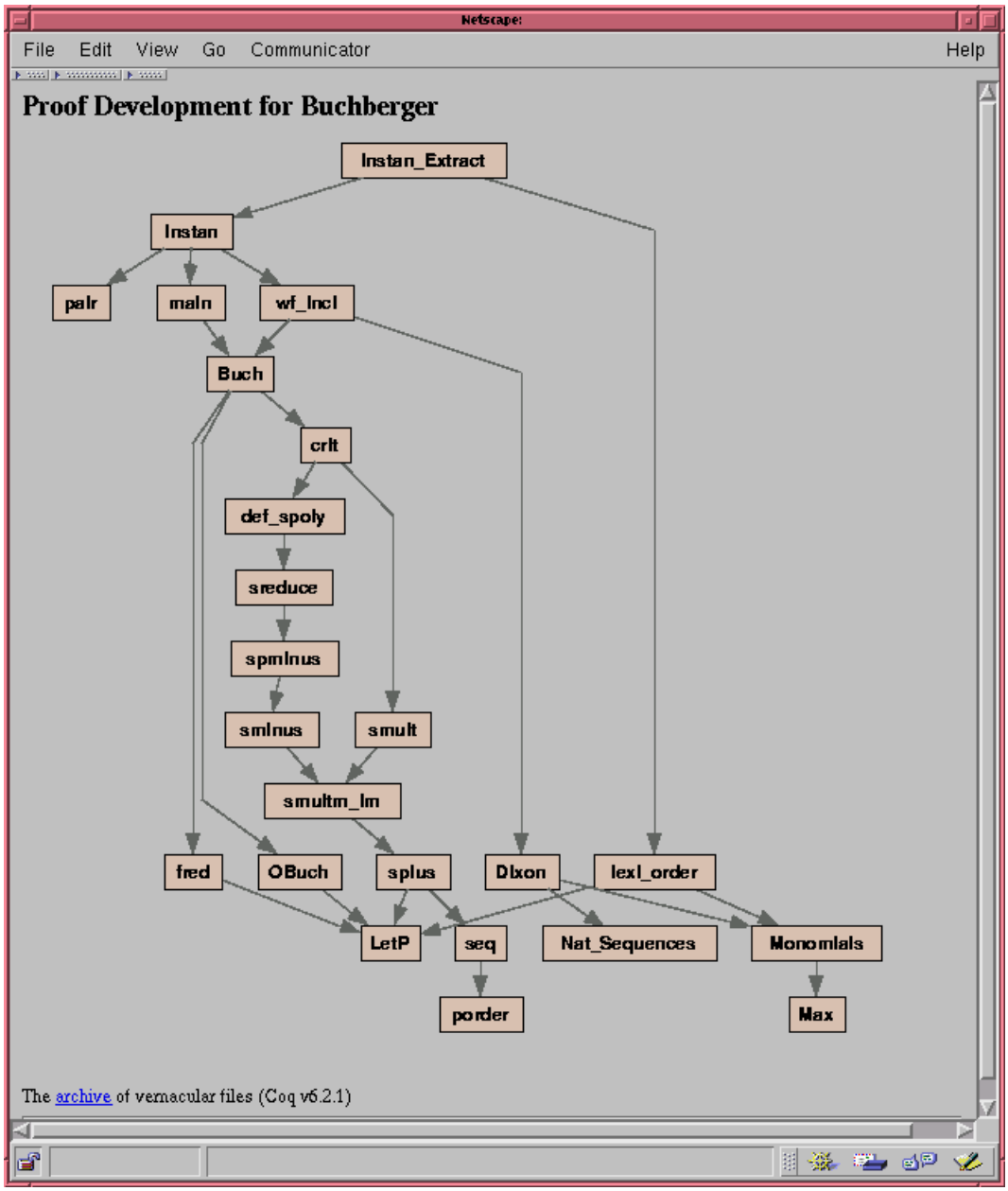
L'outil `Coqdep`

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.

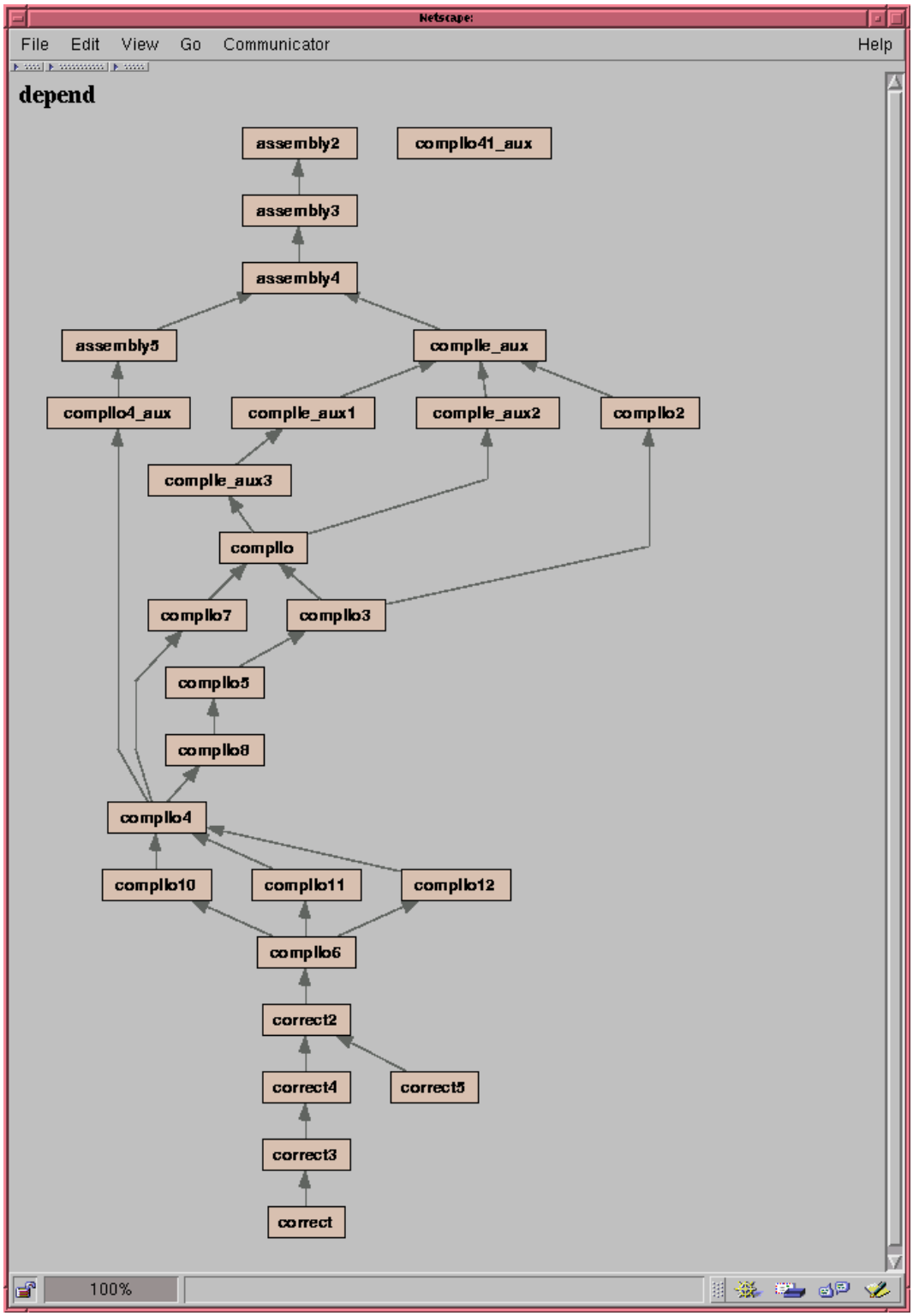
D'abord un developement dû à Laurent Théry [Thé97] conduisant à une preuve de l'algorithme de Buchberger pour le calcul de bases de Gröbner:

BROUILLON BROUILLON BROUILLON BROUILLON B



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

BROUILLON BROUILLON BROUILLON BROUILLON B



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].

BROUILLON BROUILLON BROUILLON BROUILLON B

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 d'un développement sur les anneaux:

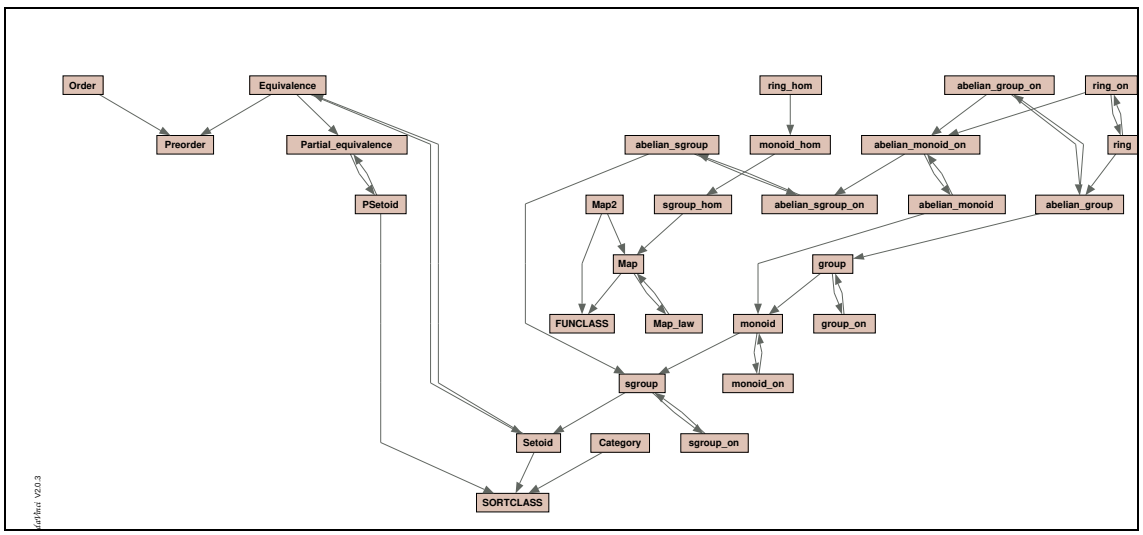


FIG. 5.1 – *Le graph de coercion de la théorie des anneaux*

BROUILLON BROUILLON BROUILLON BROUILLON B

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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

BROUILLON BROUILLON BROUILLON BROUILLON B

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

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

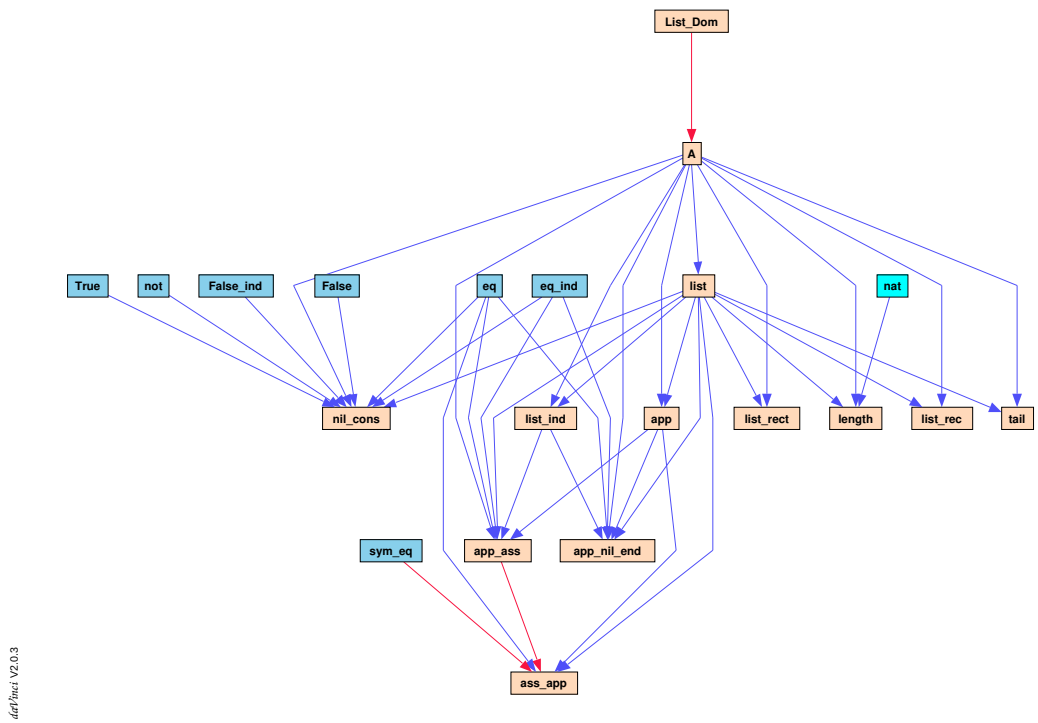
– app_ass, qui correspond à l'associativité à droite de la concaténation.

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.



BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

6.4 Quelques exemples d'applications utilisant le graphe de dépendance

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 theóreme 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

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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`.

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

Chapitre 7

Une méthode interactive de propagation des modifications

BROUILLON BROUILLON BROUILLON BROUILLON B

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 $(S_1 \dots S_n)$ 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

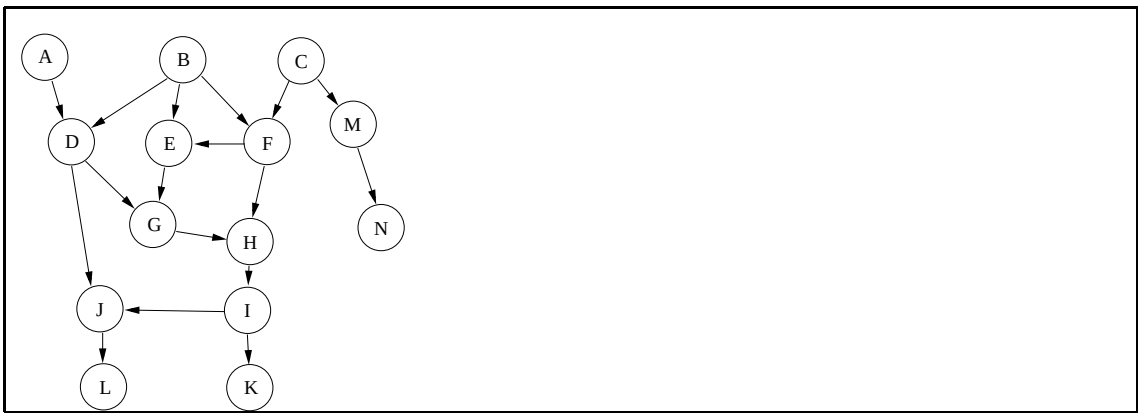


FIG. 7.1 – *Le graphe de départ*

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 proposés.

BROUILLON BROUILLON BROUILLON BROUILLON B

7.2 Méthode générale

7.2.1 Principe

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

tt NORMAL, `MODIFIE`, `DOUTEUX`, `AMODIFIER`

et une date correspondant à la dernière modification.

L'ensemble est dit cohérent si pout tout

$$\forall i, j,\ A[i] \geq_t A[j] \ \Rightarrow\ date(A[i]) \ <\ date(A[j])$$

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 qui dépendent directement ou non d'un noeud modifié, enfin les statuts `AMODIFIER` représentent les noeuds qui dépendent directement d'au moins un noeud modifié et n'ayant pas de parent douteux. Les noeuds ayant le statut `AMODIFIER` sont ceux qui peuvent être vérifiés à l'étape courante.

Au départ tous les noeuds sont annotés par `NORMAL` et le système est cohérent.. A chaque fois qu'un noeud prend le statut `MODIFIE` sa date est mise-à-jour.
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 est 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`

BROUILLON BROUILLON BROUILLON BROUILLON B

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

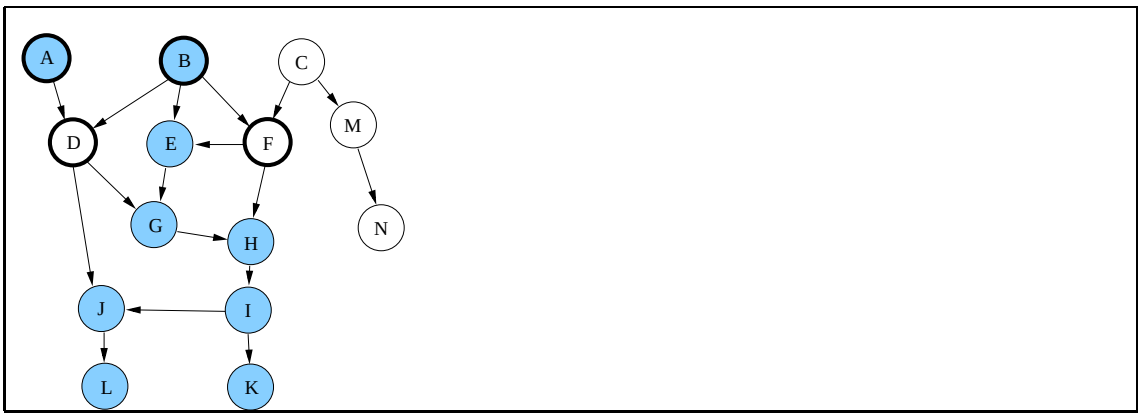


FIG. 7.2 – *Début de traitement après modification de A et B*

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.

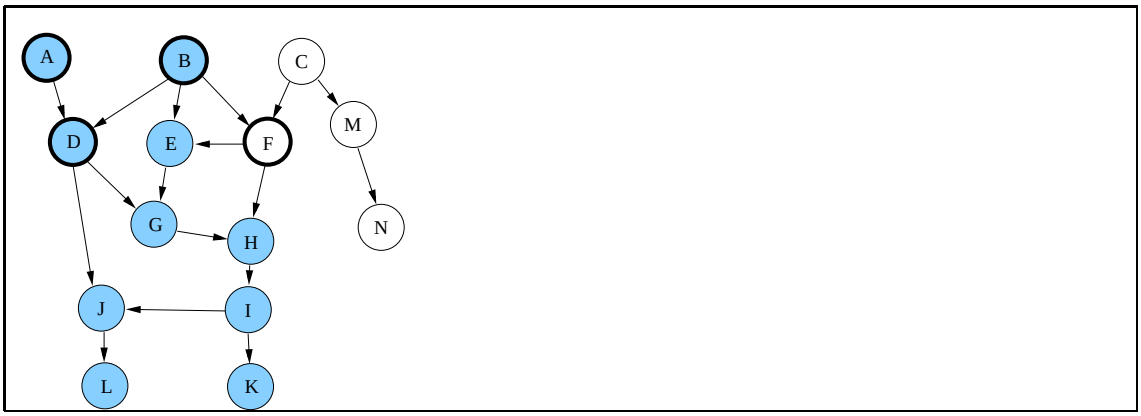


FIG. 7.3 – *Après avoir propagé la modification au noeud D*

Le seul noeud modifiable (`AMODIFIER`) reste `F`, on le modifie donc à son tour. Il prend

BROUILLON BROUILLON BROUILLON BROUILLON B

le statut `MODIFIE` tandis que le seul de ses fils qui n'a pas de parents `DOUTEUX`, (`E`) devient `AMODIFIER`.

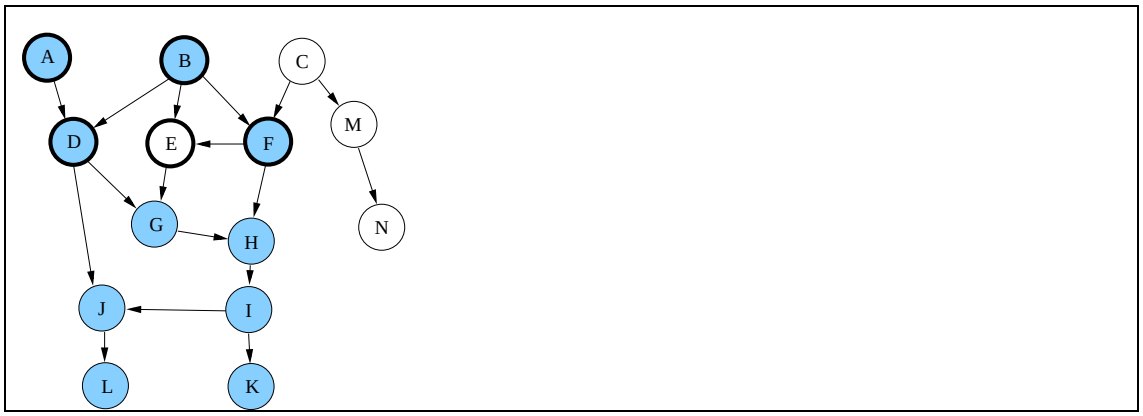


FIG. 7.4 – *Après avoir propagé la modification au noeud F*

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

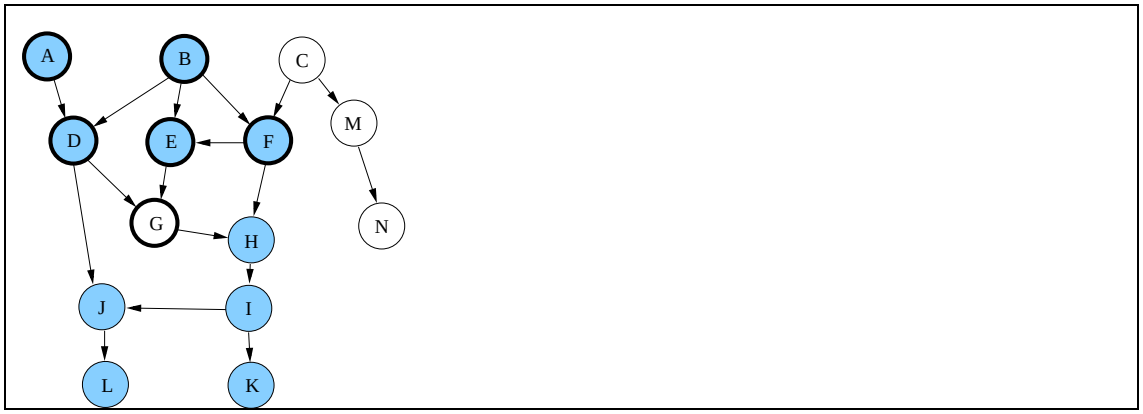


FIG. 7.5 – *Après avoir propagé la modification au noeud E*

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

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. La cohérence du systeme est preservée. C'est-à-dire que tout noeud modifié, n'est effectivement modifié qu'après les noeuds dont il dépend ie:

$$date(A[i]) \ < \ date(A[j]) \quad si \quad A[i] \geq_t A[j]$$

BROUILLON BROUILLON BROUILLON BROUILLON B

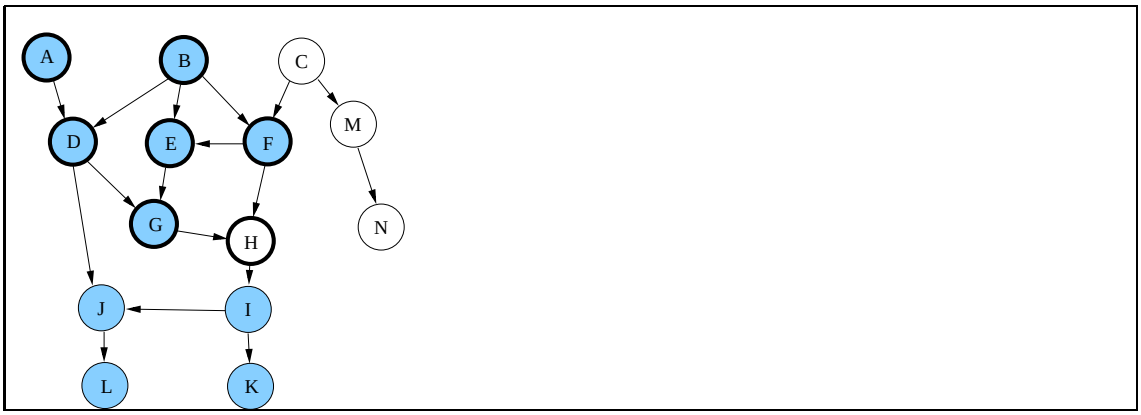


FIG. 7.6 – *Apres avoir propagé la modification au noeud G*

7.3 Spécificités des dépendances entre théorèmes

Nous avons introduit à la section 6.5.2.1 les notion d'opacité et de 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`, s' il utilise la structure du terme(ie. sa preuve) on parle de dépendance transparente.

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

7.3.1 Adaptation de la méthode précédente

Le principe reste le même, et les étapes 1 à 3 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

BROUILLON BROUILLON BROUILLON BROUILLON B

descendants, ceux qui ont des parents à `DOUTEUX` ou à `AMODIFIER` gardent le statut `DOUTEUX`, parmis les autre, ceux qui 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'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`

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

7.3.2 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.

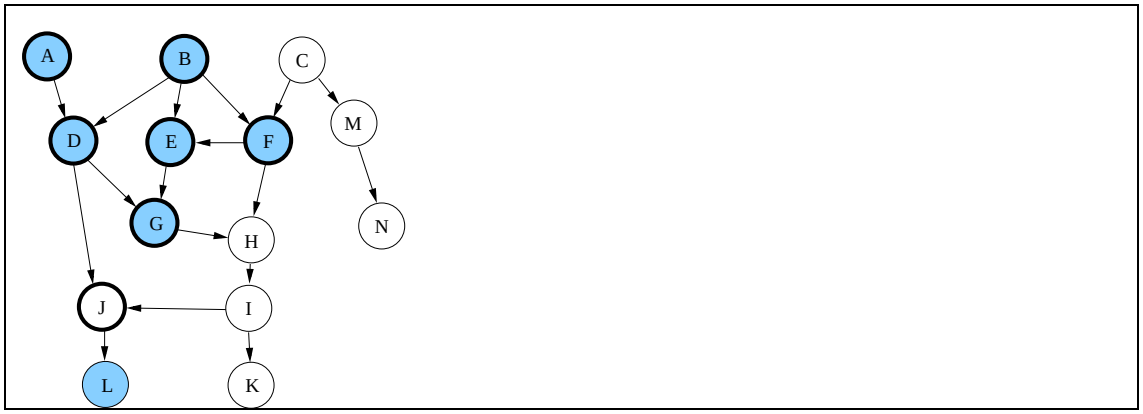


FIG. 7.7 – *Arrêt de la propagation des modifications sur les noeuds qui ne dépendent que de H*

7.4 Principe d'utilisation

7.4.1 Principe de base

1. Modifier l'axiomatique,

2. Utiliser l'algorithme de propagation pour calculer quels noeud peuvent être modifiés.

BROUILLON BROUILLON BROUILLON BROUILLON B

3. Donner le choix à 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 nouvelles modifications.

On peut vérifier de leur indépendance avec les modification précédentes, auquel cas il n'y a pas de problèmes on fait une fusion des deux graphes.

Mais que faire si les nouvelles modifications interfèrent 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. Mais peu de systeme disposent de mécanismes plus performant que la génération de fichier de description de dépendances entre fichiers sources lisibles par *Make*. Dans *Hol*[GM93],ou l' on dispose de la notion de théorie et où un fichier peut contenir plusieurs théories, 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 est 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

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

BROUILLON BROUILLON BROUILLON BROUILLON B

Chapitre 8

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

8.1 Script de preuve et objet preuve

Nous avons vu que 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 supposéăque le terme de preuve contenait plus d'information que le script.

Or dans nos premières expériences sur les coupes de théorie, il est apparu que les scripts des théories minimales que nous avions calculés ne pouvaient pas forcement être rejoués sans problème. Certains objets 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 lié à l'elimination des coupures pour reprendre le langage du calcul des séquents ou à la normalisation du λ-terme si l'on préfère la terminologie du λ calcul.

8.1.1 Insuffisance de l'information contenue 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 λ-terme stocké par le système ne contient pas forcément tous les identificateurs qui apparaissent dans le script de preuve. Comme le montre l'exemple de la figure 8.1

BROUILLON BROUILLON BROUILLON BROUILLON B

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 ait été sauvegardée, le λ-terme est réduit et la constante `trivial-lemma-1` n'y apparaît plus

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

FIG. 8.1 – *Un exemple de "perte d'information" lors de la normalisation*

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 λ-terme, comme dans l'exemple précédent.

On peut même rencontrer le cas où un indentificateur n'apparaît ni dans le script ni dans le terme normalisé mais est utilise dans la preuve comme dans l'exemple ci dessous:

```
(*
l' identificateur 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,y,z:nat)(plus x y)=(plus x z)->y=z.
Induction x;Simpl;Auto.
Save.
```

BROUILLON BROUILLON BROUILLON BROUILLON B

```
Hint titi.

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

L'identificateur `titi` n'apparaît pas dans le script. Il apparaît bien dans le terme non normalisé. On peut aussi noter que le terme de preuve contient encore une variable existentiel `?539` qui devrait être instancier si nous voulions stocker le terme sous sa forme non normalisée.

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

La réduction fait "disparaître " la variable existentielle le terme peut donc être sauver sous sa forme réduite. De plus après reduction, l'identificateur `titi` n'apparait plus dans le terme.

```
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 correspondre 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 où il est intéressant d'introduire des coupures sans être sûr que les résultat introduits dans le context 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 le problème en calculant les dépendances sur les termes non β-réduit après avoir instancier les éventuelles variables existentielles par un objet du bon type. Il reste un cas pathologique très rare pour lequel on ne pourra stocker le terme non normalisé„ celui ou la variable existentielle appartient a un type vide.

BROUILLON BROUILLON BROUILLON BROUILLON B

BROUILLON BROUILLON BROUILLON BROUILLON B

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 dans lequel les autres théorèmes sont prouvés. Si les modifications ou déplacements 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 ne sont pas sans conséquences. En effet les procédures de décision automatique peuvent devenir plus puissantes (On sait qu'elles ne deviennent pas moins puissantes car tous les lemmes qu'elles ont utilisés avant la modification apparaissent dans le graphe de dépendances et ont donc été conservés dans le nouveau contexte).

Illustrons le par un exemple:

```
TO DOOOOOOOOOOOOOOOOOOOOO
```

FIG. 9.1 – *Changement de contexte invalidant un script*

Nous nous focalisons maintenant sur les problèmes posés par la propagation de modifications mais des difficultés semblables surgissent pour nos outils de déplacement de code et de réorganisation de théories.

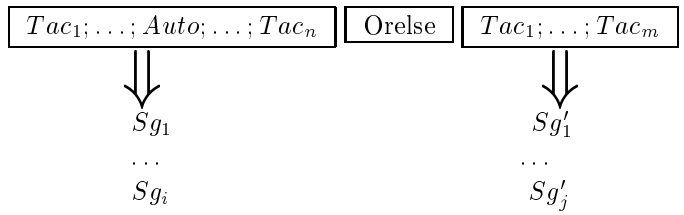
Le nouveau script est il toujours un sous ensemble de l'ancien ?

L'ancien script est il suffisant pour adapter le nouveau ? Ou autrement dit, toute tactique est elle une fonction décroissante du pouvoir des procédures de décision

BROUILLON BROUILLON BROUILLON BROUILLON B

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 tactiquelle `Orelse`.

Le schéma général du problème est le suivant suivant:



Dans le contexte le plus faible la partie gauche du `Orelse` échoue, C'est donc la partie droite qui est exécuté produisant un ensemble de sous-buts $Sfaible = \{Sg'_1 \ldots Sg'_j\}$ qui peut être vide. Si l'on rejoue ce script dans un contexte plus fort, La partie gauche peut ne pas échouer, elle produit alors un ensemble de sous-buts $Sfort = \{Sg_1 \ldots Sg_i\}$. Les deux ensembles de sous-buts $Sfort$ et $Sfaible$ peuvent très bien est totalement disjoints.

Voyons maintenant une instanciation de ce schéma sur un cas concret que nous détaillons.

On fait les déclarations suivante.

```
Inductive A: Prop -> Prop -> Prop ->Prop :=
   A_intro: (a, b, c:Prop) a -> b -> c ->(A a b c).

Inductive B: Prop -> Prop -> Prop -> Prop ->Prop :=
   B_intro: (a, b, c, d:Prop) a -> b -> c -> d ->(B a b c d).
Parameters H1, H2, H3, H4:Prop.
Parameter th1:H1.
Parameter th:(B H1 H2 H3 /\ H3 H3 /\ H3).
Parameters th2:H2; th4:H4.
```

On se place dans le contexte ou seule le `th1` est présent dans la liste de `Hint` et peut donc être utilise par la tactique `Auto`

```
Hint th1.
```

On joue le script suivant:

```
Goal (A (B H1 H2 H3 /\ H3 H3 /\ H3) (H4 /\ H4) /\ (H4 /\ H4) True).
Split; Auto.
 (Split; Auto; Apply [x, y:Prop] [h:x /\ y]h) Orelse Apply th.
```

BROUILLON BROUILLON BROUILLON BROUILLON B

A ce stade on obtient deux fois le sous-but `H4/\H4` et on peut finir la preuve par

```
Split; Apply th4.
Split; Apply th4.
```

Detaillons maintenant ce qui se passe:
La partie gauche du `Orelse` a échoué. En effet le premier `Split` produit trois sous-buts:

```
3 subgoals

  ============================
   (B H1 H2 H3/\H3 H3/\H3)

subgoal 2 is:
   (H4/\H4)/\H4/\H4
subgoal 3 is:
   True
```

Comme `th` et `th4` ne sont pas dans la liste des `Hint`, `Auto` ne peut résoudre que le troisième.

On continu sur le première sous but, le second `Split` casse `B` en quatre sous-buts

```
 H1
subgoal 2 is:
 H2
subgoal 3 is:
 H3/\H3
subgoal 4 is:
 H3/\H3
```

tt H1 est résolue par `Auto` mais pas `H2` puisque þ2 n'est pas dans la liste des hints. On essaye donc d'appliquer `Apply [x, y:Prop] [h:x / y]h)` à `H2` ce qui échoue puisque `H2` n'est pas une conjonction. on oublie donc la sous preuve qui vient d'échouer et on applique la partie droite soit `Apply th.` qui résout le sous-but (B H1 H2 H3/$\tilde{3}$ H3/$\tilde{3}$).

Sur le second sous-but, la partie gauche du `Orelse` se contente de casser le sous but `(H4/`$\tilde{4}$`)/`$\tilde{4}$`/`$\tilde{4}$ en `H4/`$\tilde{4}$ et `H4/`$\tilde{4}$.

Si maintenant nous enrichissons le contexte par

```
Hint th2.
Hint th4.
```

et que l'on rejoue le même script on obtient deux sous-buts:

```
2 subgoals
```

BROUILLON BROUILLON BROUILLON BROUILLON B

```
  ============================
H3/\H3

subgoal 2 is:
 H3/\H3
```

En effet le premier `Split` et le premier `Auto` se comporte de la même façon que précédemment. Le second `Split` casse le premier sous but restant comme précédemment mais `H1` et `H2` sont résolues par `Auto`. `Apply [x, y:Prop] [h:x / y]h)` s'applique donc sur les sous-buts $\texttt{H3}/\widetilde{3}$ qu'il laisse inchangés.

Le second `Split` casse le second sous-but comme précédemment en $\texttt{H4}/\widetilde{4}$ qui est maintenant résolu par `Auto` puisque `H4` a été ajouté àla liste des hints.

En pratique la tactiquelle `Orelse` semble néanmoins fort peut utilisée (71 fois sur les quelques 118037 lignes des "contrib" de la distribution standard de Coq soit 0,06 % du code).

Hormis ces cas pathologiques nous essayons d'utiliser l'ancien script pour automatiser la modification du nouveau. Les outils de déplacement et de modification de l'arbre de preuve développés dans la première partie de cette thèse pourront être utilisés. Et même dans les cas pathologiques ils pourront servir à isoler la partie de preuve à refaire.

9.2 Mise en oeuvre de la récupération de script

Nous avons donné au chapitre 7.1 un algorithme interactif aidant à propager les modifications du code à partir du graphe de dépendances. Mais pratique chaque modifications change le contexte et donc tout les théorème qui seront rejoué après un théorème modifié sont susceptibles de changements.

On pourra se prémunir contre ce problème en autorisant à modifier "à la volée" la liste des théorèmes utilisés par les procédures automatiques. On pourra par exemple associer a chaque théorème une liste de `Hint` réduite par exemple $Restricted_list =$

$$Hint_List \cap identificateur_dans_le_terme$$

et remplacer tout les `Auto` (ou seul ceux qui posent problème lorsque l'on veut rejouer le script) par `Auto with` $Restricted_list$.

Notons qu'un methode plus brutale consisterait à remplacer les procédure de décision automatique par le code qu'elle applique effectivement. Nous avons plusieurs objections à cela:

1. On perd en generalité. Par exemple les scripts de preuve de propriétés les listes peuvent souvent être repris par un simple copier-coller pour prouver des propriété équivalente sur les vecteur (ou liste de longueur n).

BROUILLON BROUILLON BROUILLON BROUILLON B

2. Cela peut rendre le script beaucoup plus volumineux et donc en compliquer la compréhension.

3. ...

9.3 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 modifié 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 modifié , 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 autres théorèmes "théoriquement" indépendant dans le contexte ainsi créé. Si un de ces résultats logiquement indépendants 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 supprimé les lignes de script correspondant à ces branches désormais inutiles et décalé les indices des lignes restantes le script sera rejoué sans encombre. Et passera au théorème suivant.

En fait, si l'on veut que les branches devenues obsolètes soient éliminées 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 considérions un ensemble de tactiques composées par l'intermédiaire des tactiquelles `then`, `Or` ou `Orelse` comme un seul noeud, alors que cela peuy être vu comme un sous-arbres. La figure 9.2 donne un exemple de telle décomposition.

Une telle décomposition est maintenue de façon interne par le système Coq pendant la durée de la preuve et pourra donc être sauvegarder lorsque l'on jouera une preuve en "mode debug". Quand on rejoue un script et qu'une ligne de script comprenant des tactique composé avec `Auto` en utilisant le combinateur `Then`, il est important de savoir a combien de sous but `Auto` s'applique, lesquels il résout, et sur lesquels il échoue.

9.3.1 Conclusion

BROUILLON BROUILLON BROUILLON BROUILLON B

Ainsi dans un script comme celui-ci (issue de la distribution standard `théories/ZARITHM/fast_integer.v`).

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

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

On n'a qu'un seul noeud mais on peut le décomposer pour obtenir:

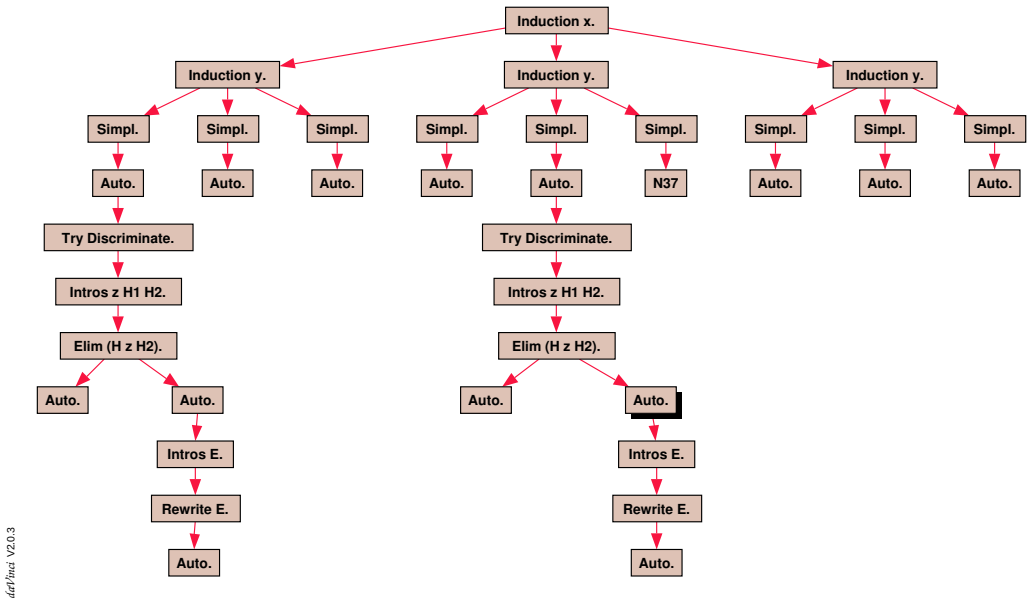


FIG. 9.2 – *Décomposition d'un noeud de l'arbre*

BROUILLON BROUILLON BROUILLON BROUILLON B

Chapitre 10

Généralisation et réutilisation de preuves

10.1 Introduction

Comme le mathématicien, l'utilisateur d'un système de preuve a parfois envie, voire besoin, de généraliser les résultats obtenus pour pouvoir les réutiliser dans d'autres développements. Pour cela, il faut montrer que les théorèmes prouvés restent vrais dans un contexte plus faible. Par exemple, les résultats de base concernant l'addition dans **N** peuvent être généralisés à tout ensemble muni d'une structure dont les lois internes vérifient les axiomes de groupe abélien.

Il est donc important de comprendre quelle partie d'un raisonnement peut être généralisée en vue d'une réutilisation ultérieure, de comprendre les analogies entre preuves et comment cela peut être exploité pour aider au développement de 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 typée.

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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 vraie 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)
```

BROUILLON BROUILLON BROUILLON BROUILLON B

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))).
```

BROUILLON BROUILLON BROUILLON BROUILLON B

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'où 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))).
```

BROUILLON BROUILLON BROUILLON BROUILLON B

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.

BROUILLON BROUILLON BROUILLON BROUILLON B

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 `(0+x)` en `x`. La réduction d'un terme fonctionnel définie par de telles règles s'appelle la ι-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 :

```
 0 => 0
(S p) =>  m  + ( p * m))
```

mais pas si on la définit par le système

```
 0 => 0
(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 "façon 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 à 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).
```

BROUILLON BROUILLON BROUILLON BROUILLON B

```
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.

BROUILLON BROUILLON BROUILLON BROUILLON B

10.4 Implémentation

- L'utilisateur fournit le nom des fonctions ou termes à abstraire. Dans l'environnement `Ctcoq`, cela peut se faire par sélection souris.
- Le systeme recherche alors son type `T`
- Il récupère la liste des identificateurs qui apparaissent dans le terme.
- Il recherche le type qui leur est associé
- Il ne garde que ceux dont le type contient le ou les identificateurs correspondant aux termes à abstraire.
- Il 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 identificateurs correspondants.
- Il 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, il 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.
- Il abstrait les propriétés crées à l'étape précédente.
- Il 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 dans l'état actuel on peut disposer d'un outil de généralisation de preuve pour un développement fait de façon 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 théorie de groupes. En effet dans les développements existant de nombreuses preuves n'utilisent que quelques propriétés des objets sur lesquelles elles travaillent.

On peut concevoir de fournir un mode debug dans lequel chaque propriété utilisée dans les preuves sur des fonctions sera sera generalisé et stocké.

BROUILLON BROUILLON BROUILLON BROUILLON B

Lorsque par la suite on aura à définir une structure, on donnera ses propriétés abstraites et en comparant avec notre stockage on pourra récupérer gratuitement dans les développements existants tout les théorèmes qui sont valables 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 généralisation devrait permettre de récupérer, en les généralisant, les preuves qui ont été faite pour des instances de ces théories 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 raisonnement fondamentaux. Cela aidera à réduire la taille des développent, et en facilitera la compréhension, la maintenance et la réutilisation.

BROUILLON BROUILLON BROUILLON BROUILLON B