Ingénierie de preuve

Olivier Pons[1,2]

1 : CNAM, Laboratoire CEDRIC
292, rue St Martin
F - 75141 Paris Cedex 03 - France

2 : INRIA Sophia Antipolis, Projet Lemme
2004, route des Lucioles - BP 93
06902 Sophia Antipolis - France
`Olivier.Pons@sophia.inria.fr`

Résumé

L'objet de cet article est de présenter un panorama d'outils d'analyse et de manipulation de preuves formelles. Le but des outils proposés est de faciliter le développement et la maintenance de théories dans les assistants de preuve afin d'améliorer la productivité des utilisateurs de ces systèmes.

Introduction

Les environnements de développement, les outils de gestion de versions, ainsi que les outils d'analyse et de manipulation de programmes issus du génie logiciel sont désormais couramment utilisés dans le cadre des langages de programmation. Il est unanimement admis qu'ils permettent d'améliorer le confort et la productivité des programmeurs. D'un autre côté, les systèmes d'aide à la démonstration tels que Coq[12], Lego[13], HOL[10], PVS[18], ou Nuprl[6] connaissent une utilisation croissante et la nécessité d'en faciliter l'emploi, en fournissant des outils se rapprochant de ceux utilisés en programmation, est généralement reconnue. Des environnements de développement de preuves commencent à être utilisés et, bien que pour l'instant marginaux, ils permettent d'accroître la productivité de l'utilisateur et l'autorisent à s'attaquer à des preuves qui, sinon, resteraient difficilement envisageables. Par exemple une preuve de l'algorithme de Buchberger a été effectuée par Laurent Thery [20] sous l'environnement CtCoq[2]. CtCoq est une interface utilisateur pour Coq basée sur le système Centaur[11, 5] mis au point à l'INRIA Sophia pour développer des environnements génériques de programmation.

L'objet de cet article est de proposer un atelier pour analyser et manipuler des preuves formelles construites en utilisant des assistants de preuve. La construction des preuves à l'aide de ces systèmes est une tâche relativement proche de l'activité de programmation. Ainsi, lorsque l'on développe de grosses preuves on rencontre des problèmes similaires à ceux rencontrés dans le développement de gros programmes. Nous nous sommes donc naturellement inspirés des idées et solutions développées pour la manipulation de programmes. La mise en œuvre pratique de tels outils ne peut se faire qu'en s'appuyant sur une "bonne" interface utilisateur. Une implémentation partielle de ces outils a été réalisée dans l'environnement CtCoq pour le système Coq.

Dans la première section de ce document, nous présentons rapidement le cadre de notre travail. Nous développons les notions de système de preuve et de preuve dirigée par les buts et montrons différentes façons de représenter les preuves. Nous survolons ensuite les caractéristiques des interfaces utilisateur utiles à notre travail.

Dans la seconde section, nous introduisons la notion de dépendances entre les différentes parties d'une preuve[1]. Nous présentons des outils de gestion et de manipulation de preuve. Les fonctionnalités principales que nous proposons sont l'aide à la compréhension (visualisation et navigation de preuve), le retour-arrière dans la construction d'une preuve, la contraction et factorisation de preuve et l'isolement de fragments de preuve. Nous discutons aussi dans cette partie les problèmes posés par la présence de variables existentielles dans les buts et des contraintes que cela introduit.

Enfin la dernière partie introduit la notion de dépendance entre objets d'une théorie puis présente rapidement les outils de gestion de théorie que nous proposons. Les fonctionnalités principales sont : la visualisation du graphe de dépendance, la coupe de théorie, le déplacement de code et la réorganisation de théorie, l'aide à la propagation de modifications. Nous abordons les problèmes sous-jacents au calcul des dépendances et à la sensibilité au contexte de certaines tactiques.

Nous concluons par un bilan de l'expérience de mise au point ainsi que par une discussion sur le premier retour issu des utilisateurs et sur les améliorations, extensions et perspectives futures.

1. Cadre de ce travail

1.1. Représentation des preuves dans les systèmes de preuve interactifs

On s'intéresse dans ce document aux systèmes dans lesquels la preuve est dirigée par les buts. Dans de tels systèmes l'utilisateur commence par donner le but qu'il désire prouver puis le transforme au moyen de *tactiques*. Ces tactiques peuvent être des règles de base de la logique sous-jacente au système, des règles dérivées ou des procédures de décision automatique. L'application d'une tactique sur un but produit une liste de sous-buts ou renvoie un message d'erreur (on dit alors qu'elle échoue). Lorsque la liste produite est vide on dit que le but sur lequel la tactique a été appliquée est prouvé. Si cette liste n'est pas vide on doit prouver les sous-buts qu'elle contient par application de nouvelles tactiques. A chaque étape ce sont donc les buts qui conditionnent les règles applicables.

Arbre de preuve : Ainsi la preuve a naturellement une structure arborescente. Cette représentation est classique dans des formalismes comme le calcul des séquents, la déduction naturelle. On parle alors d'arbres de dérivation. Dans les systèmes on emploie plus volontiers le terme d'arbres de preuve.

Script de preuve : D'un autre côté, ce que l'utilisateur conserve de sa preuve c'est la liste des tactiques qu'il a entrées. On appelle cette liste un *script de preuve*. Lorsque plusieurs sous-buts ont été générés, l'utilisateur peut généralement choisir lequel il désire attaquer en premier. Le script caractérise donc la manière dont se construit l'arbre de preuve. Si ce choix n'est pas possible, le script se résume à un parcours gauche-droite et en profondeur d'abord de l'arbre de preuve, les deux représentations (script et arbre) sont alors totalement isomorphes. Sinon, il existe plusieurs scripts correspondant au même arbre de preuve. Il existe plusieurs façons d'indiquer à quel sous-but on désire s'attaquer. Nous retiendrons celle utilisée par Coq qui consiste à ordonner les sous-buts restant à prouver par un parcours gauche-droite de l'arbre de preuve et à faire précéder dans le script chaque tactique par l'indice du but auquel elle s'adresse (l'indice est 1 par défaut).

Représentation canonique : Il est par ailleurs possible d'obtenir une représentation canonique de la preuve en considérant un arbre de preuve où les tactiques associées à chaque nœud sont toutes des règles de base de la logique sous-jacente au système. Dans les systèmes basés sur une théorie typée comme Coq, Lego ou Nuprl, on peut aussi représenter la preuve canonique par un λ-terme dont

[1] Une présentation plus complète ainsi qu'un modèle plus détaillé peuvent être trouvés dans [15]

l'arbre canonique est simplement la dérivation de typage. Suivant l'isomorphisme de Curry-Howard un énoncé est alors un type et une preuve un terme de ce type.

Opérateur de tactique : Il est généralement possible de combiner des tactiques pour obtenir de nouvelles tactiques. Pour cela on utilise des combinateurs de tactiques ou tacticielles. On retrouve principalement, bien que cette liste ne soit en rien exhaustive, les opérateurs `Then`, `ThenL`, `Try`, `Orelse`[2].

Exemple 1 (Associativité de l'addition)

Considérons l'énoncé suivant qui exprime l'associativité à gauche de l'addition[3] *:*

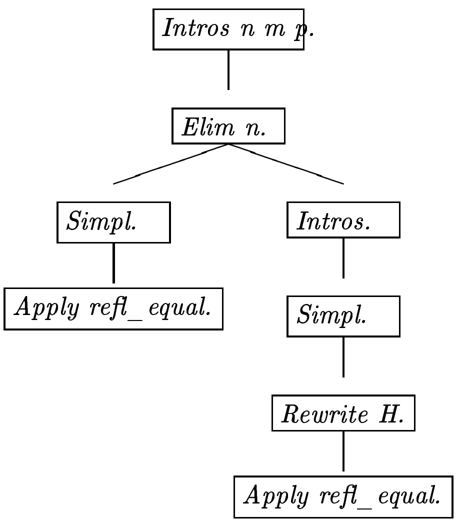
```
Lemma plus_assoc_l : (n,m,p:nat)((plus n (plus m p))=(plus (plus n m) p)).
```

On peut le prouver indifféremment par l'un des deux scripts suivants :

```
Intros n m p.
Elim n.
  (* Cas de base*)
Simpl.
Apply refl_equal.
  (* Étape d'induction*)
Intros.
Simpl.
Rewrite -> H.
Apply refl_equal.
```

```
Intros n m p.
Elim n.
(* On passe d'un cas à l'autre  *)
2:Intros.
1:Simpl.
2:Simpl.
2:Rewrite -> H.
1:Apply refl_equal.
1:Apply refl_equal.
```

Ces deux scripts correspondent à l'arbre de preuve suivant :



[2]`Then T T1 T2 T3...Tn` : applique `T` puis `T1` sur chacun des sous-buts produits par `T` puis `T2` sur les sous-buts produits par les applications de T1 etc. En Coq et dans la suite de ce papier cet opérateur de composition séquentielle est représenté en notation infixe par un "`;`". `ThenL T T1...Tn` : applique `T`. Si `T` ne produit pas `n` sous-buts il y a échec, sinon cet opérateur applique en parallèle chaque `Ti` au sous-but correspondant. L'opérateur `ThenL` de composition parallèle peut s'exprimer en utilisant `Then` et un opérateur que l'on notera `Paralel`. De ce fait on reprendra parfois la notation de Coq soit `T;[T1|...|Tn]`. `Try T` : essaie d'appliquer `T`. Laisse le but inchangé mais n'échoue pas si l'application de `T` provoque un échec. `T1 Orelse T2` : essaie d'appliquer `T1` et en cas d'échec applique `T2`. Si l'application de `T2` échoue aussi, cela produit une erreur.

[3]et qui se lit : $\forall n, m, p : nat.\ \ n + (m + p) = (n + m) + p$

Il est aussi possible de construire un script plus compact en utilisant des tacticielles (l'arbre de preuve correspondant n'a évidemment qu'un nœud) :

```
Intros;Elim n;[Simpl|Intros;Simpl;Rewrite->H];Apply refl_equal.
```

Mais toutes ces preuves correspondent à une représentation canonique donnée ici sous forme d'un λ-terme :

```
proof:
[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)
```

où la notation $[x : nat]Y$ *signifie* $\lambda x : nat.Y$
et où `nat_ind` *est le théorème de récursion sur les entiers d'énoncé :*

```
nat_ind
  : (P:(nat->Prop)) (P O)-> ((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

1.2. Interfaces et systèmes de preuve

Actuellement, la majorité des utilisateurs des systèmes comme Coq, Lego, ou HOL travaillent encore sans interface digne de ce nom : ils écrivent leur script avec un éditeur et font du "copier-coller" vers le système de preuve.

Bertot, Théry et Kahn [19] ont montré que la technologie développée autour du système Centaur pour définir des environnements de programmation pouvait être utilisée pour fournir des interfaces graphiques au système de preuve interactif ouvrant par là de nouveaux horizons comme les notions de preuve par sélection [4], ou de "drag-and-drop" [3]. Après diverses expérimentations autour des systèmes HOL, Isabelle et Lego, cette technologie a été pleinement mise en œuvre pour le système Coq donnant naissance à l'environnement CtCoq utilisé pour notre travail.

Certaines des idées développées dans le système CtCoq ont été reprises récemment par David Aspinall, Healfdene Goguen, Thomas Kleymann et Dilip Sequeira dans un cadre moins structuré que celui de CtCoq. Ils ont utilisé Emacs pour donner naissance à l'environnement générique ProofGeneral [1] qui fournit en standard des interfaces pour Coq, Lego et Isabelle.

D'autres systèmes comme PVS proposent aussi des environnements spécifiques basés sur Emacs[4].

Nous présentons maintenant certains des aspects clefs de ces interfaces qui ont servi de socle au développement de nos outils. Le point fondamental pour notre travail est la gestion du script. Le problème principal est de garder la cohérence entre le contenu de l'éditeur et l'état du système : c'est-à-dire, assurer que si l'on rejoue ce qui est dans l'éditeur, on retrouvera le même état du système. Pour cela la fenêtre d'édition est généralement divisée en plusieurs zones. L'une, protégée en écriture, contient les commandes qui ont été envoyées et acceptées par le système. L'autre éditable, contient les commandes qui n'ont pas encore été envoyées[5]. Lorsqu'une commande de la seconde zone est envoyée au système et qu'elle est acceptée, elle est ajoutée à la premiere zone. Si elle échoue, le script n'est pas modifié et un message d'erreur est émis dans une fenêtre de dialogue. Ainsi on a la garantie que ce qui est contenu dans la première zone correspond exactement à l'état du système.

[4]Pour un panorama rapide des interfaces existantes on peut consulter [15]

[5]CtCoq contient aussi d'autres zones contenant par exemple les commandes envoyées mais pas encore acceptées par le système mais nous n'en parlerons pas ici.

Une autre considération importante est la possibilité de travailler avec plusieurs fenêtres d'édition permettant de développer différentes preuves en parallèle. Enfin l'environnement structuré, basé sur la manipulation des arbres de syntaxe abstraite de CtCoq permet la mise en œuvre d'un mécanisme d'affichage incrémental (basé sur le langage de boîtes PPML) et facilite l'instrumentation du script.

2. Manipulations au niveau d'une preuve

2.1. Notions de dépendance dans une preuve

L'arbre de preuve fait apparaître les relations logiques entre les sous-buts (et donc les tactiques qui leur sont appliquées et les sous-preuves correspondant à ces sous-buts). En effet si, comme dans l'exemple 1, l'application d'une tactique génère deux sous-buts, ils sont a priori indépendants (la validité de cette hypothèse d'indépendance est discutée au paragraphe 2.4) et quelques soient les preuves que l'utilisateur fournisse pour chacun d'eux, elles permettront de construire une preuve du but initial.

Pour modéliser cette relation de dépendance, on associe à chaque nœud de l'arbre de preuve son chemin dans l'arbre. Un chemin est simplement un mot sur N^+. On introduit alors une relation d'ordre partiel $\prec_p$ sur les chemins. Cette relation est un ordre *préfixe* classique défini par :

$$c_1 \prec_p c_2 \Leftrightarrow \exists c \; c_2 = c_1.c$$

Le script retrace quant à lui l'historique de la preuve. Il introduit une relation de dépendance historique entre les tactiques (donc entre les sous-buts auxquels elles s'appliquent). Pour modéliser cette relation, on peut considérer le script comme une fonction des entiers positifs vers l'ensemble des tactiques. La fonction réciproque qui associe à chaque tactique sa position est appelée fonction de rang, on la note rg. On introduit ainsi un ordre total $<_r$ sur l'ensemble des tactiques considérées défini par :

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

Pour que le script soit valide, la fonction de rang doit être bijective et vérifier la propriété suivante : $c \prec_p c' \Rightarrow rg(c) < rg(c')$

Les mécanismes de gestion de script permettent, nous l'avons vu, d'être sûr que le script conservé est cohérent. Ils fournissent en outre un mécanisme rudimentaire de retour-arrière qui conserve cette cohérence. Nous allons maintenant présenter des outils améliorant la compréhension d'une preuve et du script conservé, puis nous montrerons comment optimiser le mécanisme de retour-arrière. Les outils présentés dans la suite de cette section, utilisent tous intensivement ces deux notions de dépendance même si cela n'apparaît pas toujours explicitement dans la présentation intuitive que nous en donnons.

2.2. Compréhension : visualisation et navigation

Visualisation graphique : Il est important de bien comprendre la structure des preuves. Pour cela, on pourra chercher à fournir à l'utilisateur une nouvelle représentation de sa preuve en visualisant l'arbre de preuve sous forme graphique[6].

Des algorithmes efficaces de dessin d'arbres existent et leur mise en œuvre ne présente que peu de difficultés de sorte que l'on peut rapidement écrire un programme spécialisé permettant de visualiser les arbres de preuve. C'est l'approche suivie par l'équipe PVS en utilisant TclTk. L'autre approche consiste à interfacer l'environnement et/ou le système de preuve avec un logiciel spécialisé. C'est cette approche que nous avons retenue, en interfaçant le logiciel de visualisation daVinci[8, 9] avec

[6] On peut décider de montrer pour chaque nœud le but correspondant ou la tactique qui lui est associée. Dans le second cas, si nous ne voulons pas perdre d'information quand la preuve est incomplète, il faut penser à signaler les nœuds correspondants à des buts ouverts (*i.e.* qui n'ont pas encore été traités par application d'une tactique).

le système Coq[7]. L'avantage de la solution distribuée apparaîtra pleinement à la section 3 où l'on réutilise l'interface pour visualiser les objets d'une théorie. Ces dépendances peuvent être modélisées par des graphes acycliques orientés. La visualisation de tels graphes et notamment la minimisation des croisements entre les arêtes, est un problème très difficile et pour obtenir des résultats utilisables, le recours à des système spécialisés comme daVinci, Dotty ou Graphlet est indispensable.

L'expérience montre la faiblesse des tels outils de visualisation de l'arbres de preuve. En effet la représentation graphique s'avère vite inutilisable sur de grosses preuves. De telles preuves croissent très vite verticalement mais aussi horizontalement. Les longues lignées verticales ne représentent que peu d'information sur la structure de la preuve et on peut donc regrouper les nœuds qui les composent, soit par un artifice graphique lors de la visualisation (on dit alors que l'on se focalise sur les points de choix) soit physiquement par modification du script comme cela est présenté au paragraphe 2.5. La croissance horizontale reflète quant à elle les différents raisonnements par cas et par induction et est fondamentale pour la compréhension de la preuve. Son altération par un artifice graphique de regroupement des nœuds est sans intérêt.

En pratique, sur de grosses preuves, la visualisation graphique n'est donc utilisable qu'en liaison avec une utilisation partielle des outils de contraction du paragraphe 2.5 qui permettront de se focaliser sur les points clefs de la démonstration.

Navigation dans le script : Une autre manière de procéder (que nous fournissons en standard dans CtCoq) est d'instrumenter le script pour permettre d'y naviguer en respectant la structure de l'arbre. Le principe de cette navigation qui n'est pratiquement utilisable qu'avec une interface graphique est de sélectionner une tactique (en utilisant par exemple un mécanisme de sélection) puis d'appliquer une commande de navigation qui déplace la sélection dans le sens voulu. Sur le second script de l'exemple 1 page 173, on peut par exemple sélectionner la tactique `1:Simpl` par un simple clic à la souris et demander quel est son père, cela déplacera la sélection sur la tactique `Elim n`.

Pour mettre en œuvre cette navigation, il suffit d'annoter chaque nœud par son chemin dans l'arbre de preuve. Cette information, invisible à l'utilisateur, permet de calculer le chemin du parent résultant de la commande de déplacement. On doit ensuite pouvoir associer à ce chemin la tactique qui lui correspond dans le script. Cela peut être fait en parcourant le script jusqu'à ce que l'on trouve la tactique annotée par ce chemin ou en maintenant une table de hachage associant à chaque chemin la tactique qui lui correspond dans le script. Mais à chaque théorème correspond un arbre de preuve différent, il faut donc une table de hachage par théorème. On stocke cela en utilisant une table globale dont les clefs sont les noms des théorèmes du développement et les valeurs, les tables de chaque théorème. Notons que cet ajout d'information consomme de la mémoire qu'il faut gérer avec soin[8].

2.3. Retour arrière

Le retour arrière est un mécanisme fondamental de tout outil interactif : il permet d'avoir l'assurance que l'on peut faire des erreurs sans trop de coût puisque l'on pourra revenir sur les décisions prises sans repartir de zéro. Nous cherchons néanmoins à minimiser ce coût. Pour cela, nous minimiserons évidement ce qui doit être défait mais comme l'a suggèré Vitter [21], l'utilisateur doit pouvoir revenir sur un retour arrière en restaurant tout ou partie de ce qui a été défait. On parle alors de mécanisme de "Undo-Redo".

Dans le cadre des systèmes de preuve, faire un retour-arrière consiste à supprimer l'action de certaines des tactiques jouées. Ces tactiques doivent donc être supprimées du script conservé. Dans une bonne interface, les tactiques effacées doivent être mises de côté pour pouvoir éventuellement être rejouées. Cela permet de revenir sur le retour-arrière comme sur tout autre ordre envoyé au système de preuve.

[7]Une description du prototype est donnée en annexe de [15]
[8]L'implémentation complète est décrite dans [14]

Retour-arrière historique : Dans tous les systèmes de preuve que nous connaissons, le mécanisme de retour-arrière est historique. C'est-à-dire que le système gère une pile des états correspondant aux commandes qu'il a acceptées. L'utilisateur dispose d'une commande lui permettant de dépiler la dernière tactique jouée et de restaurer l'état du système.

La principale conséquence de cette gestion par pile est que pour supprimer une commande, il faut supprimer toutes celles qui ont été jouées après, même si elles ne s'appliquent qu'à des sous-buts complètement indépendants (*i.e.* incomparables pour $\prec_p$). Un autre défaut est le coût mémoire élevé de la gestion de la pile qui entraîne que celle-ci et donc le nombre de retours-arrière possibles sont bornés.

Retour-arrière logique : Ce que nous appelons retour-arrière logique permet de remédier aux inconvénients de la gestion en pile. L'idée de base est d'utiliser les dépendances logiques issues de l'arbre de preuve. Lorsque l'utilisateur fait un retour-arrière logique sur une tactique, ne sont supprimées du script que celles qui appartiennent au sous-arbre dont elle est la racine.

Exemple 2 (Différence entre retour-arrière historique et retour-arrière logique)

Script initial	*Après le retour-arrière historique*	*Après le retour-arrière logique*
Intros n m p.	Intros n m p.	Intros n m p.
Elim n.	Elim n.	Elim n.
2:Intros.	2:Intros.	2:Intros.
Simpl. effacer ici	--------------------	2:Simpl.
2:Simpl.	*Simpl.*	2:Rewrite -> H.
2:Rewrite -> H.	*2:Simpl.*	$\boxed{2}$:Apply refl_equal.
1:Apply refl_equal.	*2:Rewrite -> H.*	--------------------
$\boxed{1}$:Apply refl_equal.	*1:Apply refl_ equal.*	*1:Simpl.*
--------------------	*1:Apply refl_ equal.*	*1:Apply refl_ equal.*

Lors d'un retour-arrière historique, toutes les tactiques suivant la tactique à effacer ont été supprimées. Elle sont cependant conservées dans une zone éditable de sorte qu'elle peuvent être rejouées éventuellement après avoir été modifiées. Lors d'un retour-arrière logique, seules les tactiques dépendant effectivement du `Simpl` *sélectionné sont défaites, l'autre* `Simpl`*, le* `Rewrite` *et l'application de* `refl_equal` *qui se trouvent sur une autre branche de l'arbre de preuve ne sont pas affectés.*

Cet exemple permet d'illustrer une première difficulté. Avec le retour-arrière historique en supprimant une tactique, nous restaurions un état qui avait été précédemment atteint par la preuve et qui était donc forcément cohérent. Ici le script que l'on produit ne correspond pas obligatoirement à un état de preuve par lequel on est déjà passé. Pour maintenir la cohérence, nous pouvons être amené à modifier les index des tactiques de la partie conservée (dans notre exemple `Apply refl_equal` s'applique désormais au second sous-but ouvert et son index doit donc devenir 2). Les mêmes problèmes se posent pour la partie effacée. Si l'on veut être capable de la rejouer, il faut parfois modifier les index pour maintenir la cohérence comme l'illustre l'exemple 3.

Exemple 3

Script initial	*Après le retour-arrière logique*
Intros n m p.	Intros n m p.
Elim n.	Elim n.
2:Intros.	2:Intros.
Simpl	Simpl.
$\boxed{2}$:**Simpl.** effacer ici	Apply refl_equal.
$\boxed{2}$:Rewrite -> H.	--------------------
1:Apply refl_equal.	$\boxed{1}$:*Simpl.*
$\boxed{1}$:Apply refl_equal.	$\boxed{1}$:*Rewrite -> H.*
--------------------	$\boxed{1}$:*Apply refl_ equal.*

Le mécanisme de retour-arrière logique peut se décomposer en deux phases. D'abord une réorganisation du script, dans laquelle certaines tactiques sont déplacées à la fin et où les index sont modifiés pour préserver la cohérence. Après cette réorganisation un retour arrière historique est fait sur le script obtenu de manière à supprimer les tactiques effacées.

Pour pouvoir effectuer la modification des index simplement en parcourant le script, l'annotation de chemin n'est pas suffisante. Il faut disposer d'une information supplémentaire sur le nombre de sous-buts générés par chaque sous-but. Nous pouvons alors calculer la liste des chemins des sous-buts ouverts correspondant à chaque étape. L'index de la tactique jouée à une étape donnée est simplement la position de son chemin dans la liste des sous-buts ouverts de l'étape précédente. Nous avons montré par ailleurs [15] que l'information concernant le nombre de sous-buts générés suffit à reconstruire la structure de l'arbre à partir du script.

Le mécanisme de retour-arrière que nous venons de présenter fonctionne parfaitement dans un monde où les branches de l'arbre de preuve correspondent réellement à des résultats indépendants c'est-à-dire où la façon de prouver le sous but correspondant à une branche n'influe pas sur la façon de prouver les sous-buts des autres branches. Dans les systèmes qui permettent de manipuler des variables existentielles cela n'est pas toujours vrai. Nous étudions maintenant les problèmes posés par l'utilisation des variables existentielles et leur interaction avec le mécanisme de retour-arrière logique proposé.

2.4. Variables existentielles

2.4.1. Utilisation des variables existentielles

Une utilisation classique est liée à l'introduction de quantificateurs. L ors 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, on essaie de prouver $X + 2 = 5$, et par réduction, on obtient $X = 3$ qui nous permet d'instancier X.

Exemple 4 (Contraintes introduites par le partage de variables existentielles)

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

Après application du théorème `trans_equal` *d'énoncé :*
`(x,y,z:A)x=y->y=z->x=z`*, on a deux sous-buts, soit l'arbre de preuve suivant :*

Intros;EApply Trans_ equal	
... *H : a=b* *H0 : b=c* *H1 : a=d* ======= *a=?*	... *H : a=b* *H0 : b=c* *H1 : a=d* ======= *?=c*

On résout d'abord le second sous-but par la commande

```
2:EExact H0.
```

Cela introduit la contrainte `?=b`. *On essaye alors de prouver l'autre sous-but avec* `H1`

```
Try EExact H1.
```

La tactique `EExact H1` *échoue car l'existentielle* `?` *est déjà contrainte par* `?=b` *et ne peut donc être unifiée avec* `d`. *On termine la preuve en utilisant l'hypothèse* `H`.

```
EExact H.
```

2.4.2. Conséquences dans l'utilisation du retour-arrière logique

Reprenons le script complet obtenu à l'exemple 4, et effectuons un retour-arrière logique sur la tactique correspondant à la seconde branche de l'arbre de preuve. Soit :

```
Intros;EApply Trans-equal.
 2:EExact H0.  tactique que l'on veut supprimer
Try EExact H1.
EExact H.
```

Les conséquences de l'application de cette tactique sont supprimées du côté du système de preuve (cela ne supprime pas les contraintes qu'elle a introduites). Du côté de l'interface la tactique est simplement déplacée dans la zone éditable.

```
Intros;EApply Trans-equal.
Try EExact H1.
EExact H.
----------------
 EExact H0.
```

La partie effacée peut être rejouée terminant ainsi la preuve qui peut être sauvegardée. Pourtant le script ainsi obtenu n'est pas valide. Il ne pourra en effet être rejoué dans une prochaine session, car les contraintes introduites ne seront pas les mêmes[9].

2.4.3. Solutions pour un retour-arrière logique en présence de variables existentielles

Forcer l'instanciation : L'idée est de rajouter dans le script une commande introduisant les contraintes initialement introduites par la tactique sur laquelle est fait le retour-arrière, de sorte que lorsque l'on rejouera le script les contraintes introduites soient les mêmes. Cela permet de dissocier le retour-arriere de la désinstanciation des variables existentielles.

Exemple 5

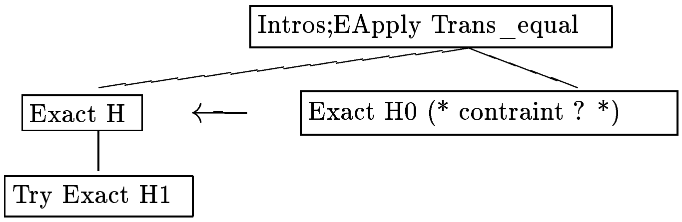
```
Intros;EApply Trans-equal.
Instanciate 1 b (* on a forcé l'instanciation initialement introduite par la tactique effacée  *)
Try EExact H1.
EExact H.
EExact H0.
```

Rajouter un niveau de dépendance historique : L'idée est d'introduire une nouvelle dépendance entre les tactiques qui manipulent des buts partageant des variables existentielles. Toute tactique jouée après la tactique à supprimer et s'appliquant sur un but contenant les mêmes variables existentielles que celles introduites par la tactique à supprimer dépendra d'elle. Les dépendances ne se traduisent plus par un arbre mais par un graphe.

[9] Il n'y aura aucune contrainte au moment de l'application de `Try EExact H1` qui n'échouera plus, résoudra le premier sous-but contraignant la variable existentielle avec la valeur `d`. Dès lors `Exact H` n'a plus lieu d'être puisque le but qu'il a attaqué n'existe plus. De plus appliquer `EExact H0` pour résoudre le second sous-but échouera car cela n'est pas compatible avec la nouvelle contrainte.

Exemple 6

Si l'on considère le script de l'exemple précédent, les tactiques des deux dernières lignes travaillent sur des buts contenant la variable contrainte par la tactique de la ligne 2. On introduit donc une nouvelle relation de dépendance qui peut se traduire par le graphe de dépendance suivant :



Comme précédemment lorsque l'on efface une tactique, toutes celles qui en dépendent par cette nouvelle relation doivent être effacées.

Cette solution ad-hoc présente un compromis entre un retour-arrière historique brut et le retour arrière logique. Il est en outre possible de minimiser la dépendance supplémentaire introduite par les existentielles en fonction de considérations sur la sémantique des tactiques. Certaines optimisations sont discutées dans [15].

Des problèmes liés aux contraintes introduites par les existentielles se rencontrent dans toutes les manipulations de l'arbre de preuve. Nous avons proposé des solutions ad-hoc pour adapter le mécanisme de retour-arrière logique et garantir son fonctionnement dans un cadre avec variables existentielles. De telles solutions n'ont pas encore été étudiées pour les outils d'expansion et de lemmification que nous présentons maintenant que dans un cadre sans variables existentielles.

2.5. Contraction et expansion de preuve

Au paragraphe 1.1 on a introduit la notion de tacticielle. Ces tacticielles peuvent être utilisées pour modifier la structure de l'arbre de preuve au travers de modifications du script (la preuve canonique restant généralement la même). On s'intéresse principalement ici aux opérateurs de composition séquentielle et parallèle qui permettent de définir deux opérations : la contraction et l'expansion. Ces opérations modifient la structure du script et celle de l'arbre de preuve mais laissent inchangée la représentation canonique.

L'opération de contraction permet principalement d'obtenir des scripts plus compacts qui peuvent être plus compréhensibles. Elle peut aussi être utilisée partiellement pour repousser les limites de la visualisation graphique de l'arbre de preuve.

Le spectre des utilisations de l'opération d'expansion est beaucoup plus large. Nous proposons la correction d'erreurs, l'élimination de code mort[10], la sécurisation du script (par exemple, en Coq, le nommage explicite des hypothèses dans le contexte local).

2.6. Lemmification et factorisation de preuve

L'idée de la lemmification est d'isoler à l'intérieur d'une preuve un résultat intermédiaire pour en faire un théorème à part entière qui pourra être utilisé dans d'autres preuves sans répéter la démonstration.

[10]Lorsqu'une tactique composée à l'aide de tacticielles échoue, l'expansion permet de localiser plus précisément l'erreur. De même l'expansion permet de supprimer d'une tactique composée, les tactiques qui ne font rien.

Exemple 7 (Lemmification dans le système Coq)

On a initialement prouvé le théorème `comPlus` *qui établit la commutativité de l'addition sur les entiers naturels.*

Script initial	*Chemins correspondants*	*Sous-buts à isoler*
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]	
(Simpl; Trivial).	[2;1;1;1;1;1]	(S (plus q n))=(plus q (S n))
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]	

Les énoncés de la colonne de droite correspondent au résultat que l'on veut isoler. Pour cela nous avons fourni une fonctionnalité qui, prenant le chemin du but à isoler et le nom du théorème à générer, construit l'énoncé complet et son script de preuve. Ici on peut donc générer les deux lemmes `plus_n_O` `plus_n_Sm` *respectivement associés au chemin* `[1;1]` *et* `[2;1;1;1;1]`*. Une fois les lemmes générés, on peut simplifier le script de départ.*

```
# 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.
```

```
# 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.
```

```
(*simplification du script  initial *)
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.
```

Cet exemple montre l'appel à cette fonctionnalité depuis Coq (ml). Pour l'utiliser telle quelle, il faut fournir un chemin dans l'arbre de preuve, information qui n'est a priori pas immédiate à connaître pour l'utilisateur. On voit ici encore pleinement l'intérêt d'une interface graphique où un simple clic suffira à désigner ce chemin.

Mise en œuvre pratique : La manière la plus intuitive de procéder est de récupérer le contexte local associé au sous-but à isoler et d'abstraire toutes les hypothèses qu'il contient pour obtenir un nouvel énoncé. Pour obtenir la preuve de ce nouvel énoncé, il suffit alors de réintroduire ces hypothèses dans le contexte local de sa preuve (commande `Intro` de Coq) puis de récupérer dans le script initial la partie qui correspond au sous-arbre de preuve issu du sous-but à isoler. Néanmoins cette technique est loin d'être optimale. Le principal inconvénient de cette approche est que le contexte local contient souvent de nombreuses hypothèses qui ne sont pas nécessaires à la preuve du sous-but à isoler. En les abstrayant on obtient un énoncé beaucoup trop contraint. On optimise donc le processus de création de l'énoncé en n'abstrayant que les hypothèses utiles à la preuve.

Cette méthode qui permet de produire des énoncés beaucoup plus généraux soulève deux problèmes. Il faut d'abord être capable de déterminer si une hypothèse intervient ou non dans la preuve. Pour cela, nous faisons une analyse de la représentation canonique de la preuve. L'intersection de l'ensemble des identificateurs libres qu'elle contient avec l'ensemble des identificateurs de la base locale d' hypothèses donne l'ensemble minimal d'hypothèses à abstraire. Le second problème est lié au nommage et au référencement des hypothèses de la base. En effet lorsqu'une hypothèse est introduite dans le contexte local, l'identificateur qui lui est associé peut être fourni explicitement par l'utilisateur (par exemple `Intros H1 H2` en Coq) ou implicitement par le système. Dans le second cas les noms choisis par le système dépendent du contexte local. Or en n'abstrayant pas toutes les hypothèses, nous avons modifié ce contexte. Ainsi s'il y a d'autres introductions implicites dans le script récupéré, les noms d'hypothèses qu'elles introduiront risquent d'être modifiés, et les éventuelles références à ces hypothèses s'en trouveront erronées. Pour éviter ce genre de problème il faut pouvoir nommer explicitement toutes les hypothèses introduites. Cela pourra être fait a posteriori, en utilisant l'outil de sécurisation du script mentionné à la section précédente.

La lemmification introduit de nouveaux théorèmes dans le contexte global. Cela modifie donc les relations de dépendances entre les objets du contexte global. Ce sont ces relations que nous étudions maintenant.

3. Manipulations au niveau d'une théorie

Cette section s'intéresse à la notion de dépendance entre objets (théorèmes et définitions) d'une théorie. Une fois cette notion définie, on peut associer à toute théorie un graphe de dépendance. Ce graphe peut servir de base à la définition de nombreux outils de manipulation des théories que nous décrivons au paragraphe 3.1.1.

3.1. Notion de dépendance entre objets

On dit qu'un théorème A dépend de B si sa preuve utilise B ou si son énoncé fait référence à ce théorème. On pourrait penser que pour calculer de quoi dépend un théorème, il suffit de récupérer la liste des identificateurs qui apparaissent dans son énoncé et dans son script de preuve. L'exemple 8 montre que cela n'est pas suffisant et qu'il faut avoir recours à la représentation canonique donnée ici sous forme de λ-terme.

Exemple 8 (Cacul des dépendances)

Soit le lemme suivant qui exprime l'associativité à gauche de la concaténation des listes[11]*:*

```
Lemma ass_app:(l,m,n:list)(app l ( app m n))=( app ( app l m) n).
```

Le script de preuve et le terme de preuve sont réspectivement :

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

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)
```

Ainsi, en calculant les dépendances de `ass_app` *sur le script on obtient seulement* `list`, `app` *et* `sym_equal` *alors que sur la représentation canonique, on obtient aussi* `app_ass` *qui caractérise l'associativité à droite et qui dans le script est introduit par la procédure de décision automatique* `Auto`.

[11] et qui se lit $\forall l, m, n : list.\ \ l@(m@n) = (l@m)@n$.

En effectuant ce calcul sur tous les objets de l'environnement (lors de leur introduction ou a posteriori) on peut construire un graphe de dépendances. Comme pour visualiser les arbres de preuve, on utilise un logiciel spécialisé dans le dessin de graphe, interfacé avec le système de preuve. La possibilité de visualiser ces dépendances suggère de nouvelles fonctionnalités comme le **Reset-logique** qui est l'analogue au niveau des théories du retour-arrière logique au niveau des preuves.

3.1.1. Utilisation du graphe de dépendance entre les objets

Coupe de théorie : L'idée est, étant donnés un théorème et un développement, de nettoyer la théorie correspondant à ce développement pour ne garder que les résultats qui sont utiles pour prouver le théorème. C'est un peu l'analogue de la coupe de programme (slicing) en analyse de programme.

Le principe est le suivant : Soit T une théorie et t un théorème. Nous calculons la fermeture transitive du sous-graphe issu de t. Nous récupérons la liste des fichiers sources où apparaissent ces identificateurs. Il nous reste alors à parcourir ces fichiers et à y supprimer tous les objets qui ne sont pas associés à un des identificateurs de la fermeture transitive du sous-graphe issu de t. Pour cela, nous avons écrit en Caml-lex un utilitaire qui analyse les fichiers sources et supprime tous ces objets.

Néanmoins certaines précautions doivent être prises. Dans l'exemple 8 l'information sur le script est forcément un sous-ensemble de l'information calculée sur la forme canonique de la preuve. En pratique, si l'on ne prend pas de précaution, cela n'est pas forcément vrai. En effet en Coq par exemple, les termes de preuve sont stockés sous forme normale et certains identificateurs (apparaissant dans la preuve d'un sous-but introduit par une coupure) peuvent disparaître à la normalisation et être néanmoins utiles pour rejouer le script de preuve. Si le script utilise des procédures de décisions automatiques, ils peuvent n'apparaître ni dans le script ni dans le terme normalisé mais être nécessaires pour rejouer le script. Des exemples sont donnés dans [15]. Nous évitons ce genre de problèmes en calculant les dépendances avant la normalisation.

Déplacement de code et réorganisation de théorie : Assurer que des théorèmes d'un même domaine soient réellement stockés dans la même zone est important pour leur réutilisation ultérieure[12] et la maintenance de preuve à long terme.

On rencontre par exemple une situation qui conduit fréquemment à avoir des résultats mal placés quand l'utilisateur découvre qu'il manque un théorème à sa théorie au moment où il l'utilise dans un autre développement (par exemple alors qu'il développe une théorie des polynômes en utilisant la théorie des listes, il s'aperçoit qu'il lui manque une propriété des listes). La plupart du temps, il prouve alors le résultat dont il a besoin dans le contexte où il se trouve et le stocke dans la théorie qu'il est en train de développer (dans notre exemple la théorie des polynômes). Cela conduit à dupliquer du code et du travail car les autres utilisateurs ne sauront pas que ce résultat a été prouvé puisqu'il n'est pas à sa place naturelle. Il est donc important de permettre de réorganiser la théorie. Mais remettre un théorème à sa place "naturelle" n'est pas toujours évident. Il faut d'abord pouvoir affirmer que le résultat à déplacer n'utilise pas de résultats de la théorie courante. Pour vérifier cela on utilisera le graphe de dépendance pour savoir quels sont les théorèmes nécessaires à la preuve du théorème à déplacer et qui devront donc être déplacés avec lui dans son fichier de destination.

Aide à la propagation des modifications : On entend par modification un changement d'axiomatique, par exemple pour satisfaire de nouvelles spécifications, des modifications de définitions ou de fonctions, pour optimiser la représentation, ou dans les systèmes comme Coq ou Nuprl qui proposent un mécanisme d'extraction de programme des modifications de preuve pour optimiser le code extrait...

[12]Même si des outils de recherche dans les bibliothèques (comme `Searchsiso`) peuvent aussi aider à retrouver un résultat dans un ensemble de bibliothèques.

Dans [15] on propose un algorithme pour aider l'utilisateur à minimiser le nombre de modifications nécessaires en proposant à chaque étape un choix de théorèmes à modifier.

Cette méthode incrémentale de choix des théorèmes à modifier autorise en outre des optimisations basées sur la nature des dépendances. En effet ce qui est important en mathématiques c'est la suite de théorèmes que l'on a prouvés, pas la façon dont ils ont été prouvés (principe de "proof-irrelevance"). Pour modéliser cela un système comme Coq autorise deux états sur les objets de l'environnement : opaque ou transparent. Les théorèmes seront généralement opaques de sorte que l'on ne peut accéder à leur preuve alors que les fonctions sont généralement transparentes. On s'est placé dans un modèle simplifié où l'on ne change pas l'opacité[13]. On peut ainsi définir les notions de dépendance opaque (si l'objet auquel on fait référence est opaque) et de dépendance transparente (si l'objet auquel on fait référence est transparent). Les dépendances opaques permettent de arrêter la propagation des modifications lorsque seule la preuve d'un théorème est modifiée et non son énoncé.

Sensibilité au contexte La majorité de nos outils produisent des modifications du contexte global qui peuvent modifier le comportement des procédures de décision automatique . Un affaiblissement du contexte global, comme cela se produit lors d'une coupe de théorie, est a priori sans risque. Les enrichissements de contexte qui peuvent résulter d'un déplacement de code ou de modification sont plus dangereux. Supposons par exemple que par une analyse des dépendances, nous sachions qu'un théorème T dépend d'un ensemble de résultats $\mathcal{R}$. Si l'on modifie un théorème T_1 du contexte global n'appartenant pas à $\mathcal{R}$, cela ne devrait pas invalider la preuve de T. Pourtant si le script de preuve contient des procédures de décision automatique, il est possible qu'il ne repasse pas. En effet, avant la modification, le théorème T_1 ne pouvait être utilisé par la procédure de décision (sinon on l'aurait retrouvé dans $\mathcal{R}$). Mais après les modifications cette procédure peut très bien se mettre à l'utiliser pour résoudre certain sous-buts, invalidant par là le reste du script.

A priori, si une procédure de décision “fait plus de choses”, on peut supposer que le script dans le nouveau contexte doit être un sous-ensemble du précédent. Dans ce cas, les outils d'analyse de la section 2 permettent de supprimer les tactiques devenus obsolètes.

Cela est vrai dans la majorité des cas. Cependant si le script utilise conjointement aux procédures de décisions automatiques, des tacticiellles non monotones comme `Orelse`[14], l'ensemble des sous-buts produits par le script lorsqu'il est joué dans le nouveau contexte ne forme plus forcement un sous-ensemble de ce qu'il était dans l'ancien contexte. Il peut même être totalement différent, rendant toute réutilisation impossible. Une solution, proposée dans [16, 15] pour éviter ce genre de problèmes est de réduire artificiellement l'ensemble des résultats utilisables par les procédures de décisions automatiques pour les limiter aux résultats présents dans l'ancien contexte.

4. Conclusion et perspectives

L'idée qui est à la base des outils d'aide au développement de preuves et de manipulation de théories que nous avons présentés est la notion de dépendance. Ces dépendances forment un arbre dans le cas des preuves et un graphe acyclique direct dans le cas des théories. La conception des outils proposés pouvait paraître simple au premier abord mais des difficultés comme la présence de variables existentielles, les problèmes de nommage et de référencement des hypothèses du contexte local, la sensibilité au contexte de certaines tactiques apparaissent rapidement.

Ce travail de recherche a pour objectif des considérations pratiques, il est donc fondamental d'en étudier l'impact sur un grand nombre d'utilisateurs.

[13] Ça ne semble pas une grosse limitation car la seule utilité que nous avons trouvé au changement d'opacité était de rendre provisoirement opaques des termes pour éviter qu'ils ne soient réduits par `Simpl.` et cela est dû à une faiblesse de cette tactique désormais résolue.

[14] Des exemples de problèmes liés à l'utilisation du `Orelse` se trouvent dans [16, 15]

Les outils travaillant sur les preuves viennent en premier dans l'apprentissage d'un système, même le débutant aura besoin de s'orienter dans ses preuves et sera vite amené à défaire des morceaux de preuves. Ainsi leur utilisation est sans doute plus instinctive.

Au contraire, les outils travaillant au niveau des théories s'adressent plus à des utilisateurs confirmés déjà capables de faire un gros développement ce qui représente déjà une population plus limitée. En outre, ils demandent sans doute un investissement plus grand pour l'utilisateur qui n'aura pas forcément utilisé notre interface pour faire son développement mais voudra y venir pour le nettoyer et le réorganiser. Il est par conséquent plus compliqué d'analyser l'impact de ces outils.

Cette distinction se retrouve dans nos résultats, les outils travaillant sur les preuves ont été les premiers développés et ont pu être utilisés par d'autres utilisateurs, par exemple dans les développements de preuves mathématiques ou de preuves de compilateurs de l'équipe Lemme de l'INRIA Sophia. On a pu en apprécier l'apport, et de nouveaux besoins apparaissent au travers des utilisations détournées des outils existants (par exemple l'utilisation du retour-arrière logique pour réorganiser le script). Pour les outils travaillant sur les théories, qui sont plus jeunes et pas tous implémentés, on n'a actuellement peu de retour, à part pour l'outil de visualisation des dépendances qui est déjà utilisé, notamment pour illustrer les résultats.

Pour permettre une réelle évaluation des outils proposés, il faudra se donner les moyens de les mettre à disposition d'une large population d'utilisateurs. Pour cela, outre la nécessité de développer et d'intégrer ceux de nos prototypes qui ne le sont pas encore à une interface graphique utilisable comme CtCoq, il faudra étudier avec un soin particulier les problèmes d'ergonomie, afin de permettre à un utilisateur, ayant développé ses théories sans interface, de pourvoir venir, à moindre coût, les retravailler, à l'aide de nos outils, dans un environnement de développement qui ne lui est pas familier.

Un autre axe de réflexion est l'étude de l'influence des caractéristiques du langage de script sur le développement des outils proposés. La facilité de développement de tels outils peut servir de critère d'évaluation du langage de commande lui même.

Enfin si notre étude se veut générique, l'implémentation n'a été réalisée que pour le système Coq. Il semble que la majorité des outils proposés puissent s'adapter à d'autres systèmes mais l'influence des caractéristiques des systèmes, telles que la présence ou l'absence de variables existentielles ou d'un terme de preuve, la façon de référencer les hypothèses locales, etc. devra être étudiée plus en détail. L'adaptation des outils de retour-arrière logique dans un langage comme Isabelle basé sur l'unification d'ordre supérieur semble par exemple une perspective interressante[15].

Bibliographie

[1] David ASPINALL, Healfdene GOGUE, Thomas KLEYMANN, and Dilip SEQUEIRA. « *Proof General 2.1* ». LFCS, University of Edinburgh., August 1999.
`http://zermelo.dcs.ed.ac.uk/~proofgen/`.

[2] Janet BERTOT, Yves BERTOT, Yann COSCOY, Healfdene GOGUEN, and Francis MONTAGNAC. « User Guide to the CtCoq Proof Environment ». Rapport technique RT0210, INRIA, 1997.

[3] Yves BERTOT. « Direct Manipulation of Algebraic Formulae in Interactive Proof Systems ». In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, September 1997.

[4] Yves BERTOT, Gilles KAHN, and Laurent THÉRY. « Proof by Pointing ». In *International Symposium on Theoretical Aspects of Computer Science*, 1994.

[15]Une solution basée sur des modifications de l'algorithme d'unification est étudiée dans [7]

[5] Patrick BORRAS, Dominique CLÉMENT, Thierry DESPEYROUX, Janet INCERPI, Gilles KAHN, Bernard LANG, and Valérie PASCUAL. « Centaur: the system ». In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[6] Robert CONSTABLE, S. F. ALLEN, H. M. BROMLEY, W. R. CLEAVELAND, J. F. CREMER, R. W. HARBER, D. J. HOWE, T. B. KNOBLOCK, N. P. MENDLER, P. PANANGADEN, J. T. SASAKI, and S. F. SMITH. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[7] K. A. EASTAUGHFFE and M. A. OZOLS. « A Proof Tree Package for Isabelle ». http://www.cl.cam.ac.uk/users/kae24/.

[8] Michael FRÖHLICH and Mattias WERNER. « The Graph Visualization System daVinci - A User Interface for Applications ». Technical Report 5/94, Department of Computer Science; University of Bremen, September 1994.

[9] Michael FRÖHLICH and Mattias WERNER. « daVinci V2.0 Online Documentation », 1996. accessible at http://www.informatik.uni-bremen.de/˜davinci/doc_V2.0.

[10] Michael J. C. GORDON and Thomas F. MELHAM. *Introduction to HOL : a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[11] INRIA. « *Centaur 1.2* », 1993.

[12] INRIA. « *The Coq Proof Assistant Reference Manual* », December 1996. Version 6.1.

[13] « The LEGO World Wide Web page ». url `http://www.dcs.ed.ac.uk/home/lego`.

[14] Olivier PONS. « Undoing and Managing a proof ». In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997 "*, Sophia-Antipolis, France, 1997. http://www.inria.fr/croap/events/uitp97-papers.html.

[15] Olivier PONS. « *Conception et réalisation d'outils d'aide au développement de grosse théories dans les systèmes de preuve interactifs* ». Thèse de Doctorat, Conservatoire National des Arts et Métiers, 1999.

[16] 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.
`http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[17] Loic POTTIER and Olivier PONS. « dependtohtml :Creating hypertext graphical representation of directed graphs », 1998.
http://www.inria.fr/croap/personnel/Loic.Pottier /Dependtohtml/README.html.

[18] Natarajan SHANKAR, Sam OWRE, and John M. RUSHBY. « A Tutorial on Specification and Verification Using PVS ». Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. (Beta Release).

[19] 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.

[20] Laurent THÉRY. « Proving and Computing: a certified version of the Buchberger's algorithm ». Technical Report 3275, INRIA, Oct 1997.

[21] Jeffrey Scott VITTER. « US&R: a New Framework for Redoing ». *IEEE SOFTWARE*, 1(4):39–52, October 1984.