Université de Nice, Sophia Antipolis

MÉMOIRE

présenté par

Yves Bertot

pour obtenir le diplôme d'

HABILITATION À DIRIGER DES RECHERCHES

Sujet du mémoire:

DES ENVIRONNEMENTS DE PROGRAMMATION AUX ENVIRONNEMENTS DE PREUVE

présenté le 4 octobre 1999 à 10h00 devant le jury composé de:

MM. les professeurs	Jean-Jacques	Lévy	Rapporteurs
	Bengt	Nordström	
	Christine	Paulin-Mohring	
MM. les professeurs	Henk	Barendregt	
	Gilles	Kahn	
	Véronique	Viguié-Donzeau-Gouge	

Table des matières

Chapitre 1

Introduction

Mon travail de recherche se concentre sur l'étude et le développement d'outils pour rendre la programmation plus fiable et plus rapide. Trois sujets précis ont été abordés. Le premier concerne la localisation des calculs dans les programmes. J'ai montré comment la localisation de ces calculs pouvait se faire systématiquement à partir d'une description sémantique du langage de programmation avec des applications possibles dans l'animation de l'exécution et la mise au point pour les "débogueurs" [8, 9, 11, 12, 10].

Le deuxième sujet abordé concerne les interfaces homme-machine pour les outils de démonstration sur ordinateur. J'ai conçu et organisé le développement d'une interface pour le système Coq [35], appelée CtCoq [68, 19, 21, 6, 17].

Le troisième sujet concerne la démonstration de correction d'outils manipulant des programmes [18], avec un exemple qui traite un outil d'optimisation de programmes par propagation de constantes. Plus récemment, j'ai également démontré sur ordinateur la correction d'un compilateur [16].

Nous allons maintenant donner une description plus précise du contexte dans lequel se sont effectuées ces recherches et revenir plus en détail sur chacun de ces sujets.

1.1 Recherche d'erreurs et calculs de position

Dans notre travail sur la recherche d'erreurs, nous nous sommes intéressé au problème de remontée dans le programme source lors de l'exécution de spécifications de sémantique naturelle.

Lorsque l'on dispose de la spécification sémantique complète d'un langage de programmation et d'un moyen d'exécuter cette spécification, on dispose effectivement d'un interprète pour le langage considéré. En effet, il est possible d'exécuter la spécification pour comprendre comment un programme opérera sur des données. Si le programme contient une erreur, l'interprète s'arrêtera, éventuellement en émettant un message d'erreur. Pour faire de cet interprète un véritable outil de programmation il est important que le message d'erreur permette de localiser l'endroit du programme qui devrait être corrigé. Pour obtenir ce résultat, l'interprète doit être capable de localiser en permanence l'expression en cours d'exécution dans le programme. Ce problème était résolu avant nos travaux seulement pour les langages séquentiels sans procédures. Nous avons étendu la solution pour des langages parallèles ou fonctionnels [8, 9].

Ce travail est basé sur la définition d'un langage de termes spécifiquement étudié pour représenter les relations entre un programme de départ et un programme modifié en cours d'exécution [12]. Ce langage permet de décrire les notions de résidus qui apparaissaient déjà dans l'étude des systèmes de réécriture et du lambda-calcul. Nous avons en particulier comparé les notions de résidus qui apparaissent dans le lambda-calcul et dans son implémentation dans des systèmes de réécriture par des mécanismes de substitutions explicites [11]. L'ensemble de ces travaux a été réuni dans ma thèse de doctorat [10].

1.2 Développement d'un environnement de preuve

Pour assurer formellement l'absence d'erreurs dans un programme, il faut disposer de moyens d'exprimer le comportement attendu et de le comparer avec le comportement effectif. Pour éviter les ambiguïtés, une bonne méthode est d'exprimer ces comportements à l'aide de formules logiques et de montrer leur équivalence. Pour effectuer ce travail, il est nécessaire de disposer d'outils de preuve.

A l'heure actuelle, une communauté très large de chercheurs s'intéresse aux problèmes de la preuve sur ordinateur. Les techniques diffèrent. Par exemple, il existe une grande communauté de chercheurs qui prônent la preuve automatique par ordinateur. D'autres défendent plutôt des techniques interactives, où l'utilisateur joue un rôle important dans la recherche de la preuve et l'ordinateur sert plutôt à vérifier les étapes proposées par l'utilisateur.

Nous nous sommes intéressés aux outils de preuve interactifs pour plusieurs raisons. Premièrement, nous pensons que ces outils permettent de vérifier sur ordinateur des résultats beaucoup plus complexes grâce à l'intervention éclairée de l'utilisateur humain, ce qui n'empêche pas d'utiliser une preuve réalisée automatiquement comme prélude à une automatisation des techniques mises à jour. Deuxièmement, les systèmes de preuves sont de bons modèles d'outils dans lesquels on peut manipuler des objets (les preuves) et des spécifications pour ces objets (les énoncés). Un environnement de preuve est donc un précurseur d'un environnement de travail où le programmeur peut manipuler ensemble des programmes et des spécifications pour ces programmes. A ce titre, le travail sur les environnements de preuve a été un formidable banc d'essai pour les outils proposés pour manipuler des programmes par l'équipe CROAP dans le système Centaur [22]. Troisièmement, nous voulions disposer d'un outil pour raisonner sur la sémantique de langages de programmation à l'aide de l'ordinateur.

Avoir utilisé l'environnement Centaur pour développer des environnements de preuve nous a permis d'aborder le problème de l'interaction entre l'homme et le système de preuve avec un point de vue novateur. En particulier, les outils de manipulation structurée ont permis de définir de nouveaux modes d'interaction qui améliorent notablement la productivité de l'utilisateur du système de preuve. Le premier de ces modes d'interaction est la *preuve par sélection* [19], un mécanisme qui interprète les positions sélectionnées par l'utilisateur pour construire des raisonnements en calcul des prédicats. Le deuxième de ces modes d'interaction est l'*interprétation des mouvements* [13], un mécanisme où les mouvements effectués par l'utilisateur avec la souris sont utilisés pour construire des raisonnements manipulant des expressions algébriques. En outre, nous nous sommes interrogés sur les données importantes manipulées au cours d'une session d'utilisation du système de preuve. Ayant compris que les données les plus importantes sont en fait les commandes envoyées au système de preuve, nous avons élaboré un outil de *gestion de script* qui fournit un support efficace à la manipulation de ces données.

Dans notre travail sur les environnements de programmation nous avons voulu dépasser le stade de la maquette et fournir un environnement de travail réellement utilisable pour aborder des développements mathématiques de grande taille. Ce travail s'est concrétisé dans un système appelé CtCoq [6, 5, 7], distribué à quelques dizaines de sites et effectivement utilisé dans quelques grands développements, dont un développement effectué par un partenaire industriel. En retour de cet effort, nous avons pu affiner notre compréhension du développement d'outils robustes et pratiques à entretenir pour l'aide à la preuve sur ordinateur.

1.3 Vérification d'outils de programmation

L'un des moyens de développer ou de maintenir des programmes est d'appliquer des transformations sur des programmes existants, en essayant de limiter les changements apportés à la sémantique des programmes manipulés. En particulier, les outils d'optimisation de programmes doivent impérativement préserver les valeurs obtenues lors de l'exécution des programmes. Nous avons étudié un exemple simple d'optimisateur de programme pour comprendre les étapes nécessaires pour prouver la correction de cet optimisateur. Ce travail a permis de mettre à l'épreuve d'une part des outils de traduction développés par d'autres membres du projet qui permettent d'entrer des spécifications sémantiques dans le système de preuve et d'autre part notre environnement de preuve. L'enseignement principal de cette expérience est que les preuves effectuées en sémantique des langages ont des caractéristiques spécifiques: grands nombre de cas, usage important de définitions inductives. Ces caractéristiques nous ont amené à définir des tactiques spécialement adaptés à ces classes de preuves.

Par la suite, nous avons travaillé à la preuve d'un compilateur pour un langage académique simpliste. Ici encore, l'objectif est de montrer que l'exécution du programme compilé est bien décrite par la spécification de haut niveau. Pour ce travail, nous avons tenu à aborder le problème du raisonnement sur les branchements dans les langages assembleurs, en considérant un langage assembleur sans structure, avec étiquettes et instructions de branchement. Ceci a permis de dégager quelques résultats simples sur les conditions que peuvent vérifier les programmes en assembleur pour faciliter leur manipulation dans des raisonnements logiques. Nous avons également montré qu'une réflexion sur les techniques de preuve par récurrence permettait de résoudre les problèmes créés par la distance entre la structure du langage source et celle du langage cible. Ces travaux sont réunis dans [16].

1.4 Remarques générales sur ce document

Derrière la variété des applications traitées dans ces travaux de recherche, on retrouve quelques fils conducteurs.

Le premier fil conducteur est la prédominance de la notion de position. Cette notion est importante lorsque l'on veut obtenir un outil de programmation pratique à partir de l'interprète dérivé de la sémantique d'un langage. Cette notion ré-apparaît dans le travail sur les environnements de preuve où elle joue un rôle central dans les mécanismes de preuve par sélection et de manipulation directe d'expressions algébriques.

Le deuxième fil conducteur est l'approche systématique par transformation de programmes. Ainsi, les calculs de position dans les spécifications sémantiques sont obtenus par transformation de ces spécifications. De même, le mécanisme de preuve par sélection se définit abstraitement comme une instrumentation des règles d'inférence de déduction naturelle ou du calcul des séquents. Enfin, l'optimiseur étudié dans notre travail sur les preuves en sémantique est lui aussi une transformation de programmes. Pour ces transformations, on cherche systématiquement à comprendre les critères syntaxiques qui les dirigent. Cette propension à retourner à la syntaxe peut sembler aride car elle conduit à pousser des symboles en s'interdisant de comprendre leur signification. En contrepartie, on permet d'obtenir des outils qui sont robustes et pourront s'adapter à des domaines variés.

Enfin, le dernier fil conducteur de ce travail de recherche est la volonté de fournir des outils pratiques, permettant de mettre des résultats théoriques à la portée d'un plus grand nombre d'utilisateurs de l'informatique. Le travail sur le calcul de positions dans les spécifications sémantiques permet de construire des outils d'animation de programmes, qui aident des programmeurs à mieux comprendre le comportement de leurs programmes sans nécessiter des connaissances profondes en sémantique. Le travail sur l'environnement de preuve CtCoq contribue, nous l'espérons, à mettre la théorie du calcul des constructions, à la portée de mathématiciens et d'ingénieurs peu habitués aux finesses de la méta-théorie et de la logique. En retour, cette approche pratique fournit des motivations pour des investigations théoriques bien ciblées.

Dans la suite de ce document, nous décrivons les différents axes de recherche dans l'ordre suivant: le chapître 2 présente le problème de calcul des positions sur les spécifications de sémantique naturelle. Les chapîtres 3, 4, 5, et 6 présentent les recherches sur les environnements de preuve, en se concentrant dans l'ordre sur l'architecture générale de l'interface homme-machine, le mécanisme de preuve par sélection, le mécanisme d'interprétation des mouvements et l'outil de gestion de scripts. Les chapître 7 et 8 présentent des preuves en sémantique des langages de programmation en se concentrant sur un petit évaluateur partiel et sur un compilateur. Enfin, le chapître 9 présente les perspectives futures de ces travaux de recherche.

Chapitre 2

Calculs de position en sémantique

La *Sémantique naturelle* [49] et la sémantique opérationnelle structurelle [57] sont des techniques basées sur des règles d'inférence proches des règles présentes en déduction naturelle [64]. Dans le projet CROAP, un effort important a été effectué pour permettre l'exécution de spécifications sémantiques écrites dans ces styles [33]. Grâce à cette capacité à exécuter les spécifications formelles, il devient possible de fournir des implémentations de référence pour des langages de programmation à partir de spécifications abstraites. Ainsi, la description formelle du typage d'un langage est utilisée directement pour produire un vérificateur de types. La description formelle de la sémantique dynamique fournit un interprète du langage, et la description de la traduction vers un second langage fournit un compilateur.

Pour une utilisation pratique, un vérificateur de type est intéressant s'il donne des informations sur les erreurs rencontrées plutôt que de simplement accepter ou rejeter le programme observé. De même, un interprète sera plus utile s'il permet de contrôler la progression de l'exécution, éventuellement en mettant des points d'arrêts. Ces deux exemples permettent de montrer que l'un des types d'information importants que doit manipuler un outil sur un langage de programmation est l'information de position.

Dans son implémentation de la sémantique naturelle, Thierry Despeyroux a fourni un système de calcul de position, appelé *suivi de sujet*, qui permet de répondre partiellement au besoin en information de position, du moins pour les vérificateurs de types et les interprètes de langages séquentiels. L'effort effectué durant ma thèse de doctorat visait l'extension de ce suivi de sujet à des formes plus générales de descriptions sémantiques, de façon à couvrir également les descriptions sémantiques suivant le style *sémantique opérationnelle structurelle*, nécessaire pour la description de langages concurrents et les descriptions formelles de traduction. Ce travail a mené d'une part à la description formelle d'une structure de donnée et de ses propriétés algébriques et d'autre part à une méthode pour intégrer des calculs sur cette structure de donnée dans les spécifications formelles.

2.1 L'outil initial de suivi de sujet

Le problème posé est d'établir une relation entre les nœuds d'un arbre T qui représente l'état courant et les nœuds d'un arbre T_i qui représente l'état initial. Si une erreur est détectée sur l'état courant, il sera alors possible d'indiquer à l'utilisateur l'endroit qui doit être corrigé sur l'arbre initial.

La solution proposée par T. Despeyroux consiste à ne s'intéresser qu'aux cas où l'arbre courant à chaque étape est un sous-arbre de l'arbre courant à l'étape précédente (donc, de proche en proche, un sous-arbre de l'arbre initial). Ainsi, on considère des règles de sémantique de la forme suivante:

$$\frac{\rho_1 \vdash t_1 \to r_1 \quad \cdots \quad \rho_n \vdash t_n \to r_n}{\rho \vdash op(t_1, \ldots, t_n) \to R}$$

La forme des différentes prémisses peut varier, mais ce qui compte est que le sujet de ces prémisses, $t_1 \ldots t_n$ est toujours une sous expression du sujet de la règle $op(t_1, \ldots, t_n)$. Si le sujet de la règle est un sous-arbre de l'arbre initial, repérable par un chemin dans ce dernier, alors le sujet de la règle utilisée pour prouver la prémisse $\rho \vdash t_k$ sera la concaténation de ce chemin et du chemin élémentaire allant vers le $k$ième fils. Le système de suivi de sujet consiste donc en l'ajout d'un paramètre à chaque jugement qui représente le chemin vers le sujet de ce jugement dans l'arbre initial. Lorsque les règles ont la forme donnée ici, on peut analyser chaque règle au moment de la compilation pour indiquer comment les chemins vers les sujets des prémisses peuvent être obtenus à partir du chemin vers le sujet de la conclusion.

L'implémentation fournie par T. Despeyroux propose même une relaxation de la contrainte de sous-terme strict: si le sujet de l'une des prémisses est de la forme $g(t', t_k)$, où t' n'est pas un sous-terme du sujet de la conclusion mais t_k l'est, il est possible de représenter le chemin vers ce sujet comme un couple (nil, p_k), où nil est utilisé pour indiquer qu'il n'y a pas d'information pour le premier sous terme et p_k est l'expression indiquant le chemin vers t_k. Ainsi les origines des nœuds des termes reconstruits sont partiellement connues.

Remarquons également que le calcul de suivi de sujet proposé par T. Despeyroux est basé sur l'hypothèse que les données sont transférées directement de la conclusion de la règle vers les prémisses. Les extensions que nous allons considérer utilisent d'autres hypothèses.

Le suivi de sujet présenté ci-dessus s'applique bien pour la majeure partie des spécifications de sémantique statique, qui se font généralement par une simple traversée de l'arbre considéré. Elle fonctionne encore pour la spécification de sémantique dynamique d'un langage qui ne contiendrait que des boucles `while` ou `for` pour décrire les calculs répétitifs. Elle ne fonctionne déjà plus si l'on considère un langage avec procédures ou avec `goto`, car alors des fragments de codes conservés pour le futur sont rangés dans une structure intermédiaire, un *environnement* qui n'est pas un sous-arbre du programme initial. Un palliatif est d'utiliser une directive qui permet de donner explicitement l'information de position à certains endroits. La spécification sémantique doit alors aussi décrire les calculs qui permettent de récupérer cette information de position pour la donner explicitement.

Cette approche où l'on ajoute explicitement des raisonnements sur les informations de position dans la spécification formelle n'est pas satisfaisante, car on n'est alors pas assuré d'avoir modifié la sémantique du langage de façon saine. Néanmoins, la solution proposée par T. Despeyroux contient sous forme embryonnaire la majeure partie des éléments qui vont être utilisés dans la solution que nous proposons.

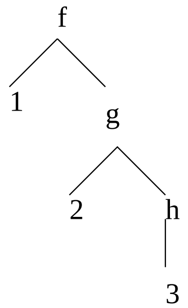
2.2 Le langage des origines

Dans cette section, nous allons progressivement introduire les différents opérateurs qui permettent de construire les termes que nous utiliserons pour dénoter des positions et des origines, c'est-à-dire des fonctions partielles de positions vers des positions.

Le moyen le plus naturel pour représenter une position dans un arbre est la succession des indices qui décrit le chemin suivi depuis la racine de l'arbre, comme dans une nomenclature à la Dewey. Nous utiliserons une variante de cette structure de donnée où les premiers indices dans la séquence décrivent les pas les plus profonds dans l'arbre. Ainsi, le premier élément de la séquence indique la position du nœud considéré par rapport à son père, le second élément indique la position du père par rapport au grand-père, et ainsi de suite. Nous verrons que le choix de cette variante s'adapte bien à l'instrumentation de règles d'inférence.

Dans mon travail de thèse, j'ai préféré considérer qu'une position était en fait une fonction partielle sur les arbres, qui à tout arbre associe le sous-arbre à cette position. Le constructeur de séquence est alors la composition des fonctions, les séquences élémentaires sont les fonctions s_k qui à tout arbre $f(t_1,\ldots,t_n)$ associe le sous-arbre t_k, et la séquence vide est simplement la fonction identité, que nous noterons id. A priori, il ne s'agit ici que d'un changement de notation. Nous verrons plus loin que ce changement de point de vue ouvre la porte à des extensions assez riches.

Pour fixer les idées considérons le terme $f(1,g(2,h(3)))$, qui peut être représenté par l'arbre suivant:



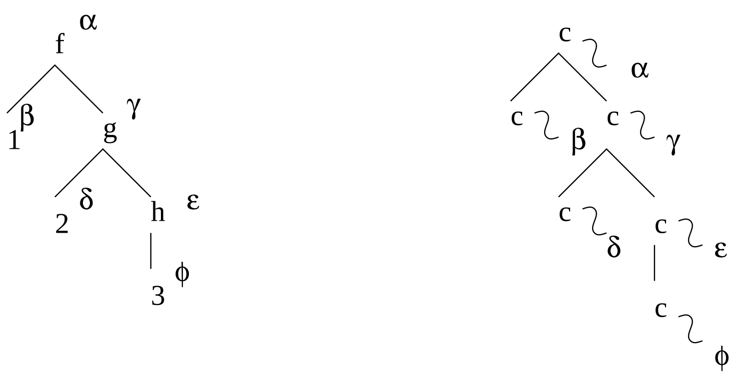
La position de l'atome 2 dans cet arbre est dénotée par la séquence $s_1 \circ s_2$ tandis que celle de l'opérateur h est dénotée par la séquence $s_2 \circ s_2$.

Nous disposons donc d'une structure de donnée pour dénoter des positions dans les arbres. Il faut maintenant se doter d'une structure de donnée pour dénoter les fonctions des positions dans un arbre vers les positions dans un autre arbre. La structure que nous proposons utilise deux techniques, l'une basée sur la structure récursive des arbres et l'autre basée sur la possibilité d'interpréter des positions comme des fonctions sur les positions.

Pour utiliser la structure récursive des arbres, nous constatons que pour décrire la valeur d'une fonction sur toutes les positions d'un arbre correspondant à un terme $f(t_1,\ldots,t_n)$, on peut donner séparément la description des fonctions sur les sous-termes $t_1,\ldots,t_n$ et la valeur à la racine. Ainsi, on peut se donner un opérateur c, qui prend deux arguments, dont le premier est la liste des descriptions de fonctions pour les sous-termes et le second est la description de la valeur à la racine, et qui retourne la description de la fonction pour toutes les positions du terme complet. Notons que ce constructeur c pourrait être utilisé pour n'importe quel type de fonction sur les positions, même si les images ne sont pas des positions.

D'un point de vue intuitif, le constructeur c permet d'associer à tout arbre un arbre isomorphe qui porte une annotation sur chaque nœud. Par exemple, supposons que nous voulions représenter la fonction qui associe des annotations α, β, γ, δ, ϵ et ϕ aux différentes positions dans l'arbre $f(1,g(2,h(3)))$, suivant la représentation donnée à gauche dans la figure suivante. Cette représentation est donnée par le terme représenté à droite, que nous écrirons:

$$c([c([],\beta),c([c([],\delta),c([c([],\phi)],\epsilon)],\gamma)],\alpha)$$



La deuxième technique proposée utilise la possibilité d'interpréter une position comme une fonction sur les positions. Le modèle utilisé est celui de l'extraction d'un sous-terme. En effet, si le terme t provient du terme T simplement en extrayant le sous-terme à la position dénotée par u, alors la position d'origine associée à une position dans t dénotée par v est simplement la position dénotée par la concaténation de u et v. Comme on le voit, cette technique n'est adaptée que pour les termes qui restent intacts après leur extraction du terme initial.

En revanche, l'interprétation des fonctions s_k comme des fonctions sur les arbres se transpose à une interprétation comme fonction d'origine. D'abord, si p_1 et p_2 sont deux positions, on arrive naturellement à l'égalité suivante lorsque l'on interprète l'une d'elles comme une fonction d'origine:

$$p_1(p_2) = p_2 \circ p_1.$$

Si f est une fonction d'origine utilisée pour donner les origines des positions dans un terme $op(t_1, \ldots ,t_n)$, alors $s_k \circ f$ dénote la fonction d'origine qui donne les origines des positions dans le sous-terme t_k. En adaptant cette remarque au cas ou l'on compose une fonction d'origine s_k avec une fonction d'origine $c([f_1, \ldots ,f_n],p)$, on arrive à l'identité importante suivante:

$$s_k \circ c([f_1, \ldots ,f_n],p) = f_k.$$

Pour la composition avec une fonction d'origine simple, on a déjà l'égalité suivante:

$$c([f_1, \ldots ,f_n],p) \circ f = c([f_1 \circ f, \ldots ,f_n \circ f],f(p)).$$

On peut également considérer des fonctions partielles en ajoutant une fonction d'origine constante nil qui représente l'origine inconnue. Cette fonction d'origine interagit de façon simple avec les autres constructeurs: elle est naturellement absorbante pour la composition. [1]

Une contribution importante de mon travail de thèse est donc de montrer que les fonctions d'origines (des fonctions des positions vers les positions) peuvent se représenter à l'aide des opérateurs $\circ$, $c([\cdots],\cdot)$, s_k, id et nil [12].

Ensuite l'étude des propriétés algébriques de ces opérateurs, $\circ$ associatif, nil absorbant, id élément neutre, et les trois égalités données ci-dessus, permettent d'exhiber un système de réécriture confluent et nœthérien sur les termes construits à l'aide de ces opérateurs.

j'ai caractérisé les formes normales closes pour ce système de réécriture. Ce sont id, nil, les positions pures obtenues uniquement par composition de fonctions s_k ou les termes $c([f_1, \ldots ,f_n],p)$ où f_1, $\ldots$, f_n sont eux-mêmes des termes en forme normale et p est une position pure.

Enfin, j'ai montré que si p est une position pure et f est un terme en forme normale, la réduction vers la forme normale de $p \circ f$ prend un temps proportionnel à la taille de p. Cette propriété sera importante lorsque nous utiliserons des règles d'inférences instrumentées pour manipuler des termes de cette forme.

2.3 Instrumentation des règles d'inférence

D'un point de vue théorique, une règle d'inférence donne une information purement déclarative, c'est-à-dire non opérationnelle. Dans la règle suivante:

$$\frac{R(x,z) \quad R(y,t)}{R(f(x,y),g(z,t))}$$

on énonce seulement que si une propriété binaire R est vérifiée pour x et z d'une part et pour z et t d'autre part, alors cette propriété est également vérifiée pour les termes $f(x,y)$ et $g(z,t)$. Cette règle n'exprime pas que x et y ont leur origine dans $f(x,y)$ et

1. Lorsque l'on joue au téléphone arabe, si le message est perdu par l'un des enfants, alors il est perdu pour toute la chaîne.

que z et t ont leur origine dans $g(z,t)$. Après tout, on pourrait considérer que ce ce sont les prémisses qui fournissent les origines.

Cette remarque montre que l'on a besoin d'informations de flux de données pour effectuer un calcul d'origine autour de règles d'inférences. Pour cela, il est également nécessaire de se donner une interprétation opérationnelle des règles d'inférences. Dans notre travail, nous avons utilisé l'interprétation suggérée par l'implémentation de Typol basée sur Prolog. Dans ce cas, le calcul effectué par la machine consiste généralement à résoudre des questions de la forme $R(A,x)$, où A est un arbre et x est une variable dont le résultat sera construit au cours du calcul. L'interprétation intuitive de ce calcul est alors la suivante:

Existe-t-il un x tel que la relation $R(A,x)$ soit vérifiée? Donne alors une valeur possible de x.

Le calcul d'origine est basé sur le sentiment que la valeur trouvée pour x est le résultat d'un calcul sur A et que ses constituants proviennent de fragments de A.

L'interprétation opératoire des règles d'inférence fonctionne alors de la façon suivante: toute règle d'inférence dont la conclusion est compatible avec la question $R(A,x)$, ce que l'on vérifie par unification, peut être utilisée pour effectuer le calcul. L'unification permet d'instancier les variables apparaissant dans les prémisses de la règle d'inférence. On peut alors considérer ces prémisses instanciées comme de nouvelles questions que l'on résoudra récursivement. Lorsque l'ensemble des prémisses sont prouvées, les réponses calculées pour ces prémisses peuvent être utilisées pour construire la réponse pour la variable x dans la question initiale.

Si l'on regarde à nouveau la règle d'inférence suivante:

$$\frac{R(x,z) \quad R(y,t)}{R(f(x,y),g(z,t))}$$

notre interprétation opératoire amène à exprimer que les variables x et y sont obtenues en prenant les sous-termes de $f(x,y)$ et que z et t sont obtenus par les calculs qui résolvent les questions $R(x,z)$ et $R(y,t)$. Enfin, le terme $g(z,t)$ est obtenu en construisant un nouvel arbre au dessus de z et t.

Ainsi, poser la question $R(A,x)$ revient à utiliser cette propriété selon un *mode* donné: le premier argument est une donnée en entrée, nous dirons qu'elle est montante. Le second argument est une donnée en sortie, nous dirons qu'elle est descendante. Le mode associé aux différentes relations ou jugements apparaissant dans une règle d'inférence permet de faire un calcul de flux de données sur les règles d'inférence: les données montantes dans la conclusion et les données descendantes dans les prémisses sont des données qui entrent dans la règle d'inférence. En revanche, les données descendantes dans la conclusion et montantes dans les prémisses sont des données sortantes. Pour effectuer un calcul d'origine pertinent, il est nécessaire de montrer comment les données sortantes sont construites à partir des données entrantes. C'est ainsi que fonctionne l'instrumentation des règles d'inférence.

Les étapes successives de l'instrumentation sont les suivantes:

1. On commence par ajouter un paramètre d'origine pour chaque position montante

ou descendante des jugements et relations apparaissant dans la règle d'inférence. Par exemple, la relation $R(\cdot,\cdot)$ devient une relation quaternaire $R(\cdot,\cdot,\cdot,\cdot)$

2. Pour chaque position en entrée dans la règle d'inférence, on met une variable distincte dans le paramètre d'origine correspondant. Par exemple la règle d'inférence suivante:
$$\frac{R(x,z) \quad R(y,t)}{R(f(x,y),g(z,t))}$$
est transformée en la règle d'inférence suivante:
$$\frac{R(x,z,\cdot,p_2) \quad R(y,t,\cdot,p_3)}{R(f(x,y),g(z,t),p,\cdot)}$$

3. Pour chaque terme apparaissant dans une position de sortie qui est également présent comme sous-terme d'une position d'entrée, on met à jour le paramètre d'origine pour traduire ce transfert de données. Dans notre exemple, la règle d'inférence prend la valeur suivante:
$$\frac{R(x,z,s_1 \circ p,p_2) \quad R(y,t,s_2 \circ p,p_3)}{R(f(x,y),g(z,t),p,\cdot)}$$

4. Pour chaque terme construit à partir de termes connus dans les paramètres en entrée, on utilise un constructeur $c([\cdots],\cdot)$ pour marquer cette origine. Dans notre exemple la règle d'inférence prend alors la forme suivante:
$$\frac{R(x,z,s_1 \circ p,p_2) \quad R(y,t,s_2 \circ p,p_3)}{R(f(x,y),g(z,t),p,c([p_2,p_3],\cdot))}$$

Il reste une information à donner, qui correspond à l'origine donnée à l'opérateur g qui apparait dans le résultat. Ici, il semble naturel de considérer que g descend directement de l'opérateur f et mettre p. Dans le cas général plusieurs opérateurs peuvent apparaitre dans les positions d'entrée et l'on dispose alors de plusieurs choix. L'instrumentation peut alors reposer sur une stratégie fixée à l'avance pour déterminer l'origine des opérateurs créés, comme g dans notre exemple.

Un problème important abordé et résolu dans mon travail de thèse est d'assurer la validité de ce mécanisme d'instrumentation. En effet, on remplace les règles originales par de nouvelles règles manipulant des jugements nouveaux. Peut-on assurer que la spécification originale n'est pas trahie par la transformation? Cette question s'exprime des deux manières suivantes:

- Pour toute question $R(A,x)$ admettant une solution B pour x existe-t-il une solution pour la question $R(A,x,id,y)$ faisant intervenir la relation quaternaire obtenue par instrumentation des règles?
- Pour toute propriété $R(A,B,f_1,f_2)$ prouvable avec les règles instrumentées, la propriété $R(A,B)$ est elle prouvable avec les règles initiales?

Le travail effectué dans la thèse montre que l'on peut répondre positivement à ces deux questions.

La question suivante est celle du coût supplémentaire des calculs ajoutés dans cette instrumentation. En fait, il faut distinguer le coût en mémoire et le coût en temps.

Pour le coût en mémoire, on voit que chaque règle ajoute un peu de matière aux paramètres d'origines manipulés, mais que cette matière est constante pour chaque règle: elle ne dépend pas des entrées et sorties de la règle. Cette indépendance, très utile en pratique, provient du fait que nous utilisons des occurrences pures pour représenter des origines: si $f(t)$ est l'arbre initial, on représentera la fonction d'origine pour le sous-arbre t par le terme s_1 quelque soit sa taille. Cette propriété d'indépendance est probablement le progrès le plus spectaculaire permis par notre langage d'origine. Lorsque l'on considère le calcul en entier, le surcoût est donc simplement proportionnel au coût initial du calcul.

Pour le coût en temps les calculs ajoutés n'apportent dans chaque règle qu'un surcoût constant, proportionnel au surcoût en mémoire. Ces calculs n'effectuent jamais de comparaison entre deux origines: l'unification est seulement utilisée pour unifier une variable à un terme. De ce point de vue, l'unification n'est utilisée que pour effectuer du transfert de données.

Ainsi le surcoût apporté par l'instrumentation des règles est proportionnel au coût initial du calcul (le coût du calcul avec les règles sans instrumentation). Néanmoins, pour étudier complètement le coût, il faudrait également regarder le coût du calcul d'origine pour une position donnée. Ici, les choses se gâtent: pour calculer l'origine d'une position, il est nécessaire de normaliser le terme représentant la fonction d'origine. Une analyse fine des règles de réécriture montre que ce processus de normalisation peut prendre un temps quadratique dans la taille de ce terme, lui-même de taille proportionnelle à l'arbre de preuve construit. Comme le calcul avance en construisant un arbre de preuve, on arrive au résultat déplorable que le calcul d'une origine prend un temps de plus grand au fur-et-à-mesure que l'exécution progresse. Dans le cas où le calcul d'origine est utilisé dans un "debugger", où l'on a besoin de calculer au moins une origine à chaque étape de calcul, l'outil devient impraticable.

Ce problème peut être résolu en profitant de la forme des termes d'origines effectuées dans les règles et du faible coût de la normalisation dans certains cas. En effet, les termes d'origine apparaissant dans les règles d'inférences sont soit des termes de la forme $p \circ u$, où p est un terme clos représentant une occurrence pure de taille bornée et u est une variable d'origine associée à un terme en entrée, soit des termes de la forme $c([p_1 \circ u_1, \ldots, p_n \circ u_n], u_0(p_0))$ où les p_i sont des occurrences pures et les u_i sont des variables d'origines. Nous avons montré dans une section précédente que la normalisation d'un terme de la forme $p \circ u$ prend un temps proportionnel à la taille de p. De plus, si les termes f_i sont les formes normales des termes $p_i \circ u_i$, alors $c([f_1, \ldots, f_n], f_0(id))$ est la forme normale de $c([p_1 \circ u1, \ldots, p_i \circ u_i], u_0(p_0))$. Ainsi, si l'on suppose que toutes les variables décrivant les origines des termes en entrée sont instanciées avec des termes normaux, alors on peut assurer en un temps constant pour chaque règle que les termes décrivant les origines des termes en sortie sont également normaux.

Si l'on se donne un prédicat $N(a,b)$ qui met dans b la forme normale de a et un prédicat $V(a,b)$ qui met dans b la valeur de $a(id)$ [2], cette solution nous amène à construire

2. Lorsque a est en forme normale, a est soit une occurrence pure et alors $a(id) = a$ soit de la forme

la règle instrumentée suivante pour notre exemple favori:

$$\frac{N(s_1 \circ p,p'_1) \quad R(x,z,p'_1,p_2) \quad N(s_2 \circ p,p'_2) \quad R(y,t,p'_2,p_3) \quad V(p,p')}{R(f(x,y),g(z,t),p,c([p_2,p_3],p'))}$$

Ainsi, nous pouvons nous ramener à une situation où le calcul de l'origine d'une occurrence ne prend plus qu'un temps proportionnel à cette occurrence, modulo un surcoût qui est encore proportionnel au calcul sans origine.

2.3.1 Origines et λ-calcul

La notion d'origines est très similaire à la notion de résidu, déjà utilisée dans l'étude du λ-calcul, par exemple pour montrer la confluence forte de ce calcul. Il était naturel de compléter notre étude pour voir comment la technique des calculs d'origine s'appliquait à ce domaine.

En particulier, le λ-calcul peut être représenté par un système de réécriture, introduit sous le nom de substitutions explicites [1]. Existe-t-il un moyen d'instrumenter ce système de réécriture avec des calculs d'origines de façon que les calculs de descendance obtenus correspondent exactement à la notion de résidu existant dans le λ-calcul.

En fait, il existe au moins deux systèmes pour les substitutions explicites. Le premier introduit par Abadi, Cardelli, Curien et Lévy dans [1] est confluent sur les termes clos seulement, tandis qu'un deuxième proposé par Hardin et Lévy [43] est confluent sur tous les termes. Selon les résultats de mon étude publiée dans [11], la distinction entre les deux systèmes se prolonge sur le calcul des origines: il n'est pas possible d'instrumenter le premier calcul, celui de [1] de façon à reproduire le calcul des résidus du λ-calcul, tandis qu'il en existe un pour celui de [43], que j'ai décrit dans la thèse et dans [11].

2.4 Conclusions

L'idée de s'intéresser à des notions de descendance dans les systèmes de réécriture et dans le λ-calcul a été introduite très tôt pour montrer la propriété de confluence du λ-calcul et étudier les réductions optimales. Dans tous ces travaux, on utilise des étiquettes, portées par les nœuds des arbres, pour repérer les descendants d'un nœud particulier. Dans l'implémentation de Typol, le suivi de sujet ne se pose pas réellement le problème du calcul d'origine: les spécifications envisagées procèdent intuitivement en traversant l'arbre considéré et l'argument ajouté aux règles par instrumentation n'a pour but que d'indiquer où on en est dans cette traversée.

La première originalité de mon travail de thèse tient donc en l'idée que l'on veut calculer l'origine de tous les nœuds et effectuer ce calcul en portant les étiquettes non pas sur l'arbre lui-même, mais sur un arbre isomorphe dans un langage spécifique, manipulé sur le coté. La deuxième idée clef est qu'une occurrence représente déjà une information d'origine condensée, qui se manipule très bien en même temps que l'arbre isomorphe portant les étiquettes.

$c([\cdots],p)$ et alors $a(id) = p$.

La troisième idée forte est plutôt inspirée par l'implémentation de TYPOL: le calcul effectif sur les origines s'implémente par une manipulation syntaxique des règles d'inférence, sans qu'il soit nécessaire de connaître en profondeur la sémantique qu'elles décrivent. De plus, ce sont seulement les règles d'inférence que l'on manipule: le moteur Prolog reste intact.

Avant même d'être complètement formalisée, cette méthode a été utilisée pour une implémentation d'un interprète ESTEREL [8] directement dérivé d'une spécification sémantique formelle de ce langage [4]. L'information d'origine était alors utilisée pour montrer en permanence l'ensemble des instructions en cours d'exécution, bien qu'il y ait en général plusieurs instructions actives au même moment. Cette application montrait l'intérêt du calcul d'origine pour le développement d'un outil d'animation de l'exécution, voire d'un debugger [9].

Cette expérience a inspiré les développeurs d'un autre générateur d'environnement basé sur des spécifications exécutables, le système ASF+SDF [51], basé sur des spécifications algébriques. Ils ont développé leur propre solution, basée sur un véritable marquage à l'aide d'étiquettes. Cette solution imposait donc de modifier le moteur d'exécution. Une autre différence notable est que ma solution impose d'indiquer une seule origine pour chaque position, alors qu'ils ont choisi de donner l'ensemble des nœuds consommés comme origine des nœuds créés à chaque étape. Leur approche a l'avantages d'éviter l'intervention humaine que ma méthode impose. Notons que la structure avec l'opérateur $c([\cdots],\cdot)$ n'interdit pas d'associer un ensemble d'origine à un nœud. La méthode que je propose pourrait donc être utilisée pour implémenter le calcul d'origine préconisé dans [69]. Néanmoins, mon travail sur la représentation du λ-calcul par des systèmes de réécriture de substitutions explicites montre que leur approche ne permettra pas de reproduire avec précision la notion d'origine naturelle du λ-calcul [11].

Une application particulièrement spectaculaire de ces outils pour le calcul d'origine a été développée par R. Fraer dans son travail sur la logique de Hoare [39]. La logique de Hoare [38, 46] permet d'annoter un programme impératif avec des assertions exprimant des propriétés des valeurs des variables au point de l'annotation. Une analyse du programme annoté, le calcul de pré-condition la plus faible [34], permet de déduire une liste de conditions dont la vérification permet d'assurer que si la première annotation est vérifié alors la dernière l'est. Ce cadre de travail permet donc de développer des programmes impératifs certifiés. Un problème avec cet outil est que les erreurs de programmation se traduisent par des formules logiques impossibles à prouver. Lorsqu'une telle formule logique est détectée, il est difficile de trouver l'origine de l'erreur dans le programme annoté. R. Fraer a montré que le calcul d'origine sur une description formelle du calcul de pré-condition la plus faible permet de désigner l'ensemble des fragments du programmes qui sont reliés à une condition de vérification donnée. Un point important de son travail, qui dérive directement de ma méthode, est que son implémentation repose sur une manipulation syntaxique des règles d'inférence décrivant le calcul de précondition. Ainsi, cette méthode pourrait s'appliquer rapidement à des implémentations de variantes, comme par exemples celles apparaissant dans les outils pour la méthode B [2].

Quelques développements sont restés inexplorés. D'une part, l'instrumentation des règles d'inférences repose sur une répartition de modes, qui indique les paramètres en entrée et en sortie. A priori, il est possible d'inférer cette répartition de mode grâce à une analyse globale de la spécification sémantique. D'autre part, il est intéressant de voir si le calcul d'origine s'adaptera à la programmation fonctionnelle.

Chapitre 3

Environnements de preuve

3.1 Motivation

Nous avons trois motivations principales pour nous intéresser aux environnements de preuve.

En premier lieu, il est intéressant de disposer d'un outil de preuve pour vérifier les propriétés logiques des spécifications sémantiques que l'on développe dans l'étude des langages de programmation. En particulier, le projet CROAP a décrit assez tôt plusieurs aspects d'un langage académique (sémantique dynamique, vérification des types, compilation) [26] et a fourni la preuve de correction du compilateur [31]. L'étape suivante est de vérifier de telles preuves sur ordinateur.

La deuxième motivation est que le système Centaur fournit une panoplie puissante d'outils interactifs pour la manipulation de données structurées. Les formules logiques et mathématiques sont elles-mêmes des données structurées pour lesquelles ces outils sont aussi adaptés. Ce point de vue a déjà été reconnu par N. Kajler, qui a utilisé Centaur pour développer des interfaces pour systèmes de calcul formel [50]. S'intéresser aux systèmes de preuve permet d'étudier une autre manière de faire des mathématiques sur ordinateur. Ceci a également permis de compléter le système de preuve, en le rendant plus robuste à des utilisations moins traditionnelles [68].

La troisième motivation est que le domaine de la programmation certifiée fait également partie des techniques de programmation pour lesquelles le projet CROAP peut fournir des outils d'assistance. Or le développement de théorèmes permet justement d'étudier un modèle assez fidèle de la programmation certifiée. Cette remarque est encore plus justifiée lorsque l'on considère le système Coq [47], puisque ce système contient justement un langage de programmation fonctionnel pur et, grâce aux outils d'extraction de programmes Caml qu'il fournit, permet le développement de véritables programmes certifiés. En approfondissant le thème de recherche des environnements de preuve, c'est donc vers les environnements pour la programmation certifiée que l'on se dirige.

L'étude de la preuve sur ordinateur est un sujet abordé dans le projet CROAP depuis assez longtemps. Laurent Hascoët a étudié la manipulation symbolique de règles d'inférence et d'arbres de preuve [45], en prenant le point de vue de l'évaluation partielle sur les règles d'inférence [44]. Cet outil, sur Prolog, a permis d'approfondir quelques notions sur représentation des arbres de preuve. J. Despeyroux a également entrepris le développement d'un système de preuve autour des règles de sémantique naturelle, THEO [32], également en Prolog. Enfin, Laurent Théry a été le véritable pionnier de la technique de développement d'interfaces homme-machine pour des systèmes de preuves externes comme HOL [40, 67, 66]. Dans notre travail, nous avons repris ses principaux choix sur l'organisation d'un environnement de preuve:

- utiliser un système de preuve déjà existant dans une architecture multi-processus [27, 48, 28],
- organiser l'interaction en plusieurs fenêtres: plusieurs pour les résultats de différents types, une pour les requêtes
- disposer de protocoles textuels avec des messages à plusieurs champs entre les différents processus [36],
- fournir des outils spécifiques pour certains types de données (comme le graphe de dépendances entre théories) ou certaines phases de l'interaction (comme la réécriture).

3.2 Architecture multi-processus

Nous cherchons à construire un environnement de programmation alliant un système de preuve et des outils de manipulation interactive. Schématiquement deux solutions à ce genre de problème ont été étudiées dans le projet CROAP. Dans la première solution, on lie les deux systèmes en un seul processus. C'est de cette manière que Centaur incorporait le système Prolog, jusqu'à la version 1.2. Dans la deuxième solution, les deux systèmes sont deux processus qui communiquent par des flux de caractères, soit des "pipes Unix" soit des sockets. Cette deuxième solution permet donc également de placer les deux systèmes sur des machines différentes.

Le système multi-processus a également servi pour incorporer à l'interface graphique des capacités fournies par une librairie graphique non utilisée dans Centaur. En effet l'interacteur utilisé pour chercher les fichiers à ouvrir est développé en Tcl/Tk par L. Rideau sur une idée que nous avons proposée et la communication avec cet interacteur repose sur un protocole textuel de communication analogue à celui utilisé pour communiquer avec le système de preuve. Ceci montre que la maîtrise de la communication inter-processus permet d'incorporer à peu de frais des capacités fournies par d'autres programmes.

Intuitivement, il n'est pas nécessaire de modifier le programme de preuve et l'interface graphique peut se content d'envoyer des commandes et d'analyser le résultat. En pratique, le système de preuve est développé en pensant à un utilisateur humain, et les réponses sont parfois trop ambiguës pour être comprises par le processus d'interface utilisateur. Une partie importante de notre travail a donc été de modifier les couches de communications du système de preuve pour pallier ce défaut. Les points principaux que nous avons développés sont les suivants:

- Chaque commande doit être précédée et suivie par des lignes spéciales. Ceci permet d'assurer que l'analyseur syntaxique du système de preuve se remet dans un état sain entre chaque commande, même si l'une des commandes contient une erreur de syntaxe.

- On assure également que chaque commande donne lieu à un accusé de réception. Ce point est important, car plusieurs des commandes du système de preuve sont muettes.
- En plus des lignes spéciales, chaque commande est accompagnée d'un identificateur de commande. Cet identificateur permet de mettre en place un mécanisme pour assurer la cohérence entre l'interface et le système de preuve.
- Toutes les données représentant des données structurées sont imprimées en utilisant un protocole de transfert d'arbres, à la place des procédures d'affichage usuelles qui produisent un texte plus difficile à analyser syntaxiquement.

Un problème particulièrement ardu dans la construction de l'environnement CtCoq provient de la possibilité fournie à l'utilisateur par le système Coq d'introduire ses propres notations, sous formes de règles d'analyse syntaxique qui sont prises en compte au fur-et-à-mesure par l'analyseur syntaxique du système de preuve. Pour assurer que l'environnement CtCoq accepte dans les mêmes conditions les fichiers reposant sur des extensions syntaxiques, il fallait s'assurer que l'analyseur syntaxique soit compatible avec celui de Coq. Dans l'approche traditionnelle, Centaur utilisait Yacc pour construire ses analyseurs syntaxiques. Dans la version de CtCoq adaptée à la version 5.8 de Coq, un effort considérable avait été fourni pour assurer cet objectif. En présence d'un analyseur syntaxique extensible, l'approche basée sur Yacc n'est plus possible, car Yacc ne permet pas de développer un analyseur syntaxique extensible. Le développement d'un autre analyseur syntaxique n'était pas raisonnable non plus.

Pour résoudre ce problème, nous avons détaché l'analyseur syntaxique de Coq en récupérant uniquement les modules reliés à l'analyse syntaxique. Le programme ainsi obtenu ressemble ainsi à un Coq qui aurait subi une lobotomie : il n'est plus capable que d'effectuer des analyses syntaxiques. Il reste à assurer que l'analyseur syntaxique se met à jour à chaque fois que le système de preuve charge une extension syntaxique. Pour obtenir ce résultat, il est nécessaire que l'analyseur syntaxique charge tous les modules que charge également Coq, car ces modules peuvent contenir des extensions syntaxiques. Ainsi, on peut dire que l'analyseur syntaxique effectue une interprétation faible des commandes exécutées par Coq, au sens où il interprète uniquement les commandes de chargement de module et les commandes d'extension syntaxique.

Un dernier point à mentionner dans l'organisation à plusieurs processus de l'environnement de preuve est l'implémentation des procédures utilisées pour transmettre des données structurées depuis le système de preuve vers l'interface. Ce transfert de données utilise un protocole d'arbre basé sur un parcours postfixé de l'arbre: pour transmettre l'arbre $f(t_1,\ldots,t_k)$, on produit d'abord le texte représentant les arbres t_1 à t_k, puis l'on transmet le texte suivant:

```
n
f
k
```

pour indiquer que l'on va construire un arbre de k fils dont l'opérateur de tête est `f`.

En fait, les structures de données manipulées dans le système de preuve sont différentes des structures de données manipulées dans l'interface graphique. Les structures de données manipulées dans le système de preuve sont trop permissives par rapport aux structures qu'il est effectivement possible de construire avec l'analyseur syntaxique. En revanche, les structures de données manipulées dans l'interface graphique sont beaucoup plus fidèles aux contraintes effectivement imposées par l'analyseur syntaxique. Il est donc également nécessaire d'effectuer une traduction d'une structure de donnée vers l'autre. Le code développé à cet effet remplit donc deux missions:

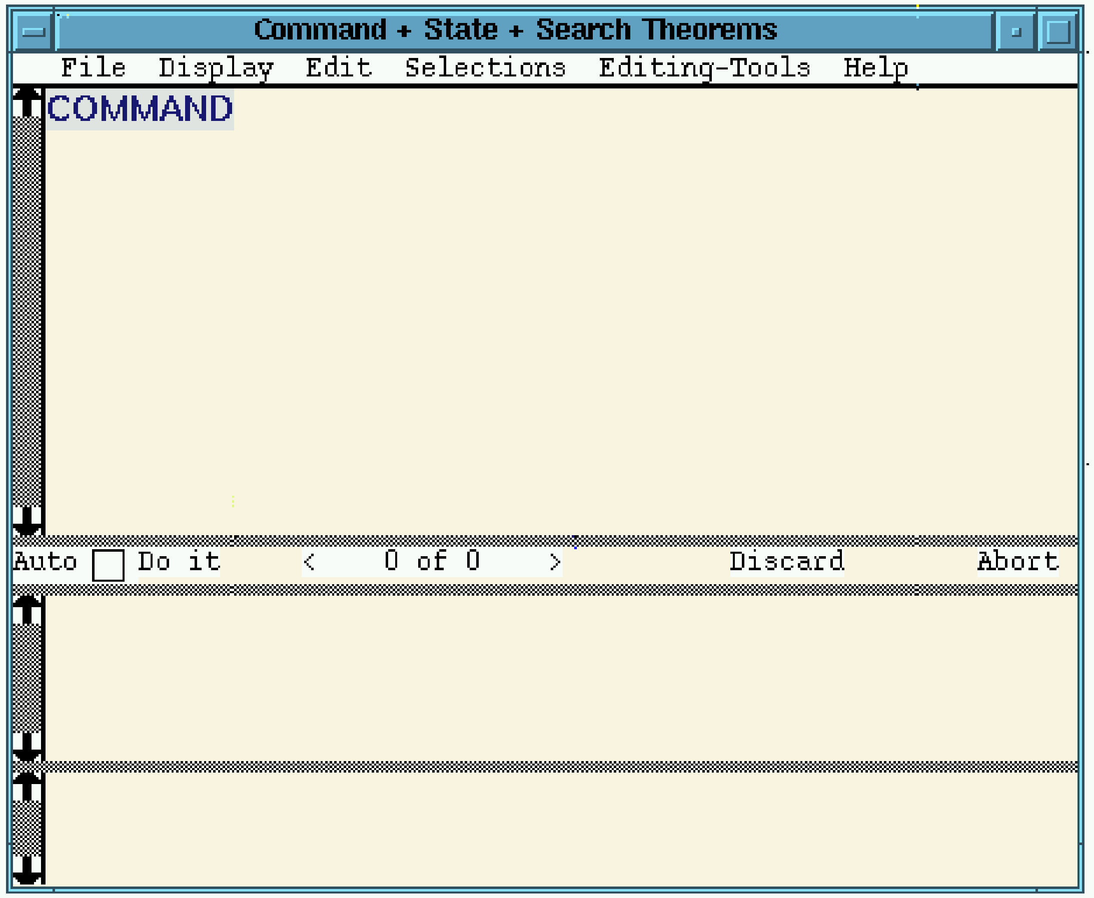
1. traduire les données de la structure employée dans le système de preuve,
2. produire le texte correspondant au protocole de transfert d'arbres pour la structure de données utilisée dans l'interface graphique.

L'approche naturelle serait de produire directement le texte final à partir de la structure de données du système de preuve. En pratique, cette approche est très fragile. Le texte produit peut aisément contenir des erreurs que l'on ne saura détecter qu'à l'utilisation, car il est difficile de prévoir la forme du texte produit seulement en observant le code qui le produira. Pour éviter cette fragilité, nous avons développé une solution qui rend plus facile la vérification du du code.

Dans cette solution, nous avons construit un outil pour traduire automatiquement une structure de donnée ML à partir d'une structure de donnée Centaur. Le résultat de cette traduction fournit d'une part la declaration d'un type ML et d'autre part des fonctions qui produisent du texte correct à partir de cette structure ML. Ainsi, le code utilisé pour produire du texte suivant le protocole est construit automatiquement et est pratiquement assuré d'être correct. Pour implémenter complètement les fonctions d'impression d'arbre, il reste à produire le code qui traduit les structures de données du système de preuve vers le type ML obtenu par traduction des structures de données de l'interface graphique. Il s'agit alors d'écrire des fonctions qui retourne des données fortement typées : le vérificateur de types de ML permet alors de vérifier partiellement la correction du code obtenu. Cette solution a donc permis d'augmenter de façon notable la robustesse de la communication entre les deux processus.

3.3 Fenêtre composite

L'environnement CtCoq fournit une fenêtre différente pour chaque catégorie de données qui apparait dans l'interaction avec le système de preuve. Ainsi, il y a une fenêtre qui contient les commandes envoyées au système de preuve, une fenêtre qui contient les buts, c'est à dire les énoncés à prouver, et une fenêtre qui contient les théorèmes, c'est-à-dire les énoncés déjà prouvés (ces théorèmes n'apparaissent que suite à des requêtes de recherche). Dans les premières versions, ces fenêtres étaient géographiquement indépendantes. Très rapidement, l'intérêt d'associer ces fenêtres en une seule fenêtre composite est apparu. Après un petit nombre d'essais, il est apparu qu'un très bon arrangement était celui qui apparait dans la figure suivante:



La fenêtre placée le plus haut contient les commandes envoyées au système de preuve. La barre intermédiaire contient quelques boutons d'usage très fréquent. La fenêtre placée au milieu contient les buts Enfin, la fenêtre placée le plus bas contient les théorèmes. Cette disposition permet d'atteindre gracieusement deux objectifs:

- réunir au centre les données les plus importantes,
- assurer la proximité géographiques de données reliées.

En effet, si l'on regarde la fenêtre du haut, les données placées en bas (près du centre de l'écran) sont souvent les plus intéressantes car ce sont les commandes que l'on vient d'envoyer. La fenêtre des buts est évidemment la plus importante, puisqu'elle présente la formule à prouver (l'objectif du travail en cours). Pour le contenu de cette fenêtre, il faut noter que les buts sont affichés avec la formule à prouver en haut et le contexte local en bas. Grâce à cette disposition, les hypothèses sont placées près des théorèmes qui apparaissent dans la troisième fenêtre. Cette proximité est naturelle: hypothèses et théorèmes s'utilisent pratiquement de la même manière.

Du point de vue des interactions entre les différentes zones de la fenêtre composite, la fenêtre des commandes joue un rôle plus important que la fenêtre des buts. Il y a en fait deux sortes de commandes envoyées au système Coq. La première catégorie correspond à des commandes d'interrogation. La seconde catégorie correspond à des commandes qui modifie l'état du système de preuve. Il est impératif que les commandes de la deuxième catégorie soit conservées dans la fenêtre des commandes. En utilisant des interactions à la souris, il est possible d'envoyer des commandes au système de preuve depuis toutes les fenêtres. Si les commandes sont de la première catégorie, elles sont simplement envoyées. En revanche, si les commandes sont de la deuxième catégorie, elles sont seulement insérées dans la fenêtre des commandes. De ce point de vue l'ensemble de la fenêtre composite est organisée comme si les deux autres fenêtres étaient des menus aidant à l'édition du contenu de la fenêtre des commandes.

3.4 Sessions multiples

L'utilisation de fenêtres composites permet également d'effectuer plusieurs preuves simultanément, en gardant dans les données nécessaires dans plusieurs fenêtres composites. Pour fournir cette possibilité, CtCoq suit le modèle de mini-sessions indépendantes. Les résultats de commandes envoyées depuis une fenêtre sont affichées dans cette fenêtre. Les commandes sont enregistrées en séquence dans une fenêtre sans tenir compte des commandes envoyées depuis une autre fenêtre.

En réalité, l'information d'ordonnancement entre les commandes des différentes fenêtres est conservée dans une structure de données globale, ce qui permet quand même de respecter l'ordre des théorèmes et définitions entre plusieurs fenêtres. Ainsi si l'on a défini l'objet A dans la fenêtre numéro 1, puis l'objet B dans la fenêtre numéro 2, puis l'objet C dans la fenêtre numéro 1 et D dans la fenêtre numéro 2, on a quand même le droit d'utilser A pour définir B et D et C pour définir D. Bien sûr, la gestion du retour en arrière tient compte de cet ordonnancement. Si l'utilisateur décide d'annuler la définition de B les objets C et D sont aussi annulés, même s'ils ne sont pas dans la même fenêtre que B et A est bien conservé.

Si l'on veut assurer une forme de stabilité à l'environnement de travail, il faut aussi fournir la possibilité de réduire le nombre de fenêtres composites en joignant le contenu de deux d'entre elles. Le problème important est de déplacer des fragments de scripts d'une fenêtre à l'autre en respectant l'ordonnancement. Nous fournissons cette possibilité en permettant de déplacer une définition ou une preuve entière d'une fenêtre vers une fenêtre parente. L'insertion de la définition dans la fenêtre parente n'est pas nécessairement faite à la fin du script que cette fenêtre contient mais à l'endroit qui correspond au moment d'exécution de cette définition. Une fois que la fenêtre fille est vide, elle peut être détruite.

Pour assurer que le modèle de "mini-sessions" fonctionne correctement, il faut que les données émises par le système de preuve en réponse aux commandes d'une fenêtre reviennent dans cette fenêtre. Pour assurer ce bon routage, les commandes émises reçoivent un argument supplémentaire : un identificateur de requête. Ensuite, les réponses portent toutes cet identificateur de requête. Ces réponses sont envoyées à toutes les fenêtres composites, mais seule la fenêtre composite qui possède le bon identificateur de requête va réagir au message.

Certaines fenêtres fournisse un service commun à toutes les fenêtres composites. Il s'agit de la fenêtre pour l'affichage des preuves sous forme textuelle et de la fenêtre utilisée pour afficher un ancien but d'une preuve en cours. Ainsi, il existe à tout moment au plus une fenêtre d'affichage de preuves textuelles à l'écran et les commandes d'affichages de preuves textuelles émanant de toutes les autres fenêtres ont toutes un effet sur cette fenêtre.

3.5 Fiabilité et assurance

Pour que le système soit réellement utilisé, il est nécessaire d'assurer que les données construites lors d'une longue session ne seront pas perdues malencontreusement. Pour diminuer ce risque, nous avons également pris le temps d'installer un système de sauvegarde automatique. Ce système fonctionne comme un service relié à une horloge. Lorsqu'une nouvelle fenêtre est créée, elle peut s'enregistrer sur le service simplement en entrant le réseau correspondant et s'apprêtant à recevoir le message adéquat. Lorsque l'horloge déclenche la sauvegarde, la fenêtre reçoit le message et provoque l'impression sur fichier des données qu'elle contient. Ensuite, l'horloge est réarmée.

Une source importante de faiblesse pour CtCoq est la limitation en mémoire. Lorsque la mémoire vient à manquer, ce système n'est pas capable d'allouer de la mémoire supplémentaire pour continuer à fonctionner normalement. A plusieurs reprises, ce défaut a provoqué des pertes importantes de données. Pour pallier ce défaut, il est important de détecter le manque de mémoire assez tôt pour permettre à l'utilisateur de prendre ses dispositions, par exemple en fermant des fenêtres contenant des données stables depuis longtemps. Lorsque le manque de mémoire est sévère, prévenir l'utilisateur ne sert à rien: le système risque de tomber en panne au milieu des opérations de sauvegardes manuelles qu'il pourrait entreprendre. Dans ce cas, nous préférons provoquer une sauvegarde des données du même type que les sauvegardes automatiques. Un moyen pour assurer que cette sauvegarde pourra fonctionner normalement est de "réserver" de la mémoire dans une variable et de la libérer juste avant de déclencher la sauvegarde automatique puis de terminer la session.

3.6 Implémentation hors Centaur

Une caractéristique importante de CtCoq par rapport aux autres expériences de développement d'interfaces pour les systèmes de preuve est la pré-dominance de la manipulation de données structurées. Cette pré-dominance permet des mécanismes d'interaction très puissant, comme le mécanisme de *preuve par sélection* présenté au chapitre suivant, mais le système Centaur est en retard sur d'autres outils d'édition de document dans d'autres domaines. Ainsi, l'éditeur de texte programmable Emacs permet également de fournir un support spécifique au langage très puissant, tout en bénéficiant d'une plus large panoplie d'outils annexes (comme les outils de sauvegarde automatique, la gestion de processus externes, la documentation en ligne, la collaboration avec des outils de gestion de version, etc).

Pour profiter des avantages d'un outil comme Emacs [25], il est nécessaire de relâcher l'attachement à la manipulation de données structurées acquis dans la pratique de Centaur. Néanmoins, un outil comme le mécanisme de preuve par sélection repose complètement sur la possibilité d'accéder avec la souris à la structure des formules logiques. Nous avons monté des expériences pour comprendre comment cette perception structurée pouvait être intégrée dans des environnements où les documents sont principalement des données textuelles [20].

Ces deux expériences ont été menées en utilisant XEmacs, une version de l'éditeur emacs particulièrement adaptée pour utiliser les possibilités des stations graphiques, et Tcl/Tk [55], un outil qui allie une librairie graphique très compléte avec un langage interprété basé sur la manipulation de chaînes de caractères.

Dans les deux cas, nous avons utilisé la possibilité de marquer des régions de texte avec une étiquette différente pour chaque région. Pour donner l'illusion de la manipulation structurée, il ne reste plus qu'à programmer l'interface pour que le click souris déclenche l'opération d'envoyer une commande au système de preuve, avec cette étiquette comme argument.

Du coté du système de preuve deux opérations sont nécessaires. En premier lieu, le système de preuve doit fournir l'information de structure en même temps que le texte représentant la formule logique. Avec cette information de structure, les régions adéquates sont activées dans la fenêtre où est affichée la formule logique. En deuxième lieu, il faut décoder la commande envoyée par la fenêtre lorsque la souris est cliquée dans l'une des régions actives.

De cette manière on peut développer des mécanismes comme la preuve par sélection sans faire appel à un environnement dédié à la manipulation structurée comme Centaur. Ce résultat est encourageant car il montre que les résultats acquis dans l'environnement Centaur pourront s'appliquer dans des contextes très variés.

3.7 Conclusion

Evidemment le travail effectué dans CtCoq est aussi le travail d'une équipe. Les idées initiales ont été expérimentées par Laurent Théry dans son travail de thèse. Tout le travail autour des fenêtres composites a été effectué par Janet Bertot, aussi bien pour les aspects graphiques que pour le mécanisme d'ordonnancement entre les commandes issues de plusieurs fenêtres composites. L'attention particulière apportée à la position relative des différents composants doit beaucoup à Gilles Kahn. Dans cette équipe, j'ai effectué tantôt un travail de direction, en prenant les décisions majeures d'organisation et en proposant certains développements, soit à des doctorants, soit à d'autres chercheurs ou ingénieurs du projet CROAP, tantôt un travail de découverte de nouveaux concepts, comme pour les travaux sur la gestion de script et l'interprétation des mouvements, tantôt un travail de tâcheron, comme la travail nécessaire pour assurer la pérennité de CtCoq à travers les évolutions de Coq. De cette expérience, nous pouvons quand même tirer quelques enseignements importants.

On peut s'interroger sur les raisons pour lesquelles l'effort autour de CtCoq a pu se développer sur une aussi grande durée et en mobilisant une équipe aussi large. La différence principale avec les expériences de L. Hascoët et J. Despeyroux a été d'utiliser un système de preuve externe, plutôt que d'implémenter un nouveau moteur. Ces systèmes souffraient dès le début du besoin de se constituer une communauté d'utilisateurs. La différence avec l'environnement cHOL développé par L. Théry est plus difficile à cerner, mais la remarque précédente peut encore s'appliquer. En effet cHOL ne répond pas bien aux attentes des utilisateurs de HOL car il ne permet pas de couvrir l'ensemble du langage ML. Par rapport à ces prédécesseurs, l'environnement CtCoq se démarque par une volonté réelle de s'adresser à l'ensemble des utilisateurs de la communauté de Coq, en couvrant complètement le langage de commande de Coq et en fournissant un environnement le plus proche possible de l'environnement traditionnel. Les utilisateurs de CtCoq peuvent donc échanger des scripts avec tous les utilisateurs de Coq. Cette compatibilité presque complète représente un effort considérable, mais cet effort est compensé par la possibilité d'attirer et de conserver des utilisateurs intensifs.

Les expériences sur les environnements de preuve utilisant Emacs ou Tcl/Tk semblent indiquer que l'effort d'utiliser Centaur n'était pas justifié et que le développement aurait aussi bien pu être effectué complètement avec ces outils. Ce point de vue est excessif dans la mesure où il néglige la richesse des inspirations provoquées par le point de vue défendu dans Centaur. Le fait de disposer en permanence d'un accès rapide aux données structurées dans Centaur permet de monter très rapidement des expériences basées sur la manipulation de ces données. Ainsi, le développement des premières versions de preuve par sélection a pris quelques jours. Pour le mécanisme de manipulations algébriques par déplacement présenté plus loin, le temps nécessaire pour mettre en place l'expérience initiale est de quelques heures. Ainsi, Centaur est un outil d'expérimentation irremplaçable : il permet d'étudier des problèmes d'informatique en se plaçant directement au niveau d'abstraction adéquat.

Chapitre 4

Preuve par sélection

Cette partie décrit une avancée spectaculaire dans le domaine des interfaces pour systèmes de preuve à laquelle nous avons participé. Il s'agit de la possibilité de guider les preuves simplement en sélectionnant les sous-expressions importantes de la formule à prouver. Ce mécanisme de preuve par sélection a été réellement introduit par Laurent Théry dans ses expériences autour du système de preuve d'Amy Felty et du système de preuve HOL. Pour l'interface du système de preuve d'Amy Felty, le mécanisme de preuve par sélection était même le seul moyen de preuve fourni à l'utilisateur. Nous avons contribué à une meilleure compréhension de ce mécanisme en étudiant son implémentation dans des interfaces pour les systèmes Isabelle [56], Coq [35] et Lego [52].

Dans le premier cas, nous avons étudié les aspects suivants:

- Comment éviter de nombreuses recherches dans le contexte des buts?
- Comment fournir une implémentation extensible, en reposant sur une implémentation en ML?

Dans le second cas, nous avons étudié les aspects suivants:

- Comment maitriser l'explosion du nombre d'hypothèses construites?
- Comment profiter de la forme systématique des tactiques pour construire des expressions plus concises?
- Comment combiner les actions de preuve par sélection avec les commandes de preuve usuelles du système?

Dans un premier temps, nous donnerons un exemple du type d'interaction permise par ce mécanisme de preuve par sélection. Puis nous décrirons successivement nos résultats pour chacun des aspects cités ci-dessus.

4.1 Exemples de preuve par sélection

Les exemples donnés dans cette section utilisent la syntaxe utilisée dans Coq, mais sont également valables pour les autres systèmes de preuve.

Considérons un but de la forme suivante, ou l'expression encadrée représente l'expression désignée par l'utilisateur et supposons que l'utilisateur sélectionne l'option indiquant qu'il veut faire une réécriture.

$\exists y : nat.p + 1 = 2 * y + 2$

$H : \exists x : nat.\boxed{p} = 2 * x + 1$

L'algorithme de preuve par sélection engendre alors la commande suivante:

```
Elim H; Intros x E; Rewrite E;Clear H.
```

Cette commande permet de passer au but suivant:
$\exists y : nat.(2 * x + 1) + 1 = 2 * y + 2$

$E : p = 2 * x + 1$
$x : nat$

Ainsi, l'algorithme de preuve par sélection permet en une étape de donner un nom à un témoin pour la formule existentielle et d'effectuer une réécriture avec l'égalité qu'elle contient. Comme nous l'indiquons dans l'article [19], le mécanisme de preuve par sélection permet d'isoler l'expression sélectionnée pour la donner en argument à une commande choisie.

Un autre exemple caractéristique d'utilisation est donné avec le but suivant:
$C \vee (\boxed{B} \vee A) \Rightarrow A \vee (B \vee C)$

Pour isoler l'expression sélectionnée, le mécanisme de preuve par sélection doit décomposer complètement l'implication et les disjonctions qui la contiennent. Ainsi, la commande engendrée brise les disjonctions et isole en même temps les trois propositions atomiques, dans trois buts indépendants.

```
Intro H;Elim H;
    [ Intro H1;Clear H |
          Intro H1;Elim H1;[Intro H2; Try Exact H2; Clear H H1 |
                            Intro H2; Clear H H1]]
```

Cette commande remplace le but initial par trois buts dont le premier a la forme suivante:
$A \vee (B \vee C)$

$H1 : C$

En continuant la preuve par la sélection de la proposition C dans la conclusion, on peut guider le mécanisme de preuve vers l'expression qu'il sera possible de prouver. La commande engendrée a la forme suivante:

```
Left;Left;Try Assumption.
```

En l'occurrence, cette commande suffit à résoudre ce but. On peut alors continuer avec les deux autres buts.

4.2 Description abstraite du mécanisme

L'algorithme de preuve par sélection se décrit aisément comme une instrumentation des règles d'inférences de la déduction naturelle telles qu'elle sont décrites dans les travaux de Gentzen [64]. Il s'agit de preuve dirigée par les buts et le raisonnement *en arrière*. Ainsi, on remonte de la conclusion des règles d'inférences vers leurs prémisses. L'algorithme de preuve par sélection fonctionne en choisissant une règle d'inférence parmi les différentes possibles et en indiquant la prémisse importante (lorsqu'il y en a plusieurs).

Prenons par exemple l'une des règles qui indiques comment fonctionne la preuve par sélection pour une hypothèse disjonctive:

$$\frac{\boxed{A} \vee B, \boxed{A} \vdash C}{\boxed{A} \vee B \vdash C}$$

Pour clarifier les notations, rappelons qu'un séquent de la forme $A \vdash C$ représente un but dont la conclusion (la formule à prouver) est C et dont le contexte (les hypothèses locales) contient au moins une hypothèse dont l'énoncé est A. Dans la règle d'inférence ainsi présentée, le séquent apparaissant en bas de la barre horizontale correspond au but avant l'inférence, les séquents qui apparaissent au dessus de la barre horizontale (ici, il n'y en a qu'un) correspondent aux buts après l'étape de raisonnement. Enfin, l'opération de sélection effectuée par l'utilisateur est représentée par l'expression encadrée dans la conclusion et la prémise. Ainsi, la conclusion indique que le but contient une hypothèse de la forme $A \vee B$ telle que l'expression sélectionnée est à l'intérieur de l'expression A. La prémisse de la règle d'inférence indique que le mécanisme de preuve par sélection doit continuer récursivement sur le but créé avec la même sélection dans l'expression A.

4.3 Recherche d'hypothèses dans le contexte

Le deuxième exemple donné dans l'expression précédente montre que plusieurs hypothèses sont construites et insérées dans le contexte du but et que la preuve par sélection progresse en utilisant ces nouvelles hypothèses. Les exemples présentés ici utilisent des noms pour faire référence aux hypothèses effectivement utilisées. Dans l'expérience que nous avons effectué pour implémenter la preuve par sélection pour le système de preuve Isabelle, le contexte est considéré comme un ensemble de formules logiques non nommées. Il est donc normalement impossible de récupérer de façon prédictible les dernières hypothèses introduites.

Pour pallier cet inconvénient, nous avons mis au point une technique qui utilise une implication pour isoler l'hypothèse importante à chaque étape de raisonnement. Chacune des règles de raisonnement utilisée par l'algorithme de preuve par sélection pour travailler dans les hypothèses est remplacée par une règle de raisonnement travaillant sur un but avec une implication. Par exemple, la règle de fonctionnement pour une hypothèse disjonctive donnée dans la section précédente est transformée en la règle suivante:

$$\frac{A \vee B \vdash \boxed{A} \Rightarrow C}{\vdash \boxed{A} \vee B \Rightarrow C}$$

De cette façon, il n'est pas nécessaire de rechercher l'expression A dans le contexte à la prochaine étape, puisque cette expression est conservée, comme dans un registre.

4.4 Implémentation extensible en ML

Dans l'implémentation de la preuve par sélection nous avons expérimenté avec plusieurs langages de programmation et plusieurs mode de fonctionnement. Ainsi, il est possible d'implanter le mécanisme dans l'interface sans que le système de preuve soit conscient de son existence. C'est ce que j'ai fait pour CtCoq et ce que L. Théry Théry a effectué pour l'interface du système de preuve d'A. Felty. Dans ce cas, l'algorithme est écrit en Lisp. Il est également possible d'implanter le mécanisme de preuve par sélection dans le système de preuve. C'est ce que j'ai fait pour le système Isabelle (déjà citée plus haut et décrite dans [19]) et pour l'expérience avec Lego (décrite dans [20]). Dans ces deux cas l'algorithme a été décrit en ML.

Le langage de programmation ML fournit des mécanismes de filtrage très puissant et bien adaptés à la programmation d'un tel algorithme. Le réflexe naturel est alors de programmer l'ensemble du mécanisme comme une grosse fonction récursive définies par cas, où chaque cas correspond à l'une des règles d'inférence. La fonction de preuve par sélection prend alors approximativement la forme suivante:

```
let rec preuve_par_selection formula path auxiliary_information =
  match (formula,path) with
  (Hypothese(h,Or(a,b)),1::path') ->
       let h' = new_name auxiliary_information in
       let aux' = add_new_name h' auxiliary_information in
       let resultat_recursif =
             (preuve_par_selection Hypothese(h',a) path' aux')
         in compose [Elim(h);Intro(h');resultat_recursif]
  ...
  _ -> terminal_action formula path auxiliary_information;;
```

Ainsi, chaque cas effectue une opération de filtrage sur la formule et le chemin fourni, puis rappelle la fonction récursive sur la prémisse déduite. Une donnée auxiliaire est également transmise lors des appels récursifs. Cette donnée auxiliaire peut par exemple contenir l'ensemble des noms d'hypothèses déjà utilisés. Cette présentation est relativement lisible mais elle n'est peu extensible. En effet il est nécessaire de recompiler toute la fonction lorsque l'on veut ajouter un nouveau cas (par exemple pour un nouvel opérateur logique).

Pour ajouter de l'extensibilité à ce développement nous avons imaginé une solution qui permet de séparer les règles et le moteur de sélection. Le moteur de sélection est alors une fonction qui prend en argument la liste des règles de comportement et appelle successive chacune de ces règles jusqu'à ce qu'une d'elles retourne un résultat significatif. Chaque règle est elle-même une fonction de type similaire, avec les deux différences suivantes:

1. chaque règle prend en argument supplémentaire la fonction totale, utilisée comme fonction de continuation pour les appels récursifs,
2. le type retourné est un type optionnel qui permet d'indiquer quand la règle échoue.

Le code du moteur d'inférence est alors le suivant:

```
type 'a option = Some of 'a | None;;

let rec preuve_par_selection formula path auxiliary_information =
    let rec preuve_rec = function
      rule1::rule_tail ->
        (match rule1 formula path auxiliary_information
                 preuve_par_selection with
             Some(r) -> r
           | None -> preuve_rec rule_tail)
      [] -> terminal_action formula path auxiliary_information in
      preuve_rec (!mutable_rule_list)
```

Le fait d'utiliser une variable mutable (c'est-à-dire réaffectable) pour contenir la liste des règles permet alors d'étendre le mécanisme de preuve par sélection à la volée, en y ajoutant de nouvelles règles.

Le code d'une règle prend alors la forme suivante:

```
let  regle_or_hypothese_gauche
     formula path auxiliary_information continuation =
   match (formula,path) with
      (Hypothese(h,Or(a,b)),1::path') ->
        let h' = new_name auxiliary_information in
        let aux' = add_new_name h' auxiliary_information in
        let resultat_recursif =
              (continuation Hypothese(h',a) path' aux') in
           Some(compose[Elim(h);Intro(h');resultat_recursif])
       _ -> None;;
```

Nous avons volontairement décrit la même règle dans cet exemple et dans l'exemple décrit plus haut où tous les cas sont réunis dans une même fonction. On voit que la transformation d'une règle à l'autre est très systématique. En fait, ce codage suggère une méthode générale pour rendre des fonctions définies par cas dynamiquement extensibles. En revanche, on peut remarquer que cette technique peut avoir des effets négatifs en termes de complexité: en effet, la complexité est au moins linéaire en la taille de la liste de règles. Une implémentation réellement efficace devrait combiner cette technique avec une pré-compilation du filtrage d'arbres. Ce style de compromis existe déjà dans d'autres contextes, comme par exemple l'implémentation modulaire de PPML dans le système Centaur[1].

4.5 Maitriser le nombre d'hypothèses construites

Intuitivement, le mécanisme de preuve par sélection effectue une descente récursive dans la formule sélectionnée en effectuant à chaque pas une opération significative. Si

1. citation?

cette descente récursive crée à chaque pas une nouvelle hypothèse, on peut se retrouver avec une grande quantité d'hypothèses contenant des information redondantes. Par exemple si la formule logique est une hypothèse de la forme $a_1 \vee \cdots \vee \boxed{a_n}$ et que l'on sélectionne a_n, on aura les n hypothèses $a_1 \vee \cdots \vee a_n$, $a_2 \vee \cdots \vee a_n$, $\ldots a_n$ alors que toutes sont une conséquence de la dernière!

Pour pallier cet inconvénient, nous sommes intervenus auprès de l'équipe Coq pour que le système de preuve contienne une commande `Clear` qui permet d'effacer une hypothèse. Ensuite nous avons modifié les règles décrivant le mécanisme de preuve par sélection en suivant le principe suivant:

A chaque fois qu'une règle d'inférence introduit des hypothèses dont le contenu suffit à prouver l'hypothèse en cours, cette dernière est détruite.

Ainsi la règle de comportement que nous utilisons en exemple est transformée en la règle suivante:

$$\frac{\boxed{A} \vdash C}{\boxed{A} \vee B \vdash C}$$

Pour éviter d'obscurcir les expressions de preuve, les instructions d'effaçage sont réunies en fin d'expression, au moyen d'une liste de noms qui fait partie des données auxiliaires transmises à chaque appel récursif.

4.6 Optimisation des expressions de preuve

Un défaut important du mécanisme de preuve par sélection est que les commandes qu'il construit ne sont pas les mêmes qu'un utilisateur écrirait à la main. Les commandes produites n'utilisent en effet que les tactiques primitives qui implémentent les règles d'inférence de la déduction naturelle et aucune des tactiques puissantes du système de preuve. Les expressions de preuve obtenues sont donc des assemblages complexes d'éléments primitifs, alors que les utilisateurs humains auront tendance à utiliser des assemblages plus simples construit avec des tactiques plus élaborées.

Dans le travail autour de CtCoq, nous avons étudié les équivalences entre les tactiques primitives utilisées pour la preuve par sélection et les tactiques plus généralement utilisées dans l'usage normal de Coq. Cette étude, décrite dans [14] a permis de dégager des équivalences faibles entre les schémas de raisonnement habituellement construits par le mécanisme de preuve par sélection des tactiques plus élaborées. Nous employons le terme d'équivalence faible car nous voulons profiter de la similarité entre deux expressions de preuve même si ces expressions diffèrent par des l'ordre des buts engendrés et le nombre d'hypothèses redondantes créées.

Ce travail sur l'équivalence entre expressions de preuve basées sur les tactiques est intéressant car il mène à prendre du recul sur ce que doit un bon ensemble de tactiques primitives dans un système de preuve. Dans le cas de notre étude, ce travail a permis de montrer l'existence d'une famille de tactiques qui ont toutes un comportement similaire du point de vue du traitement de l'implication. Cette réflexion est importante car elle permet de guider la conception de nouvelles tactiques afin qu'elle s'intègrent harmonieusement dans la famille. De plus, elle permet le développement d'outils qui manipuleront les expressions à base de tactiques de façon cohérente.

4.7 Combinaison avec d'autres commandes de preuve

L'un des paradoxes du mécanisme de preuve par sélection est que l'expression sélectionnée n'a pas d'importance: c'est seulement la position de cette expression qui est interprétée. Cette remarque rend donc possible une extension supplémentaire de ce mécanisme avec cette fois-ci une utilisation effective de l'expression sélectionnée.

L'algorithme de preuve par sélection termine lorsqu'aucune règle ne reconnait la formule et le chemin fournis, très souvent parce que le chemin est vide. A ce moment, l'algorithme doit fournir une commande qui sera appelée sur le dernier but produit. C'est à cet endroit que nous pouvons choisir d'appliquer une commande dont le but sera d'utiliser effectivement l'expression sélectionnée. Dans CtCoq, nous avons baptisé ce genre d'extension : *point-and-shoot*, "visez et tirez". L'idée intuitive est que l'on choisit d'abord l'expression intéressante puis l'opération que l'on veut effectuer sur cette expression. Parmi les deux exemples donnés en section 4.1, le premier utilise la commande `Rewrite`, tandis que le second utilise la commande `Assumption`. En fait, la commande `Rewrite` n'a pas d'effet si l'expression sélectionnée n'est pas une égalité et si cette expression ne se trouve pas en position d'hypothèse lorsque le mécanisme de preuve par sélection a fini son travail. La commande `Assumption` a un effet différent si la commande sélectionnée est dans le but (on applique alors la commande `Assumption`) ou dans une hypothèse (on applique alors la commande `Exact H`, où `H` est le nom de l'hypothèse.

Dans l'article [21], nous avons également décrit un mécanisme de réécriture complexe qui permet d'utiliser des théorèmes d'injectivité. Ce mécanisme de réécriture s'applique dans le cas où l'expression sélectionnée se trouve dans une hypothèse de la forme suivante, où les fonctions f_1, f_2, ..., f_n sont injectives:

$$H : f_1(\ldots f_n(\boxed{x}),\ldots) \quad = \quad f_1(\ldots f_n(y)\ldots).$$

Si t_1, ..., t_n sont les théorèmes exprimant l'injectivité des fonction f_i:

$$f_i : \forall x{,}y.f(x) = f(y) \Rightarrow x = y,$$

il est alors possible de construire une commande de preuve de la forme suivante:

```
Specialize t1 with 1 := H; Intro H1;
...
Specialize tn with 1 := Hn-1; Intro Hn;
Rewrite Hn.
```

Cette commande déduit l'égalité $x = y$ de H et des théorèmes d'injectivité et effectue une réécriture avec cette dernière égalité.

Cette commande de réécriture complexe n'a jamais réellement été utilisée, car les fonctions injectives sont rares. Néanmoins, cette extension montre que le mécanisme de preuve par sélection peut se combiner avec d'autres mécanismes utilisant la sélection.

4.8 Conclusion

Le mécanisme de preuve par sélection est le plus spectaculaire des outils issus des recherches sur l'interface homme-machine des systèmes de preuve. Le véritable point fort de ce mécanisme est son caractère récursif, qui d'effectuer en une seule étape la manipulation de nombreux opérateurs logiques. De cette manière, la structure des formules logiques disparait pratiquement de la vue de l'utilisateur: les formules $a \wedge (b \wedge c)$ et $(a \wedge (c \wedge b)$ sont pratiquement équivalentes à manipuler, puisqu'il suffit de sélectionner c à la souris pour l'extraire, l'ordinateur se chargera du reste.

Bien sûr, ce mécanisme garde une portée très limitée: les expressions logiques qu'il permet de prouver entièrement restent confinées à la logique des propositions. Néanmoins, ce mécanisme s'intègre très bien au cadre général de la preuve dirigée par des buts et permet l'emploi d'autres tactiques ou procédures de décision sur les buts qu'il aide à préparer. En ce sens, le mécanisme de preuve par sélection est surtout un outil d'interface homme-machine pour préparer le terrain pour des outils automatiques. Cette remarque se traduit d'ailleurs par la façon dont ce mécanisme s'utilise dans CtCoq: la preuve par sélection sert à engendrer des commandes pertinentes, à charge à l'utilisateur d'envoyer ces commandes vers le système de preuve, éventuellement après modification.

De façon surprenante, ce travail se trouve dans un prolongement naturel de nos travaux sur le suivi de sujet en sémantique naturelle. En effet, on se ramène ici encore à un calcul de position autour de règles d'inférences. Dans les travaux sur la sémantique naturelle, l'information de position était déduite de l'arbre de preuve en cours de construction. Ici, l'information de position prend un caractère plus actif: l'information de position est maintenant utilisée pour guider la construction de l'arbre de preuve. Dans le prolongement du travail sur l'extensibilité du mécanisme de preuve par sélection, il serait intéressant de voir si ce mécanisme pourrait être utilisé également pour guider l'exécution de spécifications sémantiques. Ceci pourrait être intéressant surtout pour des spécifications sémantiques faisant intervenir du non-déterminisme, comme les spécifications de langages concurrents. Ainsi, sélectionner un processus permettrait d'indiquer que l'on veut voir l'exécution progresser prioritairement dans ce processus.

Chapitre 5

Interprétation des mouvements

Le modèle intuitif du mécanisme de preuve par sélection est de faire monter une sous-expression vers la racine du but logique[1]. Ainsi, la preuve par sélection décrit un mouvement de données où la destination est imposée. Dans les raisonnements algébriques, on est également amené à considérer des mouvements de données, mais il peut être nécessaire de laisser l'utilisateur choisir la cible. Dans ce chapitre nous allons présenter un outil où l'utilisateur exprime le mouvement à l'aide des déplacements de la souris [13].

Le contexte privilégié de ce travail est celui de la manipulation algébrique par réécriture. En effet, de nombreuses propriétés des opérateurs mathématiques peuvent être comprises comme des propriétés de mouvement : la commutativité, les éléments absorbants. Ces mouvements peuvent être exprimés en sélectionnant l'expression concernée et en la tirant vers une cible. Nous montrerons comment ce mécanisme peut être ajouté à l'environnement CtCoq grâce à un algorithme simple et extensible et nous nous attarderons sur les moyens d'ajouter de nouveaux comportements dans ce mécanisme. Enfin, nous réfléchirons aux possibilités d'élargir le contexte d'application de ce mécanisme.

5.1 Quelques catégories de raisonnements par déplacement

La première catégorie de raisonnements qu'il est intéressant d'exprimer par un déplacement de la souris sont les raisonnements qui font réellement intervenir des déplacements de données, comme dans les égalités suivantes:

$$\begin{aligned} x + y &= y + x \\ x + (y + z) &= y + (x + z) \\ x \times y &= y \times x \end{aligned}$$

Pour chacune de ces égalités, on peut exprimer la requête de réécrire une instance du terme de gauche par l'instance correspondante du terme de droite en déplaçant la souris de x vers y. En fait, la même requête peut également être exprimée en déplaçant la souris

1. Nos arbres, comme fréquemment en informatique, poussent vers le bas.

de y vers x. En effet, la structure du terme ne change pas et la réécriture de la droite vers la gauche est exactement la même opération que la réécriture de la gauche vers la droite.

De ce cas de figure, il est facile de passer au cas de l'équation de distributivité, qui fait encore intervenir un déplacement de données:

$$x \times (y + z) = x \times y + x \times z$$

Pour cette égalité, on peut encore exprimer la requête de remplacer le terme de gauche par le terme de droite en déplaçant la souris de x vers y. En revanche, la structure du terme change et le déplacement de y vers x ne semble pas intuitif pour effectuer la même opération. En fait, ce déplacement serait plutôt utilisé pour utiliser la commutativité de la multiplication, comme au cas précédent.

Pour cette équation, on peut également étudier la réécriture de la droite vers la gauche qui est différente cette fois-ci. Ce mouvement consiste à transformer une somme en produit et il est naturel d'exprimer cette modification en sélectionnant l'un des signes de multiplication "$\times$" et en le déplaçant vers le signe d'addition "$+$". Ainsi, le mouvement ne s'exprime en déplaçant la souris de l'une des variables du motif vers une autre variable du motif, mais les positions d'opérateurs dans ce motif sont aussi utilisables pour exprimer des opérations algébriques. En prenant ce point de vue, on peut se donner une autre méthode pour exprimer la réécriture du membre de gauche vers le membre de droite. En effet cette opération consiste à remplacer un produit par une somme, ce qui s'exprime en déplaçant l'un des deux opérateurs vers l'autre.

Dans le même style, on peut également s'intéresser à l'équation d'associativité:

$$x + (y + z) = (x + y) + z$$

Ici, on est loin des premiers exemples donnés dans cette section. La position relative des variables x, y et z ne change pas dans la réécriture et c'est au contraire la structure du terme qui change. On peut néanmoins exprimer la transformation en indiquant que l'on change l'importance relative des deux opérations d'addition. On peut alors exprimer cette réécriture en sélectionnant l'un des signes "$+$" et en relâchant sur l'autre. Un autre moyen d'exprimer cette transformation est de considérer que l'on déplace les parenthèses vers le signe "$+$" qui devrait les porter. Il suffit alors de sélectionner l'une des parenthèses (en fait c'est la somme complète qui sera sélectionnée) et de la relâcher sur l'autre.

Les dernières opérations qui s'expriment naturellement sont des combinaisons entre deux opérateurs qui mènent en quelque sorte à la consommation de ces opérateurs. Dans cette catégorie, on trouve les équations décrivant les propriétés d'éléments neutres ou absorbants ou d'involutivité:

$$\begin{array}{rcl} 0 + x & = & x \\ 0 \times x & = & 0 \\ -(-x) & = & x \end{array}$$

Dans chacun de ces cas, la réécriture du membre de gauche vers le membre de droite peut s'exprimer dans un mouvement qui démarre sur l'un des opérateurs et termine sur l'autre. Ainsi pour la première équation, la requête de réécriture se fait en sélectionnant le 0 et déplaçant la souris sur le signe $+$, pour la deuxième en sélectionnant le 0 et en déplaçant la souris sur le signe $\times$ et pour la troisième en sélectionnant l'un des signes $-$ et en déplaçant la souris sur l'autre.

5.2 Un algorithme pour la manipulation directe

L'implantation du mécanisme d'interprétation des mouvements passe par trois phases:

1. Reconnaître le mouvement effectué par l'utilisateur,
2. Sélectionner une règle de comportement,
3. Mettre en œuvre cette règle en tenant compte des données effectivement manipulées.

Pour reconnaître le mouvement effectué par l'utilisateur, nous nous reposons sur les primitives de manipulation structurée fournies par Centaur et les possibilités de réagir à des événements d'interaction. Nous utilisons la possibilité de calculer à tout moment le chemin depuis la racine de la formule logique jusqu'à l'expression qui se trouve sous le pointeur de la souris. Ainsi, notre calcul va s'effectuer sur la donnée de l'expression entière affichée (sous la forme d'un arbre), du chemin vers l'expression désignée lorsque le bouton a été enfonçé et du chemin vers l'expression désignée lorsque le bouton a été relâché. En effet de bord, les trois fonctions rendent visibles les expressions désignées, avec des couleurs différentes pour l'expression choisie au début de l'action et pour l'expression couramment désignée (dans sa configuration actuelle, le mécanisme utilise une couleur rose pour l'expression de départ et verte pour l'expression en cours). Avec les deux chemins désignant l'expression de départ et l'expression d'arrivée, il est possible de déterminer le plus petit ancêtre commun à ces deux expressions: c'est l'expression désignée par le chemin qui est le plus grand préfixe commun à ces deux chemins.

Chaque règle de comportement est décrite par un schéma d'arbre, un chemin de départ et un chemin d'arrivée et une action. Le schéma d'arbre est un arbre contenant des variables d'arbres, plus souvent appelées méta-variables car il ne faut pas les confondre avec les variables mathématiques apparaissant dans la formule représentée. Le chemin de départ et le chemin d'arrivée désignent des sous-expressions du schéma d'arbre. Pour choisir une règle il est nécessaire que le schéma d'arbre apparaisse dans la formule logique manipulée et que les expressions de départ et d'arrivée dans la formule logique correspondent aux expressions de départ et d'arrivée dans le schéma d'arbre.

Pour décrire le choix de la règle d'inférence de façon formelle, supposons que la formule logique manipulée est F et que les chemins de départ et d'arrivée dans cette formule soient désignés par p_i et p_f et supposons que la règle de comportement envisagée soit décrite par le schéma d'arbre P et les chemins p_1 et p_2. Pour que la règle soit sélectionnée il est nécessaire qu'il existe des chemins p, p' et p'' tels que $p_i = pp_1p'$ et $p_f = pp_2p''$ et la sous-expression de F désignée par p soit une instance de P. Pour trouver ce chemin p, le calcul du plus petit ancêtre commun à p_i et p_f joue un rôle important.

Lorsque la règle est sélectionnée, il reste à la mettre en œuvre. Dans la première version du mécanisme de manipulation, cette mise en œuvre s'effectue par une opération de réécriture. Pour respecter les demandes de l'utilisateur, il est nécessaire d'assurer que le théorème de réécriture va s'appliquer exactement à l'endroit désigné. Pour assurer cela, la première technique est d'instancier complètement le théorème de réécriture. Dans de nombreux cas, cette instanciation suffit à indiquer sans équivoque l'endroit où doit avoir lieu la transformation. Néanmoins, il peut arriver que la formule logique contienne plusieurs instances de la sous-expression à transformer. Il est alors possible d'insérer une commande `Pattern` pour restreindre le domaine où la réécriture peut avoir lieu.

Pour instancier le théorème on calcule la substitution σ telle que la expression de F à la position p soit égale à $\sigma(P)$. Il suffit ensuite d'appliquer cette substitution sur le schéma de commande donné comme action de la règle de comportement.

5.3 Interaction avec l'affichage et ambiguïtés

Comme dans le cas de la preuve par sélection, l'utilisation de la souris pour exprimer des mouvements dans les arbres repose fortement sur le mécanisme d'affichage. La méthode utilisée pour permettre la manipulation à la souris est en effet d'afficher l'arbre en gardant le moyen de remonter des fragments de texte affichés vers les nœuds de l'arbre. Le mécanisme d'affichage ne peut pas assurer que tous les nœuds de l'arbre seront rendus accessible par cette méthode. C'est pour cette raison que l'on impose pas que les p_i et p_f soient égaux aux chemins pp_1 et pp_2, mais seulement qu'ils soient plus "longs".

La présentation séparée des différentes équations dont l'utilisation en réécriture peut être requise par un déplacement cache un problème important: plusieurs opérations peuvent être effectuées pour une expression et un mouvement donnés. Ainsi si nous reprenons l'expression suivante:

$$x + (y + z)$$

Si l'utilisateur sélectionne la somme $y + z$ et déplace la souris vers x, ceci peut être compris comme une utilisation de la commutativité de l'addition, avec comme résultat l'expression suivante:

$$(y + z) + x$$

Alternativement, ce mouvement peut être compris comme une utilisation de l'associativité, avec comme résultat l'expression suivante:

$$(x + y) + z.$$

Nous avons deux solutions pour gérer ce type d'ambiguïté. Une solution serait d'ajouter une phase d'interaction supplémentaire après chaque mouvement ambigu.

L'autre solution est d'utiliser une méthode systématique pour donner priorité à certaines règles de façon prédictible. Dans l'outil que nous avons implémenté, les règles de comportement sont placées dans une liste ordonnée de telle sorte que la première règle trouvée est exécutée.

Lorsque l'on introduit une règle dans la liste, il est possible de masquer complètement le comportement décrit par une règle déjà présente. Pour cela, il suffit que le schéma d'arbre soit plus général et que les chemins soient plus longs. Nous devons tenir compte de cette caractéristique lorsque nous fournissons des moyens d'extensions manuels ou automatiques.

5.4 Extension du mécanisme

L'avantage le plus précieux de cette présentation sous forme de règles de comportement indépendantes est qu'il permet une extension rapide, simplement en ajoutant de nouvelles règles. Dans notre expérience nous fournissons deux moyens principaux d'extension: un moyen automatique par analyse de théorèmes et un moyen interactif par lequel l'utilisateur peut fournir le schéma d'arbre, les chemins de départ et d'arrivée et le schéma de commande correspondant à chaque règle de comportement.

L'outil d'analyse de théorèmes observe les énoncés et construit des règles de comportement dès que ces énoncés ont une forme adéquate. Les règles sont ensuite introduites dans la liste en faisant attention de ne pas masquer de règle plus spécifique. Les classes d'énoncés reconnus jusqu'à maintenant sont les énoncés d'élément neutre, de distributivité, de commutativité.

L'outil interactif d'extension comporte une fenêtre multiple avec trois régions importantes. La première région est prévue pour recevoir le schéma d'arbre qui devra être reconnu. L'interface fournit la possibilité d'insérer des meta-variables dans les expressions apparaissant dans cette fenêtre. Aussi, l'utilisateur peut indiquer les positions de départ et d'arrivée dans le schéma d'arbre simplement en simulant le mouvement reconnu par la règle sur ce schéma d'arbre: les positions de départ et d'arrivée apparaissent alors en couleur, avec les mêmes couleurs que dans l'utilisation réelle. Dans la deuxième région, l'utilisateur peut éditer la commande de réécriture qui sera déclenchée. Pour exprimer les données qui seront instanciées à partir du filtrage effectué par le schéma de formule logique, l'utilisateur peut se contenter d'insérer dans la commande des méta-variables de même nom que celles apparaissant dans le schéma de formule logique.

La troisième région affiche les noms de toutes les règles utilisées par le mécanisme de manipulation directe, dans l'ordre où ces règles apparaissent dans la liste. L'utilisateur peut mettre à jour la position relative des règles simplement en déplaçant leur nom dans la liste.

Cet outil d'édition interactive du comportement fournit un exemple intéressant de programmation "sans le savoir". En effet, l'utilisateur dispose ainsi d'un moyen d'extension assez puissant où la programmation s'effectue avec des concepts très simple. Il s'agit d'une programmation par l'exemple, puisque l'utilisateur doit seulement fournir un exemple de formule logique et un exemple de commande déclenchée. Enfin, le mouvement est bien décrit par l'exemple puisque l'utilisateur exprime ce mouvement en le mimant.

5.5 Conclusions

Le mécanisme de manipulation décrit dans cette section est d'un usage assez spectaculaire, mais rapidement fastidieux. Deux aspects semblent particulièrement faibles. D'une part ce mécanisme ne permet d'effectuer des raisonnements que pour des égalités de type "Leibniz" avec l'opération de réécriture. Cette technique ne permet pas, par exemple, de remplacer un but de la forme $x + y = x + z$ par un but de la forme $y = z$, ce qui serait une interprétation naturelle d'un mouvement de l'un des x vers l'autre. D'autre part, ce mécanisme ne permet pas de faire de grands déplacements d'un seul mouvement: chaque opération de réécriture décrit une transformation locale dans la structure de la formule logique. On rêverait de disposer d'une extension récursive du mécanisme, en s'inspirant du caractère récursif du mécanisme de preuve par sélection.

Chapitre 6

Gestion de scripts

6.1 Introduction

Alors que les outils de génération de commandes présentés dans les deux sections précédentes apportent une aide évidente et permettent de faire des démonstrations spectaculaires, l'outil de gestion de script que nous présentons dans cette section apporte une aide beaucoup plus discrète, mais peut-être beaucoup plus importante. Plutôt que de faire gagner du temps, cet outil va éviter d'en perdre, ce qui est souvent plus efficace, en pratique.

L'environnement de preuve se charge de conserver et de sauvegarder dans un script les commandes envoyées au système de preuve dans l'ordre où elles ont été envoyées. Cet ordre n'est pas parfaitement exact, car les commandes inutiles, les commandes erronées et les commandes défaites sont enlevées. Lorsque l'utilisateur effectue une preuve, la présence de l'outil se fait à peine remarquer: tout juste voit on les commandes envoyées changer d'aspect au cours de leur exécution. En revanche, le confort est sensible lorsque l'utilisateur revient sur ses pas: l'interface se charge pour lui d'écarter les commandes défaites, sans les perdre, et sans perdre les commandes non défaites.

C'est ici que l'outil de gestion de script évite la perte de temps. Puisque l'utilisateur n'a plus à se charger d'enlever les commandes devenues inutiles, il ne peut pas se tromper en le faisant. Dans ce travail, l'erreur est particulièrement ennuyeuse, car c'est souvent longtemps après l'avoir commise que l'on en découvre les méfaits. Il est alors difficile de reprendre une vieille preuve oubliée pour retrouver la commande perdue.

6.2 Automate de courtage

Pour enregistrer la séquence de commandes correctement exécutées par le système de preuve, l'outil de gestion de script s'interpose entre l'utilisateur et le système de preuve dans les deux sens. Cet outil est donc construit comme un automate à deux entrées et deux sorties. Les deux entrées correspondent aux commandes envoyées par l'utilisateur et aux réponses envoyées par le système de preuve. Les deux sorties correspondent aux commandes envoyées au système de preuve et aux réponses envoyées à l'utilisateur.

Dans le sens de l'utilisateur vers le système de preuve, l'automate interfère avec le dialogue: l'automate impose une temporisation entre les commandes pour assurer qu'une commande n'est pas envoyée vers le système de preuve tant que la réponse à la commande précédente n'est pas revenue. Il peut arriver qu'une commande envoyée par l'utilisateur ne soit jamais envoyée effectivement au système de preuve, par exemple lorsque l'utilisateur envoie un contre-ordre avant que le système de preuve ne soit disponible pour exécuter cette commande.

Dans le sens du système de preuve vers l'utilisateur, il serait naturel que l'automate laisse passer les données sans interférer, et qu'il se serve uniquement de l'information de passage. C'est approximativement ce qui se passe, mais nous verrons qu'il peut aussi être intéressant de ne pas retransmettre toutes les réponses à l'utilisateur.

L'automate doit être en mesure de répondre à n'importe quelle séquence de commandes venant de l'utilisateur, même si cette séquence est incohérente: a priori, l'utilisateur est humain et ce serait stupide de lui demander d'être cohérent. En revanche, il est raisonnable de supposer que le système de preuve répond à des contraintes simples. La première de ces contraintes est que ce système de preuve envoie exactement une réponse après réception de chaque commande.

Les évènements en provenance de l'utilisateur peuvent être de trois formes:

1. `requête.` Il s'agit des commandes usuelles.
2. `défaire.` Il s'agit de la commande qui permet d'annuler l'effet de la dernière commande usuelle.
3. `abandonner.` Il s'agit de la commande qui permet d'annuler toutes les commandes depuis la dernière commande de début de preuve si celle-ci n'a pas été conclue.

Les événements en provenance du système de preuve peuvent être de deux formes:

1. `bien reçu.` Il s'agit d'un accusé de réception pour une commande qui a été exécutée normalement.
2. `erreur.` Il s'agit d'un message indiquant que la dernière commande n'a pas été exécutée.

L'outil de gestion de script repose sur une hypothèse importante: lorsque le système de preuve réagit à une commande par un message d'erreur, on suppose que son état ne change pas. Ainsi, il doit être équivalent d'envoyer une commande provoquant une erreur et de ne pas l'envoyer. En revanche, nous ne supposons pas que toutes les commandes envoyées puissent être défaites. Simplement, l'outil de gestion de script doit prendre en compte la possibilité qu'une commande d'annulation provoque une erreur.

Comme nous ne supposons rien de l'utilisateur, nous devons faire face à l'hypothèse que ce dernier envoie de nombreuses commandes sans laisser le temps au système de preuve de les exécuter toutes. Pour cette raison, notre automate n'est pas un automate à états finis: il doit disposer de deux mémoires non bornées: l'une pour conserver toutes les commandes exécutées et l'autre pour conserver toutes les commandes en cours d'exécution. La première de ces mémoires s'utilise comme une pile: les opérations effectuées sont seulement d'ajouter de nouvelles commandes au cours de l'exécution ou d'en enlever lorsqu'elles sont défaites, mais toujours dans l'ordre opposé à celui où elles ont été introduites. Nous appellerons cette mémoire la *pile finale*. La deuxième de ces mémoires s'utilise comme une file d'attente. Cette mémoire reçoit toutes les commandes envoyées par l'utilisateur avant que le système de preuve n'ait le temps de les exécuter. Les commandes peuvent sortir de cette mémoire de deux manières différentes, soit parce qu'elles vont être exécutées, soit parce que l'utilisateur a requis qu'elles soient défaites. Dans le premier cas, les commandes sortent de la file d'attente dans l'ordre ou elles ont été introduites. Dans le deuxième cas, elles sortent dans l'ordre inverse de celui où elles ont été introduites. En plus de ces mémoires non bornées, l'automate utilisera également des mémoires de taille finie pour conserver la commande en cours d'exécution (le *tampon*), le nombre de requêtes d'annulation effectuées par l'utilisateur, et si une requête d'abandon a été envoyée par l'utilisateur.

Pour comprendre intuitivement le comportement de l'outil de gestion de script, étudions rapidement deux scénarios possibles d'exécution. Dans ces scénarios nous supposerons que l'utilisateur envoie les commandes `A`, `B`, `C` et `D` au système de preuve.

Supposons dans le premier scénario que l'utilisateur envoie la commande `A`, attend la réponse du système de preuve, puis les commandes `B`, `C` et `D` sans attendre de réponse, mais que la commande `C` provoque une erreur. Pour l'automate les évènements se succèdent de la manière suivante:

1. L'utilisateur envoie la requête `A`. Cette requête est directement stockée dans le tampon et envoyée au système de preuve.
2. Le système de preuve envoie un accusé de réception. La commande `A` est enlevée du tampon et passe dans la pile finale.
3. L'utilisateur envoie la requête `B`. Cette requête est directement stockée dans le tampon et envoyée au système de preuve.
4. L'utilisateur envoie les requêtes `C` et `D`. Ces requêtes sont conservées dans la file d'attente.
5. Le système de preuve envoie un accusé de réception. La commande `B` est enlevée du tampon et passe dans la pile finale. La commande `C` est enlevée de la file d'attente, mise dans le tampon et envoyée au système de preuve.
6. Le système de preuve envoie un message d'erreur. La commande `C` est enlevée du tampon et la commande `D` est enlevée de la file d'attente.

Notons ici que les messages d'erreurs ont pour effet de vider la file d'attente. En effet, il serait idiot de continuer obstinément à envoyer des commandes au système de preuve si toutes ces commandes se mettent à échouer.

Dans un deuxième scénario, l'utilisateur envoie les trois commandes `A`, `B` et `C` sans attendre la réponse du système de preuve. Puis il envoie des requêtes d'annulation pendant que le système de preuve execute la commande `B`. Dans ce cas, les évènements se succèdent de la manière suivante:

1. L'utilisateur envoie la requête `A`. Cette requête est directement stockée dans le tampon et envoyée au système de preuve.
2. L'utilisateur envoie les requêtes `B` et `C`. Ces requêtes sont conservées dans la file d'attente.
3. Le système de preuve envoie un accusé de réception. La commande `A` est enlevée du

tampon et passe dans la pile finale. La commande `B` est enlevée de la file d'attente, mise dans le tampon et envoyée au système de preuve.

4. L'utilisateur envoie une requête d'annulation. La commande `C` est enlevée de la file d'attente.
5. L'utilisateur envoie une requête d'annulation. Le nombre de requête d'annulation stocké dans l'automate est incrémenté.
6. Le système de preuve envoie un accusé de réception. La commande `B` est enlevée du tampon et passe dans la pile finale. Le nombre de requêtes d'annulation est décrémenté et une commande d'annulation est envoyée au système de preuve. L'automate passe dans un état indiquant que la commande en cours d'exécution est une commande d'annulation.

Dans [21], nous donnons une description précise du comportement de l'automate de courtage. Cette description est assez précise pour fournir des critères formels de correction. Ces critères formels expriment pratiquement que l'automate de courtage doit sauvegarder l'ensemble des commandes exécutées sans erreurs dans la pile finale. L'énoncé de ces critères est légèrement compliqué par le comportement en cas d'erreur, où l'on veut que les commandes en attente soient simplement annulées.

6.3 Services pour l'interaction

Une vision alternative du comportement de l'outil de gestion de script est que cet outil gère plusieurs zones à l'intérieur de la fenêtre qui contient les commandes envoyées à l'utilisateur. Ces zones peuvent être repérées par la couleur de fond et déplacer une commande d'une mémoire à l'autre s'effectue alors simplement en changeant la couleur de fond. De cette manière, l'utilisateur dispose d'un retour d'information important: du premier coup d'œil il peut comprendre l'état d'avancement de son travail en regardant les commandes qui sont dans la couleur de fond correspondant à la pile finale. Il est néanmoins important d'interdire les opérations d'édition sur le contenu de cette pile finale: ce n'est qu'à cette condition que l'automate de courtage peut garantir que la pile finale représente effectivement la séquence de commandes exécutées sans erreur par le système de preuve.

Puisque l'historique des commandes est affiché dans une fenêtre visible de l'utilisateur, nous lui fournissons également la possibilité de revenir en arrière simplement en sélectionnant la commande qu'il veut défaire. L'automate se charge alors d'envoyer la bonne séquence d'annulation pour revenir à la position désignée.

Une utilisation fréquente du système de gestion de script est de rejouer rapidement des fichiers de commandes qui peuvent contenir des erreurs. Dans ce cas, l'outil de gestion de script met directement l'ensemble des commandes dans la file d'attente et les exécute l'une après l'autre. En cas d'erreur, la file d'attente est vidée, la sélection courante est placée sur la commande fautive et l'utilisateur peut la voir au premier coup d'œil.

Dans le cas où la file d'attente contient de nombreuses commandes il est inutile d'afficher le résultat de toutes les commandes envoyées. L'automate remplit donc cette fonction supplémentaire de filtrer le résultat des commandes. Lorsque la file d'attente n'est pas vide, le résultat des commandes est pris en compte pour changer le statut des commandes exécutées, mais il n'est pas retransmis vers le dispositif d'affichage. Il faut alors traiter avec soin les cas d'erreur. En effet, lorsqu'une erreur se produit, il faut re-demander au système de preuve le résultat de la dernière commande pour que l'utilisateur puisse le voir.

6.4 Conclusion et extensions

Le modèle d'automate présenté dans cette section suppose que l'on attend bien sagement que le système de preuve réponde aux commandes envoyées. Il est raisonnable de s'interroger sur la possibilité d'interrompre un calcul trop long ou éventuellement infini. Pour mettre en œuvre une telle fonctionalité, il faut fournir à l'utilisateur un moyen d'interaction, par exemple un bouton ou une option dans un menu pour qu'il stoppe le calcul en cours. L'effet de cette interaction se traduira par l'envoi d'un signal au système de preuve. Pour assurer que le dialogue entre le système de preuve et l'interface homme-machine reste dans un état cohérent il faut mettre en place les procédures d'interruptions telles que:

1. un signal reçu pendant que le système est en attente d'une commande est ignoré,
2. un signal reçu pendant que le système analyse une requête, mais avant qu'il commence à émettre la réponse doit interrompre l'exécution de cette requête, l'annuler, et émettre un message d'erreur,
3. un signal reçu pendant que le système envoie la réponse d'une commande précédente doit être ignoré.

Ainsi les interruptions requises par l'utilisateur doivent simplement se traduire par un message d'erreur en retour, mais seulement lorsque le système n'est ni oisif ni en train d'émettre. Le système de gestion de script n'a alors pas besoin de modifications pour traiter ce type d'interaction.

Le système de gestion de script présenté dans cette section repose sur un modèle très simple du système de preuve, qui suppose que chaque commande dépend effectivement de l'ensemble des commandes qui la précèdent. En fait, une analyse plus fine de la sémantique des commandes et des objets manipulés permet de comprendre que les dépendances entre les différentes commandes sont plus lâches que la dépendance chronologique. Des outils basés sur ces analyses de dépendance pourront aider l'utilisateur à gérer son script de façon encore plus efficace que ce que permet l'automate présenté dans cette section. L'étude de ces dépendances et des outils associés fait l'objet d'une thèse effectuée dans le projet CROAP par O. Pons [59], que j'encadre en collaboration avec L. Rideau.

Chapitre 7

Preuves sur la sémantique

7.1 Introduction

Le travail décrit dans ce chapitre a été effectué en collaboration avec Ranan Fraer au cours d'un stage de DEA et au début de sa thèse [18]. C'est un exemple de description complète d'un langage de programmation qui met en œuvre la majeure partie des outils développés dans le projet CROAP pour la sémantique des langages de programmation pour l'exécution des spécifications sémantiques et pour l'utilisation dans des preuves. La motivation initiale est de montrer que le formalisme TYPOL est un bon candidat pour la description formelle des langages de programmation car il est adapté à ces deux catégories d'utilisation de spécifications sémantiques.

Le langage étudié dans cette expérience correspond au noyau impératif minimal des langages de programmation usuels. Il contient des affectations, des séquences, des boucles `while` et ne manipule que des données primitives: booléens et entiers. Cette expérience n'est pas tout à fait satisfaisant, car ce langage n'est pas encore de taille comparable à un véritable langage de programmation. L'une des raisons pour cette petite taille est que l'expérience était aussi conçue pour servir de base à un cours sur la sémantique des langages et la preuve [15] et il était nécessaire de se cantonner à un exemple que l'on puisse survoler en quelques heures. Néanmoins, ce travail a permis de mettre à l'œuvre pour la première fois une palette large d'outils: exécution en Prolog de spécification TYPOL grâce au compilateur de T. Despeyroux, traduction en Coq à l'aide des outils développés par D. Terrasse, et développement de preuves en Coq à l'aide de l'environnement CtCoq. En particulier, la réalisation de cette preuve a déjà permis de découvrir de nombreuses erreurs dans la réalisation des outils mis en œuvre.

Cette expérience a fourni des enseignements sur la réalisation de telles preuves autour de spécifications sémantiques. En particulier, nous avons isolé des styles de preuves spécifiques pour lesquels nous avons mis au point de nouvelles procédures de preuve.

7.2 Traduction de la sémantique naturelle vers Coq

Le travail décrit dans cette section est celui de Delphine Terrasse pour sa thèse de doctorat [65]. Il est mentionné ici car les expériences que nous avons réalisées sont fortement influencées par les choix effectués dans ce travail de thèse.

7.2.1 Choix d'implémentation des phyla

Avant de traduire une spécification sémantique il est nécessaire de traduire les données sur lesquelles cette sémantique opérera, c'est à dire le langage. La méthode employée est de représenter chacun des opérateurs du langage par un constructeur d'un type récursif. Un cas particulier doit être pris en compte pour les opérateurs de listes, qui seront représentés par une paire d'opérateurs: l'un est binaire et ajoute un élément en tête d'une liste, l'autre a un ou zéro éléments suivant que la liste doit avoir au moins un élément ou non.

Pour décrire précisément les opérateurs du langage il est également nécessaire de décrire précisément les catégories syntaxiques (ces catégories sont appelées *phyla* dans la terminologie de la sémantique naturelle). Ainsi, il faut pouvoir exprimer qu'une instruction conditionnelle prend une expression en premier argument et deux instructions en deuxième et troisième arguments. Pour représenter ces phyla, deux possibilités sont offertes. La première est de représenter chaque phylum comme un type récursif différent. Le principal problème est que les phyla et les types n'ont pas les mêmes contraintes: un opérateur peut appartenir à deux phyla différents, ce qui n'est pas possible pour les constructeurs de types récursifs. Pour pallier cet inconvénient il est nécessaire d'ajouter des constructeurs artificiels, appelés *opérateurs de coercion*, qui indiquent que des arbres d'un phylum donné sont aussi acceptés dans un autre phylum. Cette construction n'est possible que si la syntaxe abstraite est bien formée: si deux phyla ont des opérateurs en commun, alors tout opérateur commun appartient à un phylum inclus dans les deux. La deuxième solution pour représenter les phyla est de fournir une collection de propriétés caractéristiques qui décrivent les arbres présents dans chaque phylum.

L'avantage de la première solution est que le raisonnement phylum par phylum est plus aisé. De plus, la vérification qu'un arbre appartient bien à un phylum se fait simplement par typage et par filtrage en faisant apparaître les opérateurs de coercion. En revanche, cette solution est moins générale car elle ne s'applique que pour certaines syntaxes abstraites et elle est moins robuste car l'ajout d'une nouvelle catégorie syntaxique peut provoquer un changement dans la structure des catégories existantes. Il est alors plus difficile de réutiliser des preuves entre deux langages qui ne diffèrent que par l'ajout d'un opérateur et d'une catégorie syntaxique.

L'avantage de la deuxième solution est qu'elle est plus générale et plus robuste aux changements de langages. En revanche, elle est moins efficace, puisque la vérification de l'appartenance d'un arbre à un phylum devient maintenant une opération de preuve, alors que c'était avant uniquement une opération de typage.

Dans son travail, Delphine Terrasse a implémenté successivement les deux solutions et nous avons expérimenté avec les deux. Les buts d'appartenance d'une expression à un phylum apparaissent alors régulièrement, mais leur preuve peut généralement se faire systématiquement. Partant de cette expérience, nous avons élaboré des instructions de composition de tactiques qui permettent d'appliquer des tactiques particulières automatiquement sur les buts d'appartenance, en les traitant de manière analogue aux *type-checking conditions* de PVS [60] ou Nuprl [29]. Nous décrivons plus loin les instructions mises au point.

7.2.2 Spécifications sémantiques et définitions inductives

La traduction vers Coq des spécifications sémantiques repose également sur les définition inductives, mais il s'agit cette fois-ci de propriétés inductives. Schématiquement, la traduction va maintenant reposer sur une définition inductive par jugement. Chaque règle d'inférence est traduite en un constructeur pour cette définition inductive. Chaque règle d'inférence est encapsulée dans une série de quantifications universelles qui introduisent les variables de la règle d'inférence.

L'intérêt d'associer une définition inductive à chaque jugement est que le système Coq va fournir automatiquement des théorèmes d'induction. En effet, la définition inductive exprime intuitivement que la propriété définie est la plus petite définition au sens ensembliste qui vérifie les propriétés énoncées par les constructeurs.

Ces théorèmes d'induction permettent exactement d'effectuer les raisonnements par récurrence sur la taille des preuves, où l'on considère que les règles d'inférences sont les seuls moyens de construire des preuves de la propriété qu'elles définissent. Ainsi, les preuves par récurrence sur la taille des preuves, comme celles apparaissant dans l'article [31] et celles appelées *rule induction* dans [70], se feront simplement à l'aide de la tactique `Elim` de Coq.

7.2.3 Outils d'inversion

La tactique `Elim` utilisée pour faire les raisonnements par récurrence fonctionne en utilisant la forme particulière des théorèmes de récurrence. Pour une propriété inductive `(P x y)` le système de preuve construit systématiquement un théorème de la forme suivante:

$$\forall\ \mathtt{Q}\colon \mathtt{A} \rightarrow \mathtt{B} \rightarrow \mathtt{Prop}.\ \ldots\forall\ \mathtt{x},\ \mathtt{y}.\ (\mathtt{P}\ \mathtt{x}\ \mathtt{y}) \rightarrow (\mathtt{Q}\ \mathtt{x}\ \mathtt{y})$$

Si `H` est une hypothèse de la forme `(P a b)` la tactique `Elim` décompose le but en l'application d'une propriété aux arguments `a` et `b`. En particulier, la propriété `Q` est constante si `a` et `b` n'apparaissent pas dans le but. De plus, cette procédure ne donne pas de résultats satisfaisants si `a` et `b` sont eux-mêmes des termes complexes. Dans ces cas, il faut exprimer que seules certaines règles peuvent s'appliquer.

Christina Cornes et Delphine Terrasse se sont intéressées à ce problème et ont construit une tactique complexe pour gérer systématiquement ce type de raisonnement [30]. Du point de vue de l'organisation des preuves, cette tactique a un comportement difficile à maitriser: elle engendre un nombre important d'égalités qu'elle introduit comme nouvelles hypothèses. L'utilisateur a alors très peu de contrôle sur le nom des hypothèses et sur leur nombre dans chaque cas. Ce comportement rend le traitement systématique des preuves complexe et a également servi de motivation pour mettre au point des instructions de composition de tactiques.

7.3 Un exemple non trivial: la propagation de constantes

La propagation de constante est une tranformation de programme utilisée fréquemment dans l'optimisation des compilateurs. Cette transformation présente une complexité non trivial, car elle effectue une transformation globale du programme. Dans [18], nous avons également étudié des transformation locales dont la preuve est beaucoup plus simple.

7.3.1 Description intuitive

L'exemple choisi est celui d'un outil de transformation de programmes. Ce travail permet d'aborder les mêmes problèmes que dans une compilation, tout en diminuant l'importance des données préliminaires car on n'utilise alors qu'un seul langage de programmation.

La tâche effectuée par l'outil de transformation de programme considéré est de reconnaître partiellement les variables dont la valeur est fixée en certains endroits du programme, quelles que soient les valeurs initiales et d'effectuer alors en avance les calculs qui peuvent être déduits de ces informations. Prenons par exemple le fragment de programme suivant:

```
x := 3;
y := x + 4;
x := z + y;
```

Quelles que soient les valeurs de `x`, `y` et `z` avant l'exécution de ce fragment de programme, on sait néanmoins que `y` vaudra 7 et `x` vaudra `z + 7` après cette exécution. Ce fragment peut donc être remplaçé par le fragment suivant:

```
x := z + 7;
y := 7;
```

7.3.2 Environnements de variables connues

Pour représenter les valeurs connues des variables, l'outil de propagation de constantes utilise un environnement qui est mis à jour au cours de l'analyse de chaque instruction. Si l'instruction considérée est une affectation d'une variable, alors c'est l'information concernant cette variable dans l'environnement qui sera mise à jour.

1. On enlève toute information sur la variable en partie gauche si l'expression en partie droite ne peut pas être déterminée statiquement.
2. On ajoute ou on remplace l'information existante pour la variable en partie gauche si l'expression en partie droite peut être déterminée statiquement.

Pour une instruction conditionnelle, il y a déjà le cas simple où l'expression testée peut être déterminée statiquement: dans ce cas l'instruction conditionnelle se réduit à la branche correspondante et la propagation de constante continue seulement sur cette branche. Si l'expression testée ne peut pas être déterminée statiquement, il faut effectuer séparément la propagation des constantes dans les deux branches. Ensuite on compare les deux environnements obtenus dans chaque branche. Toutes les variables qui apparaissent dans les deux environnements avec la même valeur sont conservées dans l'environnement résultant. Les variables qui n'apparaissent que d'un coté doivent être ajoutées à la fin de la branche sous forme d'affectation. Prenons par exemple le fragment de programme suivant:

```
x := 3;
if a = true then
  begin
    y := 4;
    z := x + 3;
    t := z * t;
  end
else
  begin
    y := 4;
    z := x + 4;
    t := 3 * t;
  end
```

En l'absence d'informations supplémentaires, l'outil de propagation de constantes va traiter les deux branches séparément. A la fin de la première branche, les variables `x`, `y` et `z` sont connues statiquement, avec les valeurs respectives 3, 4 et 6. L'affectation `t := z * t` doit être laissée sous la forme `t := 6 * t`. A la fin de la deuxième branche, les variables `x`, `y` et `z` sont connues également, mais avec les valeurs respectives 3, 4 et 7. Ainsi, les valeurs de `x` et `y` sont les mêmes en fin de calcul, quelle que soit la branche exécutée. En revanche, la variable `z` reçoit une valeur connue statiquement pour chaque branche, mais différente d'une branche à l'autre, tandis que la variable `t` reçoit une valeur non connue statiquement. En se basant sur ces informations, l'outil de propagation remplacera l'instruction donnée en exemple par l'instruction suivante:

```
if a = true then
  begin
    t := 6 * t;
    z := 6;
  end
else
  begin
    t := 3 * t;
    z := 7;
  end;
```

```
x := 3;
y := 4;
```

Plutôt que d'introduire une nouvelle structure de données, l'algorithme développé implémente l'environnement comme une séquence d'affectations où toutes les expressions en partie droite sont des valeurs immédiates (dans notre cas, des entiers ou des booléens) et où les variables en partie gauche doivent être deux à deux distinctes.

La transformation de programme peut ainsi être représentée par un jugement de la forme suivante:

$$\rho \vdash I \rightarrow I' + \rho'$$

Ce jugement se lit de la façon suivante: *la propagation de constantes sur l'instruction I lorsque les constantes connues initialement sont données par ρ donne l'instruction I' et les constantes décrites dans ρ'.* Ce jugement est entièrement défini à l'aide de 77 règles d'inférences.

7.3.3 Preuves

Pour raisonner sur la validité de la transformation, il est nécessaire de disposer également de la description formelle de la sémantique du langage[1]. De même que pour l'algorithme de propagation, la sémantique du langage de programmation s'exprime à l'aide d'un environnement pour représenter la valeur des variables à chaque étape. En fait, la sémantique s'exprime à l'aide d'une relation ternaire qui indique qu'une instruction transforme un environnement en entrée en un environnement en sortie. Nous utiliserons un jugement de la forme suivante:

$$\sigma \vdash I \rightarrow \sigma'$$

Ce jugement peut être lu de la façon suivante: *l'exécution de l'instruction I à partir de l'environnement σ termine et retourne l'environnement σ'.* Ce jugement est entièrement défini à l'aide de 27 règles d'inférences.

Pour prouver la validité de la transformation de programmes, on peut chercher à prouver les propriétés suivantes:

1. La spécification de la transformation de programmes est déterministe pour les deux premiers arguments, pour σ et I fixés, il existe un seul couple I',σ' tel que la relation suivante soit prouvable:

   $$\sigma \vdash I \rightarrow I' + \sigma'$$

   En corollaire, la description formelle de la propagation de constantes décrit une fonction.
2. L'algorithme de transformation termine toujours,
3. La propagation de constantes préserve la sémantique des programmes transformés. Intuitivement, on veut exprimer que les programmes transformés font les mêmes calculs que les programmes de départ. Ceci s'exprime en indiquant que la valeur

1. 16 règles pour eval.ty, 2 règles pour update.ty, 9 règles pour dynamics.ty

finale des variables est la même pour le programme original et pour le programme final. Ici, il faut traiter avec soin le cas des variables qui sont constantes en début et en fin de calcul. L'énoncé le plus général doit prendre la forme suivante:

$$\begin{array}{l} \forall \rho,\rho',I,I',\sigma_1,\sigma_2 \\ \quad \sigma_1 \vdash \rho; I \rightarrow \sigma_2 \quad \& \\ \rho \vdash I \rightarrow I' + \rho' \quad \Rightarrow \\ \qquad\qquad\qquad \sigma_1 \vdash I'; \rho' \rightarrow \sigma_2 \end{array}$$

La première hypothèse exprime que l'on considère uniquement l'exécution de I dans un environnement où les variables ont initialement des valeurs compatibles avec celles exprimées dans ρ. Pour cela, il suffit de commencer par exécuter cet environnement ρ, puisqu'il est implémenté comme une séquence d'affectations. Cette exécution va alors retourner un environnement σ_2. La seconde prémisse indique que l'on transforme le programme en un nouveau programme I' accompagné d'une nouvelle liste de constantes ρ'. La conclusion est que l'execution du nouveau programme suivie de l'exécution de l'environnement ρ' retourne le même environnement σ_2.

Si l'on se concentre sur la preuve de la dernière propriété, celle-ci a été entièrement vérifiée mécaniquement en Coq en la décomposant en 47 lemmes. Ces lemmes peuvent être triés en six catégories:

1. La première catégorie concerne les manipulations de l'environnement utilisé pour la sémantique dynamique: montrer que l'on peut permuter les mises à jour pour deux variables distinctes, montrer que la recherche d'une valeur après une mise à jour retourne la même valeur, etc.
2. La deuxième catégorie concerne la forme des environnements utilisés pour la propagation de constantes. En effet, nous avons implémentés ces environnements à l'aide de séquences d'affectation. Toutes les séquences ne sont pas acceptables: les bonnes séquences doivent contenir uniquement des affectations à des variables distinctes deux à deux et les valeurs affectées doivent toujours être des constantes. Les lemmes de cette catégorie assurent que les environnements intermédiaires utilisés en cours de traitement sont bien formés dès que l'environnement initial l'est.
3. La troisième catégorie concerne le comportement de l'algorithme de propagation pour les variables qui ne sont pas constantes.
4. La quatrième catégorie de lemmes montre comment les environnements utilisés pour la propagation des constantes interagissent avec les environnements utilisés pour la sémantique dynamique. En effet, les environnements utilisés pour la propagation de constantes sont des séquences d'instructions que l'on peut exécuter pour agir sur les environnements de la sémantique dynamique.
5. La cinquième collection de lemmes décrit les opérations de base sur les environnements de propagation des constantes.
6. La sixième collection de lemmes compare les opérations principales de la propagation des constantes avec les opérations principales de la sémantique dynamique: la mise à jour des environnements dans les deux cadres, la propagation des

constantes dans les expressions avec l'évaluation des expressions et la propagation des constantes dans les instructions avec l'exécution des instructions.

La preuve complète comprend légèrement plus de 1000 lignes de commandes pour le système Coq.

7.4 Des tactiques spécialisées

Les preuves effectuées pour raisonner sur la sémantique naturelle ont une structure différente des preuves effectuées dans d'autres domaines dans Coq. Nos définitions inductives ont typiquement une taille plus importante: le langage de programmation jouet utilisé pour notre étude comporte déjà 20 opérateurs significatifs, le phylum des instructions en contient 5 et le phylum des expressions en contient 12. Par nature, il faut s'attendre à ce qu'un vrai langage de programmation contienne 10 fois plus d'opérateurs. Si l'on observe les descriptions sémantiques, celles que nous avons manipulées contiennent 27 règles pour la sémantique dynamique et 77 règles pour la propagation des constantes. Le jugement principal pour l'exécution des instructions est défini par 7 règles d'inférence et le jugement principal pour la propagation des constantes sur les instructions est défini par 10 règles d'inférence. A titre de comparaison, la spécification formelle du langage Java contient de l'ordre de 400 règles d'inférence.

Ces chiffres sont importants à garder en mémoire: usuellement, les preuves de sémantique se font "par cas" sur les opérateurs du langage ou "par cas" sur les règles d'inférence qui définissent un jugement. On aura donc naturellement des preuves par cas faisant apparaitre un grand nombre de cas (une demi-douzaine pour notre langage, plusieurs dizaines pour les preuves de déterminisme), même pour des lemmes relativement simples. Ces remarques nous ont conduits à mettre au point des tactiques spécialisées qui permettent de traiter de nombreux cas similaires en une seule commande de preuve.

7.4.1 Primitives de composition importantes

Pour traiter plusieurs preuves en une seule commande nous nous reposons sur deux primitives de composition de tactiques importantes et fournies par Coq en standard.

La première de ces primitives permet de traiter tous les sous-buts produits par une première tactique avec une même seconde tactique. La syntaxe de cette primitive de composition est la suivante[2]:

$$\mathtt{t}_1\mathtt{;t}_2$$

Etant donné un but g la commande $\mathtt{t}_1\mathtt{;t}_2$ appliquera d'abord $\mathtt{t}_1$ à g, ce qui produira un certain nombre de buts $\mathrm{g}_1, \ldots, \mathrm{g}_n$. Ensuite la tactique $\mathtt{t}_2$ sera appliquée successivement à chacun de ces buts. Si la tactique $\mathtt{t}_1$ ne produit aucun but, la tactique $\mathtt{t}_2$ est tout simplement ignorée.

Si l'on réfléchit à la maintenabilité des preuves, cette primitive de composition est très intéressante. En effet, si l'on modifie la spécification du problème et si le nombre de buts engendrés par la tactique $\mathtt{t}_1$ change, il sera souvent possible de continuer à faire la

2. Dans la suite de l'article nous nommerons également cette tactique la tactique `Then`.

preuve en utilisant la même composition de tactiques. En sémantique des langages, ce cas de figure apparait dès que l'on essaie de modifier le langage: si l'on ajoute un opérateur au langage et que la tactique $\mathtt{t}_1$ effectue une preuve par cas sur les opérateurs du langage, on trouvera alors un cas de plus. La tactique $\mathtt{t}_2$ fonctionnera sans changement sur ce cas si le nouvel opérateur est similaire à un opérateur déjà existant.

La deuxième primitive de composition importante pour notre travail est la primitive `Orelse`. Cette primitive permet d'indiquer que l'on essaie successivement deux tactiques de preuve sur un but.

Notons que ces tactiques sont fournies en standard par de nombreux systèmes de preuve, mais pas nécessairement par tous. Par exemple, la primitive `Then` n'est fournie dans le système Lego d'Edimbourg que depuis une version récente. De même, cette tactique est d'une utilisation délicate dans Isabelle. La raison de cette difficulté provient de l'utilisation de variables existentielles, qui peuvent être instantiées en une opération sur plusieurs buts. Retenons que le comportement de cette primitive est simple si l'on peut considérer la résolution des différents buts engendrés par une tactique indépendamment, mais qu'elle devient beaucoup plus difficile à maîtriser lorsque la preuve d'un but peut avoir des effets de bords sur ses voisins.

7.4.2 Filtrage de buts

La méthode que nous utilisons pour traiter efficacement les preuves comportant un grand nombre de cas est de regrouper ces cas par classes et de résoudre chaque classe à l'aide d'une commande particulière. Ainsi, nous construisons des commandes de la forme suivante, où `t` est la commande qui engendre tous les cas, $\mathtt{t}_1$ est la commande qui résout les buts appartenant à la première classe, $\mathtt{t}_2$ est la commande qui résout les buts appartenant à la deuxième classe, et ainsi de suite.

$$\mathtt{t;(t_1\ Orelse\ t_2\ Orelse\ ...)}$$

Chacune des commandes $\mathtt{t}_1$, $\mathtt{t}_2$, et ainsi de suite peut elle-même être une suite de commandes composées grâce à la tactique `Then`.

L'inconvénient de cette organisation apparait lorsque la tactique $\mathtt{t}_1$ modifie les cas qui appartiennent à la deuxième catégorie sans les résoudre, mais sans échouer. Dans ce cas, la tactique $\mathtt{t}_2$ souffre en quelque sorte de la pollution laissée par celle qui la précède. La première méthode pour éviter ce type de pollution est d'assurer que chacune des tactique $\mathtt{t}_i$ échoue si elle ne réussit pas à prouver complètement le but qu'elle traite. Il suffit pour cela de remplacer la tactique $\mathtt{t}_i$ par un tactique $\mathtt{t}'_i$ définie de la façon suivante, où `Fail` est une tactique qui échoue toujours:

$$\mathrm{t}'_i = (\mathrm{t}_i; \mathrm{Fail}).$$

En effet, si la tactique $\mathtt{t}_i$ résout complètement le but, elle ne produit pas de sous-buts et `Fail` est simplement ignorée.

La deuxième méthode pour éviter la pollution est de filtrer à priori les buts en observant leur forme. Dans des cas assez fréquents, ceci peut être fait en appliquant une tactique qui ne change pas le but, mais échoue s'il n'a pas la forme adéquate. Si l'on dispose de telles tactiques $\mathtt{t}_{f1}$, $\mathtt{t}_{f2}$, et ainsi de suite pour chaque catégorie, il suffit alors de remplacer chaque tactique $\mathtt{t}_i$ par la tactique $\mathtt{t}'_i$ définie de la façon suivante:

$$\mathtt{t}'_i \texttt{ = } \mathtt{t}_{fi}\texttt{; } \mathtt{t}_i\texttt{.}$$

Il est en général assez difficile de trouver des tactiques de filtrage adéquates. Néanmoins, le problème est plus simple lorsque les classes de buts à discriminer sont de la forme suivante, ou le nom P_i diffère d'une classe de buts à l'autre:

$$(P_i f_1 \dots f_n)$$

Si P_i est un prédicat de type $\tau_1 \to \cdots \tau_n \to Prop$, alors la tactique suivante permet de discriminer les buts de la classe i:

$$\text{Apply } \lambda \mathrm{x}_1 : \tau_1 \dots \lambda \mathrm{x}_n : \tau_n .\ \lambda \mathrm{h} : (\mathrm{P}_i\ \mathrm{x}_1\ \dots\ \mathrm{x}_n).\mathrm{h}$$

7.5 Primitives d'essai sans engagement

La contrainte importante dans la construction de tactiques de filtrage est qu'une telle tactique devrait s'appliquer avec succès sur un but sans le modifier. Nous avons exhibé une telle tactique dans certains cas, mais c'est en général difficile d'en trouver. Il serait plus aisé de disposer d'une primitive qui permet d'appliquer une tactique, tester si elle échoue ou réussit, oublier les modifications qu'elle a effectuées, et choisir une branche d'exécution définie grâce à ce test. Nous avons développé une telle tactique dans Coq. La syntaxe de cette primitive est la suivante:

$$\texttt{Success } \mathtt{t}_1 \texttt{ Then } \mathtt{t}_2 \texttt{ else } \mathtt{t}_3$$

Cette primitive essaie $\mathtt{t}_1$ sur le but considéré. Si cette tactique échoue alors elle exécute $\mathtt{t}_3$ sur le but considéré. Sinon, l'effet de la tactique $\mathtt{t}_1$ est annulé et la tactique $\mathtt{t}_2$ est exécutée.

7.5.1 Renommage local d'hypothèses

Dans les preuves par cas, le nombre et la forme des hypothèses introduites dans le contexte local peut également varier d'un cas à l'autre. Néanmoins, il peut être intéressant de retrouver des hypothèses d'une forme particulière lorsque cette hypothèse joue systématiquement un rôle clef dans la résolution des buts d'une classe donnée. Un exemple d'hypothèse particulière est l'hypothèse de récurrence.

Il n'existe que deux commandes dans le système Coq qui introduisent des hypothèses dans le contexte local. La commande `Intro` laisse l'utilisateur contrôler exactement le nom qui sera donné à une hypothèse introduite. En revanche, la commande `Inversion`, qui joue un rôle prépondérant dans les preuves par cas sur la forme des dérivations de sémantique naturelle, ne laisse pas ce contrôle. Pour pallier ce défaut, il est intéressant de disposer d'une primitive qui permette de retrouver des hypothèses par filtrage, comme nous l'avons fait pour les buts.

La première primitive que nous avons conçue s'inspire de la primitive `Success` décrite à la section précédente. Sa syntaxe est la suivante:

$$\texttt{Successhyp X with } \mathtt{t}_1 \texttt{ Then } \mathtt{t}_2 \texttt{ Else } \mathtt{t}_3 \texttt{ End.}$$

Cette commande essaie successivement la tactique t_1 en remplaçant le nom X par les noms d'hypothèses présents dans le but considérés et en commençant par le dernier introduit. Si cette tactique réussit pour un nom H, l'effet de la tactique est annulé comme pour la primitive Success et la commande t_2 est exécutée, également en remplaçant le nom X par le nom H. Si aucune hypothèse ne satisfait la tactique t_1, la tactique t_3 est exécutée.

7.5.2 Filtrage d'hypothèses par un motif

De même qu'il est aisé de construire une tactique de filtrage pour reconnaître un but de la forme $(Pf_1 \ldots f_n)$, il est aisé de reconnaître une hypothèse de cette forme. En supposant que P a le type $\tau_1 \to \cdots \tau_n \to Prop$, la commande suivante permet de retrouver une hypothèse de cette forme:

Specialize $\lambda x_1 : \tau_1 \ldots \lambda x_n : \tau_n\ \lambda h : (P\ x_1\ \ldots\ x_n).h$ with 1 := H

En pratique, la majeure partie des recherches d'hypothèses que nous sommes amenés à faire utilise ce type de recherche par prédicat. Ceci nous a amené à définir une forme plus concise pour cette forme de recherche:

Matchhyp X with $\lambda x_1 : \tau_1\ \ldots \lambda x_n : \tau_n.\ (P\ x_1\ \ldots\ x_n)$ Then t End

Cette commande est équivalente à la commande suivante:

Successhyp x with
Specialize $\lambda x_1 : \tau_1\ \ldots \lambda x_n : \tau_n.\ \lambda h : (P\ x_1\ \ldots\ x_n).h$ with 1 := X
Then t Else Fail End

7.6 Conclusions

L'expérience décrite dans ce chapitre a permis d'explorer l'ensemble des techniques et méthodes utilisées pour décrire formellement des propriétés de langages de programmation et les prouver. Il s'agit donc d'un témoin de faisabilité intéressant, mais qui ne représente pas une surprise importante, dans la mesure où des preuves similaires avaient déjà été effectuées avec d'autres systèmes de preuves ou sur le papier.

En pratique, cette expérience ne permet pas réellement de conclure à la faisabilité pratique de ce type de preuve en Coq pour des langages réalistes. Le nombre de règles apparaissant dans nos spécification dynamiques est réduit à moins d'une trentaine, alors que dans la description de vrais langages [3, 8] le nombre de règles peut dépasser la centaine. En particulier, certaines preuves, comme les preuves de déterminisme, ont un caractère quadratique dans le nombre de règles d'inférence des spécifications sémantiques. Néanmoins, les travaux que nous avons menés sur des tactiques spécialisées montrent qu'il est possible de réduire le nombre de cas qui doivent être traités séparément par l'utilisateur.

La sémantique naturelle est avant tout une utilisation de la programmation logique pour définir la sémantique de langages de programmation. En ce sens, notre travail sur la preuve de sémantique est simplement un travail sur la preuve de programmes logiques et peut être vu comme un complément intéressant des travaux en théorie des types sur la programmation fonctionnelle. Il serait également intéressant de raisonner directement sur des outils de manipulation de programmes qui seraient simplement décrits comme des programmes fonctionnels purs en Coq. Nous avons commencé un travail dans ce sens en nous intéressant cette fois-ci à un compilateur. Dans ce travail, nous devons faire cohabiter un style "programmation logique" pour la sémantique dynamique du langage et un style "programmation fonctionnelle" pour le compilateur. Le style programmation logique est imposé par la nature du problème, puisque la sémantique dynamique d'un langage complet au sens de Turing ne peut être décrite par une fonction de Coq, à cause de l'indécidabilité du problème de l'arrêt.

Avec les travaux en cours dans l'équipe Coq, nous pouvons ainsi espérer disposer à terme d'un environnement de déduction qui permette de raisonner à la fois sur des programmes impératifs, des programmes fonctionnels et des programmes logiques.

Chapitre 8

Vérification mécanique d'un compilateur

La démonstration présentée dans la section précédente suit une option "tout programmation en logique". La sémantique dynamique, la sémantique statique et la description de l'outil de propagation des constantes sont toutes données par des spécifications en Typol puis traduites en Coq avant d'effectuer la démonstration. Le programme que l'on vérifie est donc un programme logique, exécutable grâce à un interprète ou un compilateur Prolog. Une approche alternative est de décrire le programme à vérifier directement en Coq comme une expression fonctionnelle pure. On pourra alors exécuter ce programme en se reposant sur les primitives d'extraction de Coq pour obtenir un programme ML que l'on pourra compiler et exécuter. C'est cette approche que nous avons voulu expérimenter dans la démonstration de correction d'un compilateur pour le même langage que celui utilisé au chapitre précédent. Une motivation supplémentaire de ce travail est de compléter le panorama de l'étude d'un langage de programmation entamé dans [15, 18]. Le travail décrit ici fait également l'objet d'un article [16].

Le sujet des démonstrations de correction de compilateur est assez ancien, puisque McCarthy et Painter [53] ont publié dès 1972 un article qui décrit la correction d'un compilateur pour des expressions arithmétique, mais cette démonstration n'a pas été vérifiée sur ordinateur. Il faut attendre les travaux de Polak en 1980 [58] pour trouver un compilateur spécifié et vérifié sur machine, à l'aide du vérificateur *Stanford Pascal Verifier* [61]. Dans cette expérience, le langage compilé est proche de Pascal et est décrit en utilisant la sémantique dénotationelle [63]. Néanmoins, une partie des preuves est effectuée à la main. Ensuite Moore [54] et Young [71] ont effectué la vérification mécanique d'un compilateur pour un assembleur, `Piton`, et pour un langage de style algol, *Gypsy*, respectivement. Le langage Piton a été utilisé comme langage cible pour le langage *Gypsy*. Ces démonstrations ont été effectuées ou vérifiées à l'aide du système de preuve Nqthm [24]. Ce système est conçu principalement comme un système de preuve pour Lisp, utilisant principalement la réécriture comme technique de calcul. Les preuves par récurrence y sont guidées en fournissant des indices (*hints* en anglais). Young indique par exemple que l'un des indices fournis est une fonction à 12 paramètres et faisant 250 lignes. Par comparaison, notre travail est beaucoup plus modeste, mais nous espérons que les théorèmes et méthodes de démonstrations que nous proposerons permettent d'éviter ce genre de "sorcellerie informatique". Il est également important de citer le travail de Gutman, Wand, et leur équipe [41], qui ont fourni la démonstration complète de correction d'un compilateur pour le langage `scheme`, à partir d'une description en sémantique dénotationelle de ce langage. Cette démonstration est très complète et s'intéresse aussi aux problèmes de l'optimisation (surtout pour les récursions terminales) et de la gestion de mémoire. Notons toutefois que cette vérification n'a pas été réalisée sur machine. Enfin, et plus près de nous il faut aussi noter le travail de démonstration de correction de la traduction du langage `Mini-ML` vers la machine abstraite `CAM` effectué par J. Despeyroux [31]. Cette démonstration est la première à utiliser la sémantique naturelle pour décrire les langages source et cible. Initialement faite à la main, cette démonstration a été vérifiée partiellement mécaniquement plus tard par J. Hannan et F. Pfenning [42] et complètement mécaniquement par S. Boutin [23]. Ce travail se distingue du nôtre, d'une part par l'absence de structure dans notre langage cible, d'autre part par notre description du compilateur comme un programme fonctionel pur au lieu d'une collection de règles d'inférence.

Pour ce travail nous avons été amenés à porter une attention particulière à deux aspects. D'une part, le langage cible utilise seulement des instructions de branchement et de branchement conditionnel pour gérer le contrôle de flux. La technique usuelle de raisonnement sur des programmes est "diviser pour conquérir". Nous avons établi des théorèmes qui indiquent dans quelles conditions on sait diviser un programme en fragments pour les conquérir indépendamment. D'autre part, le langage cible utilise seulement des instructions structurées pour gérer le contrôle de flux, il y a alors une grande distance entre la structure d'un programme source et celle du programme cible. Nous avons montré comment une technique systèmatique pouvait s'appliquer pour fournir un théorème d'induction générale pour réduire cette distance.

8.1 Compilation vers un langage sans structure

Comme langage cible du compilateur, nous avons choisi un langage qui représente assez fidèlement les capacités des langages assembleurs usuels, ce qui nous a amenés à étudier un langage avec étiquettes et instructions de branchement. Dans cette section, nous allons décrire brièvement la machine abstraite considérée et le langage assembleur.

8.1.1 un modèle réduit de processeur

Le processeur que nous considérons possède une mémoire illimitée pour les données, une mémoire pour les programmes, une pile illimitée, un registre, et un compteur ordinal. Les valeurs placées dans les cellules de la mémoire, dans la pile et dans le registre sont toujours des entiers naturels non bornés. Les instructions exécutées par cette machine peuvent être rangées en n catégories:

1. Il existe deux instructions qui agissent seulement sur le registre: la première `immediate` porte toujours une valeur entière et permet de charger cette valeur

dans le registre. La seconde `not`, met la valeur 1 dans le registre s'il contenait une valeur nulle avant exécution, et 0 dans les autres cas.

2. Les instructions `and`, `add`, `sub`, `gt` et `eq` agissent sur la pile et le registre: elles effectuent le calcul binaire indiqué par leur nom sur les arguments donnés par la valeur en sommet de pile et la valeur du registre. Le résultat est mis dans le registre et le sommet de pile est décrementé.
3. Les instruction `push` et `pop` agissent également sur le registre et la pile. La première incrémente le sommet de pile et y place la valeur du registre, la seconde prend la valeur en sommet de pile, la place dans le registre et décrémente le sommet de pile.
4. Les instructions `load` et `store` agissent sur le registre et la mémoire: ces deux instructions portent une valeur entière qui représente une adresse dans la mémoire des données. L'instruction `load` charge la valeur en mémoire à cette adresse dans le registre. L'instruction `store` place la valeur du registre en mémoire à cette adresse.
5. Les instructions `goto` et `branch_if_0` modifient le compteur ordinal: ces instructions sont accompagnées d'une valeur entière n et le programme continue à la n^{ieme} instruction. Notons que toutes les autres instructions se contentent d'incrémenter le compteur ordinal.

La machine fonctionne en commençant avec un compteur ordinal nul et s'arrête lorsque le compteur ordinal atteint une adresse de la mémoire du programme qui ne contient pas d'instruction.

Ce modèle réduit est réaliste dans la mesure où il suppose que le contrôle de flux se fait uniquement par passage à l'instruction suivante ou par branchement. Il n'est pas réaliste dans sa gestion des données, avec des valeurs non bornées une mémoire illimitée et une pile illimitée. Nous avons réalisé cette expérience surtout pour étudier les problèmes reliés aux contrôles de flux et nous avons délaissé les autres aspects.

8.1.2 Description du langage assembleur

Le langage assembleur utilisé comme langage cible permet déjà d'abstraire une caractéristique du processeur. Nous introduisons une instruction supplémentaire `label` qui sert à étiquetter des positions dans le programme à l'aide d'une valeur entière. Il s'agit d'une pseudo instruction, puisqu'elle ne jamais exécutée par le processeur et sera éliminée à l'assemblage. L'assembleur accepte donc les mêmes instructions que le processeur, mais les valeurs entières portées par les instructions de branchement `goto` et `branch_if_0` sont interprétées différemment: le branchement à lieu à l'étiquette correspondante plutôt qu'à une adresse absolue.

A ce langage assembleur correspond une machine d'exécution qui gère la mémoire, la pile et le registre exactement comme le processeur, mais gère la mémoire des programmes comme une longue liste d'instructions (parmi lesquelles peuvent apparaitre des étiquettes) et le compteur ordinal est lui même une liste d'instruction : la séquence des instructions qui seront exécutées tant que l'on n'aura pas de branchement.

L'intérêt du langage assembleur est de permettre la concaténation de fragments de programmes: si les étiquettes apparaissant dans les fragments sont distinctes, chaque fragment effectuera les mêmes calculs qu'il soit seul ou avec d'autres.

8.2 Théorèmes de fragmentation

Dans un langage sans branchement, exécuter la concaténation de deux fragments de programme correspond à exécuter d'abord le premier fragment puis le deuxième. Cette proposition n'est plus vraie dès que l'on fait intervenir des instructions de branchement. Nous avons décrit deux propriétés qui permettent de retrouver cette proposition, en nous plaçant à un niveau plus abstrait.

8.2.1 Un langage abstrait

Puisque nous nous intéressons seulement aux problèmes de branchement, il est intéressant de nous débarasser complètement des problèmes de calculs sur les données. Pour ce faire, nous avons introduit un nouveau langage abstrait qui ne contient plus que trois instructions: une instruction d'étiquetage, une instruction de calcul, et une instruction de branchement conditionnel. Les instructions d'étiquetage prennent en argument une valeur prise dans un type abstrait *Lab* quelconque, mais pour lequel l'égalité est décidable. Les instructions de calcul prennent comme argument une valeur prise dans un type abstrait *Inst* quelconque. Les instructions de branchement prennent deux arguments, l'un est pris dans le type *Inst* et l'autre est pris dans le type *Lab*. La machine quant à elle, est aussi décrite de façon abstraite en sachant que son état est décrit par des valeurs prises dans un type abstrait *state*, et que le comportement de chaque instruction est régi par deux fonctions abstraites:

$$\begin{array}{rcl} new\_state & : & Inst \rightarrow state \rightarrow state \\ test\_branch & : & Inst \rightarrow state \rightarrow bool \end{array}$$

La fonction *new_ state* est utilisée pour connaitre l'effet de chaque instruction de calcul et de chaque instruction de branchement sur l'état de la machine. La fonction *test_ branch* est utilisée uniquement pour les fonctions de branchement pour déterminer si le branchement doit avoir lieu.

8.2.2 Propriétés de base

Une première propriété importante des programmes en assembleur est que les étiquettes doivent être uniques. Cette propriété se vérifie aisément en parcourant le programme. Cette propriété est remarquablement stable vis-à-vis de la concaténation. D'une part, si toutes les étiquettes sont uniques dans la concaténation des programmes pg_1 et pg_2 elles le sont dans le programme pg_1 (respectivement pg_2) pris séparément. D'autre part, si la concaténation de pg_1 et pg_2 vérifie la propriété d'unicité alors la concaténation de pg_2 et pg_1 la vérifie aussi.

Une deuxième propriété importante des fragment de programmes en assembleur est d'être stable par référence (nous avons utilisé le terme *self-contained* en anglais), au sens où tout branchement dans le fragment considéré va faire sauter l'exécution vers une étiquette qui est également présente dans le fragment. Ici encore, cette propriété est stable vis-à-vis de la concaténation, mais d'une autre manière. D'une part, si pg_1 et pg_2 sont stables par référence, alors leur concaténation l'est aussi. D'autre part, si la concaténation de pg_1 et pg_2 est stable par référence alors la concaténation de pg_2 et pg_1 l'est aussi.

8.2.3 Théorèmes principaux

Nos travaux sur cette machine abstraite nous conduisent à deux théorèmes que nous appelons des programmes de fragmentation et qui expriment que les fragments stables par référence se comportent pratiquement comme des fragments de programmes sans branchements. Ceci s'exprime par les deux théorèmes suivants:

Théorème 1 (décomposition) *Si toutes les étiquettes dans $pg.pg_1.pg_2$ sont uniques, pg_1 est stable par référence et l'exécution de $pg_1.pg_2$ avec $pg.pg_1.pg_2$ comme programme complet et s comme état initial termine normalement dans l'état s', alors il existe un état intermédiaire s_1 tel que:*

1. *l'exécution de pg_1 avec pg_1 comme programme complet et s comme état initial termine normalement dans l'état s_1,*
2. *et l'exécution de pg_2 avec $pg.pg_1.pg_2$ comme programme complet et s_1 comme état initial termine normalement dans l'état s'.*

Théorème 2 (composition) *Si toutes les étiquettes dans $pg.pg_1.pg_2$ sont uniques, pg_1 est stable par référence et si les deux conditions suivantes sont vérifiées:*

1. *l'exécution de pg_1 avec pg_1 comme programme complet et s comme état initial termine normalement dans l'état s_1,*
2. *et l'exécution de pg_2 avec $pg.pg_1.pg_2$ comme programme complet et s_1 comme état initial termine normalement dans l'état s',*

alors l'exécution de $pg_1.pg_2$ avec $pg.pg_1.pg_2$ comme programme complet et s comme état initial termine normalement dans l'état s'.

Ces théorèmes expriment donc que les fragments stables par référence peuvent être étudiés indépendemment du context dans lequel on les exécute. Cette propriété est particulièrement intéressante pour nous car le compilateur produit systématiquement des programmes stables par référence.

Dans notre preuve de compilateur, nous avons vérifié la preuve de ces théorèmes sur le langage abstrait avant de les reporter sur le langage concret.

8.3 Récurrence générale

L'avantage de Coq est qu'il permet à l'utilisateur de définir de façon récursive des propriétés et calcule automatiquement le principe de récurrence associé à ces propriétés.

Dans notre cas, la machine d'exécution est décrite par une propriété *execute(pg, s, pg', s')* qui exprime que l'exécution du programme pg' avec pg comme programme total et s comme état initial termine normalement et retourne l'état final s'. Cette propriété est décrite par une définition récursive à deux constructeurs. Le premier constructeur indique que l'exécution d'un programme vide termine toujours et que l'état final et l'état initial sont identiques. Le second constructeur indique que pour exécuter un programme non vide, il faut calculer l'effet de la première instruction pour obtenir un nouvel état et un nouveau programme à exécuter et continuer en exécutant ce nouveau programme à partir du nouvel état. L'état final de l'exécution complète est alors identique à l'état final de la deuxième exécution.

Le principe de récurrence construit pour cette définition par Coq exprime que si l'on veut démontrer une propriété de pg, s, pg' et s' sachant que *execute(pg, s, pg', s')* est vrai, on peut le faire en démontrant que cette propriété est vraie lorsque $s = s'$ et pg' est vide et que l'on peut la démontrer sachant que la première instruction de pg' est i, que l'exécution de i produit un nouveau programme pg'' et un nouvel état s'', que *execute(pg, s'', pg'', s')* est vrai et que la propriété est déjà vérifiée pour pg, s'', pg'' et s' (ceci constitue l'hypothèse de récurrence).

Ainsi, on fait un raisonnement par récurrence en s'autorisant à utiliser l'hypothèse de récurrence sur les paramètres de l'exécution qui suit immédiatement le pas considéré.

Dans démonstration, ce principe de récurrence s'est montré inadapté pour travailler sur les boucles. Pour mieux comprendre, observons le résultat de la compilation d'une boucle *while(E,I)*, où E et I sont une expression et une instruction quelconques, et *[[E]]* et *[[I]]* représentent leur forme compilée.

`label(n).`*[[E]]*`.branch_if_0(n+1).`*[[I]]*`.goto(n).label(n+1)`

Pour vérifier le compilateur, on veut montrer que l'exécution de ce fragment de programme modélise correctement l'exécution d'une boucle *while*. Si on applique directement un raisonnement par récurrence, on est ennuyé: après exécution de la première instruction, `label(n)`, on se trouve seulement devant l'exécution du fragment complet sans l'étiquette dans le même état, ce qui n'est pas très intéressant.

En fait, lorsque l'état initial est choisi pour que E s'évalue à *true*, l'exécution exécute successivement *[[E]]*, `branch_if_0(n+1)`, *[[I]]*, `goto(n)` et revient au début du fragment de programme. Pour effectuer la preuve, on a besoin de montrer: *(i)* que l'exécution de *[[E]]* modélise bien l'évaluation de E, *(ii)* que l'exécution de *[[I]]* modélise bien celle de I et *(iii)* que l'exécution du fragment complet modélise bien l'exécution de *while(E,I)* à partir de l'état intermédiaire résultat de I. La preuve de *(i)* peut s'obtenir par un théorème similaire adapté à la compilation des expressions. La preuve de *(ii)* peut s'obtenir en faisant une preuve par récurrence sur la structure des instructions compilées, car I est un sous-terme strict de *while(E,I)*. En revanche, la preuve de *(iii)* ne peut pas s'obtenir de cette façon.

En observant attentivement nos besoins, on s'aperçoit que l'on a besoin que l'hypothèse de récurrence soit vérifiée pour un quadruplet *(pg, s", pg'', s')* qui n'est pas celui qui apparait directement après le quadruplet initial dans l'exécution considérée: nous serions capable de conclure si l'hypothèse de récurrence était vérifiée pour tous les états rencontrés après la première étape au lieu d'être seulement valable pour l'état suivant la première étape.

Le même phénomène se produit lorsque l'on fait des preuves par récurrences sur les entiers naturels. Le principe de récurrence primitif est le suivant:

$$\forall P.(P(0)\&(\forall n.P(n) \Rightarrow P(n+1))) \Rightarrow \forall n.P(n).$$

Mais les mathématiciens sont habitués à utiliser un principe beaucoup plus puissant, où l'on autorise à utiliser l'hypothèse de récurrence pour n'importe quel nombre plus petit que l'entier considéré:

$$\forall P.(\forall n.(\forall m.m < n \Rightarrow P(m)) \Rightarrow P(n+1))) \Rightarrow \forall n.P(n).$$

L'ordre $<$ joue le rôle d'un itérateur: $m < n$ signifie que n est obtenu à partir de m en appliquant un nombre arbitraire non nul de fois la fonction "+1". Ce théorème de récurrence général se prouve aisément à l'aide du principe de récurrence primitif.

A partir de cette observation, nous avons montré que l'on pouvait construire une relation *execute′(pg, s, pg′, s′, pg″)* dont le sens est que l'on peut exécuter un nombre arbitraire d'instructions élémentaire à partir de pg' avec pg comme programme global et s comme état initial, arriver à l'état intermédiaire s' et avoir encore le programme pg" à exécuter. Avec cette relation, on peut écrire le principe de récurrence suivant:

$\forall P.$
$(\forall pg{,}s{,}pg'{,}s'.$
$\quad(\forall s''{,}pg''.\ execute'(pg{,}s{,}pg'{,}s''{,}pg'') \Rightarrow execute(pg{,}s''{,}pg''{,}s') \rightarrow P(pg{,}s''{,}pg''{,}s')) \Rightarrow$
$\quad execute(pg{,}s{,}pg'{,}s') \Rightarrow P(pg{,}s{,}pg'{,}s')) \Rightarrow$
$\forall pg{,}s{,}pg'{,}s'.\ execute(pg{,}s{,}pg'{,}s') \Rightarrow P(pg{,}s{,}pg'{,}s')$

Cet énoncé semble assez compliqué mais il copie très fidèlement l'énoncé du théorème de récurrence général sur les entier. La démonstration de cet énoncé est également très facile à obtenir.

Ce théorème de récurrence général pour l'exécution des programmes assembleur s'applique ensuite très bien pour effectuer la preuve du point *(iii)* cité plus haut.

8.4 Conclusion

La vérification de la correction de ce compilateur est la plus grosse démonstration que j'ai effectuée en utilisant Coq et CtCoq. l'ensemble des scripts de preuve réunis arrive à un total de 8000 lignes (191 lemmes). Arrivé à cette taille, CtCoq devient très utile, du point de vue des notations, du point de vue de la gestion de script et du point de la génération des commandes. Même, l'outil d'interprétation des mouvements s'est montré utile pour manier des concaténations de programmes assembleur, dans les démonstrations d'unicité des étiquettes et de cohérence.

Ce travail pourra être complété du coté du modèle réduit de processeur, en décrivant comment l'assembleur enlève les étiquettes et ajuste les branchement et en vérifiant la correction de cet algorithme. On pourrait également étendre le langage source pour y permettre l'utilisation de tableaux, des structures de contrôle de type switch, et des procédures récursives, en ajoutant au fur et à mesure les instructions adéquates dans l'assembleur. Une fois cette puissance atteinte, le langage source pourrait à son tour être utilisé comme langage cible pour la compilation de langages de plus haut niveau.

Une autre direction de recherche possible est de comprendre comment les compétences acquises sur ce modèle réduit peuvent s'utiliser sur d'autres langages assembleurs comme le langage de la machine abstraite Java pour lequel on dispose d'une description formelle [62].

Chapitre 9

Perspectives

Au moment de conclure cette revue des travaux effectués au cours de ces dix années de recherche, nous pouvons maintenant nous tourner vers l'avenir et examiner les perspectives ouvertes pour les différents thèmes abordés. Nous allons reprendre les différents axes de recherche, cette fois-ci en les classant suivant l'importance qu'ils occupent dans mon travail d'aujourd'hui.

9.1 Travaux sur les environnements de preuve

Ces dernières années, mes travaux sur les environnements de preuve ont profité d'un contexte favorable. En particulier, des contacts avec des équipes du milieu industriel, comme l'équipe d'E. Ledinot chez Dassault-Aviation et celle J.-F. Monin au CNET ont permis d'ajuster les objectifs de recherches en fonction de ces utilisateurs intensifs et de trouver des ressources pour attirer un ingénieur, J. Bertot, et un doctorant, O. Pons, sur ce thème de recherche. L'interface de preuve CtCoq est l'application principale pour ce domaine de recherche. Cette interface, disponible sur le réseau Internet, nous permet d'avoir de bons contacts avec la communauté des utilisateurs de systèmes de preuve et avec tous les chercheurs qui s'intéressent aux problèmes des interfaces utilisateurs pour les systèmes de preuve. Par exemple, les membres de l'équipe ont participé aux trois dernières éditions du colloque UITP (User-Interfaces for Theorem Provers) et j'ai organisé l'édition 97 et aux comités de programmes deux éditions de la conférence TPHOL, et nous participerons à l'organisation de cette conférence pour l'année 99.

Dans la suite, nous allons voir que l'avenir de cette direction de recherche est profondément marqué par des considérations technologiques. Cela peut sembler dommage, dans la mesure où ces considérations sont relativement éloignées de considérations scientifiques. Mais ces considérations donnent de la crédibilité à notre effort de fournir des outils réellement utilisés, ce qui motive notre effort de recherche.

9.1.1 Choix technologiques

Nos recherches sont relativement liées aux caractéristiques du système Coq, également développé à l'INRIA. Ce choix peut être discuté. En effet, le système Coq est issu d'un travail très théorique et son utilisation passe par un langage particulier qui le rend dificilement accessible. S'intégrer dans la communauté des utilisateurs de Coq n'est peut-être pas le meilleur moyen d'éviter l'isolement vis-à-vis de la communauté scientifique. En outre, le système Coq est encore un système jeune où les choix de conceptions peuvent être remis en cause assez brutalement. La conception du système CtCoq souffre alors de cette instabilité, surtout si les points qui sont sujets à modification sont proches de l'interface homme-machine. A titre d'exemple, le système Coq a utilisé quatre techniques d'analyse syntaxique différentes depuis les premières versions du système. Ces interrogations sont compensées par des arguments en faveur du choix du système Coq. Premièrement, il serait déraisonnable de développer notre propre moteur de preuve et il n'existe pas d'alternative viable. Les expériences que nous avons menées avec HOL et Isabelle ont montré que le développement d'une interface de preuve pour un système dont le langage de commande est directement le langage ML est une tâche beaucoup plus difficile. De plus, il faut noter que l'accueil bienveillant réservé à nos travaux par l'équipe Coq est un atout précieux[1]: l'équipe Coq fournit une aide importante pour la description du fonctionnement interne du système et accepte volontiers d'intégrer des modifications pour nos besoins.

En marge de notre travail avec Coq, nous pouvons également envisager de travailler avec les équipes industrielles qui fournissent des environnement de travail pour la méthode B, et qui ont également besoin d'environnements de preuve. Les premiers contacts avec ces équipes ont été pris grâce aux travaux de R. Fraer, qui ont montré que les outils Centaur pouvaient fournir de l'aide dans ce domaine, et particulièrement les techniques de suivi de position. L'avenir de collaborations dans ce domaine dépend plus de l'aboutissement de ces prises de contacts (et ici la crédibilité de nos choix technologiques est importante). Pour l'instant, la réponse n'est pas si positive qu'on pourrait l'espérer.

En deuxième lieu, l'implémentation actuelle de CtCoq doit beaucoup au système Centaur et ce système a un avenir incertain. Le travail à venir dans ce domaine doit donc s'accompagner d'un examen des composants qui constituent une interface comme CtCoq, pour comprendre leur dépendance vis-à-vis de Centaur. Nous avons déjà entamé cet examen. Par exemple, nous avons étudié comment les notions de preuves utilisant la souris, comme la preuve par sélection, peuvent être implémentées sans utiliser d'outils de manipulation structurée, avec des éditeurs de textes comme Emacs, ou les éditeurs de textes fournis par la librairie graphique Tcl/Tk. De plus, nous voulons expérimenter avec le module d'affichage d'arbre en cours de ré-implémentation en Java dans le projet. Il faudra aussi étudier comment les navigateurs hypertextes popularisés par le "World-Wide-Web" pourront être utilisés pour effectuer une communication efficace avec le système de preuve, en utilisant des formats d'échange standards basés sur XML et le protocole HTTP.

1. Même s'il est décevant de n'avoir pas pu convaincre un membre de cette équipe d'utiliser CtCoq pour ses propres développements de preuve.

9.1.2 Améliorations de CtCoq

Dans sa forme actuelle, le système CtCoq fournit effectivement de l'aide dans la tâche d'écrire des preuves simples. Lorsque l'on cherche à effectuer de grandes preuves, de nouveaux problèmes surgissent et sont autant de nouveaux sujets de recherche. Nous allons ici nous concentrer sur deux aspects.

En premier lieu, les grandes expériences de preuves réalisées dans notre équipe montrent que l'utilisateur a souvent besoin de revenir en arrière. Nous sommes en train de réfléchir a ce problème dans le cadre de la thèse d'O. Pons, co-encadrée avec L. Rideau. Dans ce travail, nous remarquons que tous les outils que nous pouvons envisager pour aider à résoudre ce problème tournent autour d'une analyse de dépendance entre les objets mathématiques ou entre les commandes envoyées au système de preuve.

En second lieu, tous les meilleurs systèmes de preuve sur ordinateur fournissent à l'utilisateur la possibilité de programmer ses propres procédures de recherche de preuve. Le langage ML est d'ailleurs né de cette tradition, puisque c'est le langage fourni à l'utilisateur dans les systèmes descendants de Edinburgh LCF. Le système utilise également ce langage pour décrire les tactiques, même si l'intégration d'une nouvelle tactique se fait de manière moins souple que dans HOL ou Isabelle, où ML est directement le langage d'interaction avec le système de preuve. Cette volonté des concepteurs de Coq de mettre un peu de distance entre ML et le langage de commande du système de preuve est un bon point: cela rend les scripts de preuve plus stables en fixant l'ensemble de procédures utilisées par la communauté et facilite la maintenance à long terme des preuves. Pour le développement de l'interface homme machine, cette distance est également un avantage: elle assure que l'activité de programmation de nouvelles tactiques est bien comprise comme une activité distincte de l'activité de preuve et permet donc d'envisager des outils spécialisés pour cette tâche. Nous n'avons encore commencé aucun travail sur ce sujet, mais les travaux effectués par L. Rideau et L. Théry sur un environnement de programmation pour ML représentent un atout important pour aborder ce domaine. L'objectif de ce travail de recherche sera de diminuer le niveau d'expertise en Coq et en ML nécessaires pour programmer ses propres tactiques. En particulier, il est possible que les techniques de "programmation par l'exemple" comme celles qui apparaissent dans notre travail sur l'interprétation des mouvements aident à ce type de développement.

9.1.3 Perspectives à long terme

A long terme nous envisageons de faire évoluer ce travail sur les environnements de preuve vers un environnement de travail mathématique qui rapproche déduction et calcul formel. Avec les algorithmes de calcul formel déjà vérifiés, nous disposons maintenant d'un noyau important de chercheurs intéressés par l'étude de ces deux aspects du calcul mathématique sur ordinateur (L. Rideau, L. Théry, L. Pottier, et moi-même). Ici encore, les spécificités du système Coq sont un atout pour cette direction, car l'extraction de programme permet d'obtenir des programmes très compétitifs à partir de nos preuves d'algorithmes, et les techniques de réflexions décrites par S. Boutin permettent d'envisager l'utilisation de certains de ces algorithmes directement dans le système de preuve. Enfin, notons que l'objectif d'intégrer calcul formel et déduction est également perçu par plusieurs équipes à travers le monde et que nous pourrons nous intégrer à cette communauté.

9.2 Preuves en sémantique

Le travail sur les preuves en sémantique a également bénéficié d'un contexte favorable dans le projet CROAP. En effet, ce domaine a été étudié depuis assez longtemps en prolongement des recherches sur la sémantique des langages de programmation, avec en particulier une preuve à la main de compilateur pour le noyau de langage Mini-ML. Par la suite, les travaux de Delphine Terrasse ont permis d'établir une passerelle entre les spécifications TYPOL et les preuves en Coq. Pour l'avenir, la preuve de compilateurs a éveillé un intérêt réel chez des collaborateurs industriels comme Dassault-Aviation. La raison de cet intérêt est que les langages dédiés comme LUSTRE sont intéressants pour ces industriels qui veulent générer avec ces langages du code qui sera embarqué dans des véhicules. Or ce code embarqué doit être validé par un organisme de certification et une façon possible d'obtenir cette validation est de vérifier formellement le générateur de code. Notre travail sur la preuve d'un compilateur pour un langage académique répond en partie à la volonté de disposer d'une expérience pour ce genre de tâche.

On peut s'interroger sur la compétitivité de notre approche, basée sur la sémantique naturelle et sur Coq. Notons que Coq peut représenter un atout important dans la mesure où ce système permet de prouver formellement la correction d'un algorithme et d'extraire automatiquement un code efficace pour cet algorithme, ce qui évite la réécriture manuelle du programme et les erreurs qui pourraient en découler. Cet atout apparaît surtout si l'on n'utilise pas le langage TYPOL mais le langage fonctionnel pur présent dans Coq pour décrire le compilateur et les extensions fournies pour travailler sur les programmes impératifs [37]. Enfin, notons que nos expérience de vérification d'outils de manipulation de programmes peuvent également contribuer à notre objectif de travailler sur un environnement pour la programmation certifiée.

9.3 Travaux sur les calculs de position

Depuis 1991, J'ai délaissé les travaux sur le calcul de position en sémantique au profit des travaux sur les preuves. Si les techniques sont décrites précisément dans deux articles et dans la thèse de doctorat, il n'existe en revanche pas d'outil permettant de mettre en œuvre rapidement ces techniques. L'histoire récente du projet CROAP conduit à s'interroger sur la pertinence de cet abandon. Les travaux réalisés par S. Dissoubray et R. Fraer ont souffert de l'absence d'un tel outil. Un argument supplémentaire pour la reprise de ces travaux est l'avènement du langage Java, pour lequel il sera utile de fournir une panoplie d'outils pour faciliter la mise au point de programmes concurrents (en utilisant les librairies de tâches concurrentes, *multi-threads* en anglais). Or, les calculs concurrents se modélisent typiquement par des spécifications de sémantique opérationnelle structurelle "little-step", pour lesquelles le mécanisme traditionnel de suivi de sujet de TYPOL échoue.

9.4 Le projet CROAP et Le système Centaur

Mes recherches ont été fortement influencées par les activités présentes dans l'équipe CROAP, d'abord parce que ce projet, sous la houlette de Gilles Kahn, m'a accueilli et m'a fourni mes premiers sujets de recherche, ensuite parce que j'ai assuré la direction de cette équipe de 1995 à 1998. Une grande partie de cette influence passe par le système Centaur, dont il est important de comprendre la position actuelle et le devenir.

Au cours de ces dix dernières années, le système Centaur a représenté un atout formidable pour le projet CROAP, dans la mesure où il a permis d'explorer des problèmes de recherche et des applications inaccessibles sans cette avance technologique. Ainsi Centaur, d'abord objet de nos recherches, est devenu un outil crucial pour de nouvelles recherches abordant des sujets nouveaux. Maintenant Centaur représente également des risques importants pour l'avenir.

Le système Centaur souffre actuellement d'un avenir incertain, dû à la disparition prochaine d'un support industriel pour le système Le_lisp sur lequel il est construit. Cette perspective diminue les chances de voir ce système adapté aux nouveaux systèmes d'exploitation, elle entrave les négociations de collaborations avec des partenaires industriels et elle rend le système beaucoup moins attrayant pour les chercheurs du monde académique. En réponse à cette situation, nous avons fait des efforts pour rendre nos travaux de recherche de plus en plus indépendants de Centaur. Cette effort culminera bientôt, nous l'espérons par la distribution d'une version de l'environnement de preuve complètement indépendante de Centaur et Le_Lisp.

9.5 Conclusion et remerciements

Les travaux décrits dans ce document trouvent tous leur justification et leur inspiration dans une approche volontairement expérimentale. Si certains aspects de ces travaux prennent parfois un tour théorique, il s'agit souvent de décrire un phénomène apparaissant lors de l'utilisation concrète de l'ordinateur pour résoudre des problèmes pratiques, de décrire de façon plus abstraite des outils dont il existe d'abord un prototype qui a montré son utilité, ou bien de mettre en pratique des idées théoriques inspirées par des collaborateurs. Dans mon travail d'encadrement, j'ai également tenté d'insuffler ce souci de pragmatisme aux jeunes chercheurs.

Cette démarche expérimentale a fourni des résultats d'autant meilleurs qu'elle a pu s'intéresser dans des efforts collectifs, d'abord autour de Centaur puis autour de CtCoq. A ce titre je voudrais remercier tous les membres de l'équipe CROAP qui m'ont accueilli, qui m'ont fait part de leur suggestions, qui m'ont fourni des outils originaux, qui ont participé aux développements que je proposais, et qui ont testé et les logiciels qui ont été produit. Mais bien sûr, comme c'est toujours le cas, il y a des collaborateurs pour qui j'ai une gratitude particulière et je voudrais nommer ici Janet Bertot, Ranan Fraer,

Gilles Kahn, Francis Montagnac, Laurence Rideau, et Laurent Théry, qui ont été a des degrés divers des co-auteurs de nombre des travaux décrits ici.

Mon travail de recherche a également été marqué par la responsabilité que j'ai prise de diriger le projet CROAP après le départ de Gilles Kahn. Dans cette tâche, le travail autour de CtCoq a été un atout important car il a permis de focaliser les recherches de plusieurs chercheurs en donnant un objectif collectif à long terme. En revanche, cet objectif n'a pas convaincu l'ensemble de l'équipe et j'ai du également prendre en compte l'existence dans le projet d'objectifs divergents. Avec le recul, cette situation a été une source d'enrichissement, dans la mesure où elle m'a forcé a réfléchir souvent aux motivations de mes collègues, à l'évolution du contexte de nos recherches et à la façon de maintenir dans une équipe un climat propice aux recherches. Dans ces réflexions j'ai toujours été aidé par mes collègues chercheurs ou professeurs. Ici encore je voudrais remercier tout particulièrement Gilles Kahn et Michael Brady pour leurs conseils amicaux.

Enfin, le travail de tout chercheur se nourrit de l'appréciation critique de ses pairs. A ce titre, je tiens à remercier Jean-Jacques Lévy, Bengt Nordström et Christine Paulin d'avoir accepté d'évaluer mon travail en étudiant ce mémoire et les différents articles que j'ai publiés pour décrire mes recherches. Je remercie également Henk Barendregt et Véronique Viguié-Donzeau-Gouge pour l'intérêt qu'ils montrent en acceptant de participer au jury de cette habilitation.

Bibliographie

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *Principles Of Programming Languages*, 1990.

[2] J.-R. Abrial. *The B-Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.

[3] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(5), Nov. 1996.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[5] J. Bertot and Y. Bertot. CtCoq: A system presentation. In *Automated Deduction (CADE-13)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 231–234. Springer-Verlag, July 1996.

[6] J. Bertot and Y. Bertot. The ctcoq experience. In *Electronic proceedings for the conference UITP'96*, University of York, July 1996.

[7] J. Bertot, Y. Bertot, Y. Coscoy, H. Goguen, and F. Montagnac. User guide to the CtCoq proof environment. Rapport technique RT0210, INRIA, 1997.

[8] Y. Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *European Symposium On Programming.* Springer Verlag, 1990. LNCS 432.

[9] Y. Bertot. Occurrences in Debugger Specifications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[10] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, University of Nice-Sophia Antipolis, 1991. (In french).

[11] Y. Bertot. Origin Functions in λ-calculus and Term Rewriting Systems. In *CAAP'92*, 1992. Springer Verlag LNCS 581.

[12] Y. Bertot. A Canonical Calculus of Residuals. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 147–163. Cambridge University Press, 1993. (Also appears as INRIA Report no. 1542, Oct. 1991).

[13] Y. Bertot. Direct manipulation of algebraic formulae in interactive proof systems. In *Electronic proceedings for the conference UITP'97*, Sophia Antipolis, Sept. 1997.

[14] Y. Bertot. Head-tactics simplification. In *Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[15] Y. Bertot. Programming language specifications and environments. Rapport technique RT0212, INRIA, 1997.

[16] Y. Bertot. A certified compiler for an imperative language. Research Report RR-3488, INRIA, 1998.

[17] Y. Bertot. The CtCoq system: Design and architecture. *Formal Aspects of Computing*, 1999. to appear.

[18] Y. Bertot and R. Fraer. Reasoning with Executable Specifications. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 531–545, 1995.

[19] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160, 1994.

[20] Y. Bertot, T. Kleymann-Schreiber, and D. Sequeira. Implementing proof by pointing without a structure editor. Technical Report ECS-LFCS-97-368, University of Edinburgh, 1997. également rapport de recherche INRIA 3286.

[21] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25:161–194, 1998.

[22] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Symposium on Software Development Environments*, 1988. (Also appears as INRIA Report no. 777).

[23] S. Boutin. Preuve de correction de la compilation de min-ML en code CAM dans le système d'aide à la démonstration COQ. Rapport de recherche RR2536, INRIA, Apr. 1995.

[24] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic. In *10th Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science. Springer-Verlag, July 1990.

[25] D. Cameron and B. Rosenblatt. *Learning GNU Emacs.* O'Reilly & Associates, Inc., 1991.

[26] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Aug. 1986.

[27] D. Clément. A distributed architecture for programming environments. *Software Engineering Notes*, 15(5), 1990. *Proceedings of the 4th Symposium on Software Development Environments.*

[28] D. Clément, F. Montagnac, and V. Prunet. Integrated software components: a paradigm for control integration. In *Proceedings of the European Symposium on Software Development Environments and CASE Technology*, volume 509 of *Lecture Notes in Computer Science*, June 1991.

[29] R. 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.

[30] C. Cornes and D. Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[31] J. Despeyroux. Proof of translation in natural semantics. In *Symposium on Logic in Computer Science*, pages 193–205. IEEE Computer Society, June 1986.

[32] J. Despeyroux. Theo: an interactive proof development system. *Scandinavian Journal on Computer Science and Numerical Analysis, BIT*, 32:15–29, 1992.

[33] T. Despeyroux. Executable Specifications of Static Semantics. In *International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.

[34] E. W. Dijkstra. *A discipline of Programming*. Prentice Hall, 1976.

[35] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[36] A.-M. Déry and L. Rideau. Distributed programming environments: an example of message protocol. Rapport Technique 165, INRIA, 1994.

[37] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[38] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science : 19th symposium on Applied Mathematics*, pages 19–31, 1966.

[39] R. Fraer. Tracing the origins of verification conditions. In *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*. Springer Verlag, 1996.

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

[41] J. D. Guttman and M. W. (eds.). Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), 1995.

[42] J. Hannan and F. Pfenning. Compiler verification in lf. In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, 1992.

[43] T. Hardin and J.-J. Lévy. A Confluent Calculus of Substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989.

[44] L. Hascoët. Partial evaluation with inference rules. *New Generation Computing Journal*, 6 (2 & 3):187–209, 1988. Selected Papers of the Workshop on Partial Evaluation and Mixed Computation, Gammel Averneas, Denmark.

[45] L. Hascoët. A tactic-driven system for building proofs. In *actes du 7e Séminaire de Programmation en Logique*, 1988. also INRIA Research Report 770 (1987).

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.

[47] INRIA. *The Coq Proof Assistant Reference Manual*, December 1996. Version6.1.

[48] I. Jacobs, F. Montagnac, J. Bertot, D. Clément, and V. Prunet. the Sophtalk reference manual. Rapport Technique 150, INRIA, 1993.

[49] G. Kahn. Natural Semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*. North-Holland, 1988. (also appears as INRIA Report no. 601).

[50] N. Kajler. CAS/PI: A portable and extensible interface for computer algebra system. In *International Symposium on Symbolic and Algebraic Computation (ISSAC'92)*, pages 376–386. ACM, 1992.

[51] P. Klint. A meta-environment for generating programming environments. *ACM Transaction on Software Engineering and Methodology*, 2:176–201, 1993.

[52] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version. Available by anonymous ftp with LEGO distribution.

[53] J. McCarthy and J. Painter. correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science 1*, volume 19 of *Proceedings of Symoposia in Applied Mathematics*, 1967.

[54] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[55] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing series. Addison-Wesley, 1994.

[56] L. C. Paulson and T. Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[57] G. Plotkin. Structural operational semantics. lecture notes DAIMI FN-19, Aarhus University, 1981. (reprinted 1991).

[58] W. Polak. *Compiler Specification and Verification*, volume 124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[59] O. Pons. Undoing and managing a proof. In *Electronic Proceedings of "User Interfaces for Theorem Provers 1997"*, Sophia-Antipolis, France, 1997.

[60] N. Shankar, S. Owre, and J. M. Rushby. A tutorial on specification and verification using PVS. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993. (Beta Release).

[61] Stanford Verifier Group. Stanford pascal verifier user manual. Technical Report 11, Stanford Verification Group, 1979.

[62] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM Press, Jan. 1998.

[63] J. E. Stoy. *Denotational Semantics — The scott-Starchey Apprach to language theory*. MIT Press, 1977.

[64] M. E. Szabo. *The Collected papers of Gerhard Gentzen*. North-Holland, 1969.

[65] D. Terrasse. *Vers un Environnement d'Aide au Développement de Preuves en Sémantique Naturelle*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 1995.

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

[67] L. Théry. *Une méthode distribuée de création d'interfaces et ses applications aux démonstrateurs de théorèmes*. PhD thesis, University of Paris VII, 1994.

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

[69] A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special issue on automatic programming.

[70] G. Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.

[71] W. D. Young. A mechanical verified code generator. *Journal of Automated Reasoning*, 5:493–518, 1989.