INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Action CERTILAB

Spécifications formelles, certification de logiciel

Sophia Antipolis

THÈME 2A

Table des matières

1 Composition de l'équipe **2**

2 Présentation et objectifs généraux **2**

3 Fondements scientifiques **3**
3.1 Preuve de propriétés de langages . . . 3
3.1.1 Étude d'exemples . . . 3
3.1.2 Méthodes . . . 4
3.1.3 Théories typées . . . 7
3.2 Preuve de correction de programmes . . . 8
3.2.1 Étude d'exemples . . . 8
3.2.2 Méthodes . . . 8
3.3 La part de rêve, ou veille technologique: mobilité, réseaux . . . 9

4 Domaines d'applications **10**
4.1 Panorama . . . 10
4.2 Logiciel embarqué . . . 10
4.3 Programmation sur internet . . . 11

5 Résultats nouveaux **12**
5.1 Resultats . . . 12
5.1.1 Formalisation de langages . . . 12
5.1.2 Theories typées . . . 12
5.1.3 Programmation certifiée . . . 12

6 Contrats industriels (nationaux, européens et internationaux) **13**
6.1 Action de développement Génie . . . 13

7 Actions régionales, nationales et internationales **13**
7.1 Action de Recherche Coopérative XXX . . . 13
7.2 Groupe de travail européen Types . . . 13

8 Diffusion de résultats **13**
8.1 Animation de la communauté scientifique . . . 13
8.2 Enseignement universitaire . . . 14
8.3 Thèses et stages . . . 14
8.3.1 Thèses en cours dans le projet . . . 14
8.3.2 Stages effectués dans le projet . . . 14
8.4 Participation à des colloques, séminaires et invitations . . . 14

9 Bibliographie **14**

1 Composition de l'équipe

Responsable scientifique

Joëlle Despeyroux [CR Inria]

Assistante de projet

Nathalie Bellesso [à temps partiel]

Chercheurs doctorants

Guillaume Gillard [allocataire MENESR]

Collaborateurs extérieurs

André Hirschowitz [professeur, université de Nice]

Stagiaires

Pierre Corbineau [stage de première année de l'ENS-Ulm]

Etienne Lozes [stage de première année de l'ENS-Lyon]

2 Présentation et objectifs généraux

Le thème de notre projet est la *certification de logiciel*. Les programmes sont souvent écrits dans des langages et outils spécialisés ; leur certification nécessite donc deux parties : d'une part la validation d'outils généraux tels que des compilateurs, d'autre part la preuve de correction des programmes eux-mêmes. Notre projet recouvre ces deux phases de développement des logiciels.

La preuve de correction de compilateurs (pris dans leur intégralité : vérification de propriétés statiques, dont vérification ou inférence des types, puis traduction vers un formalisme intermédiaire ou vers du code) est l'exemple typique de *preuve de propriétés de langage*.

La *preuve de correction de programmes* consiste généralement à prouver certaines propriétés d'un programme, typiquement la validité d'un invariant. Cependant, dans le cas idéal où l'on peut extraire un programme à partir de sa spécification, la preuve de correction de programmes devient *programmation certifiée*.

Ce thème de recherche (spécification et preuves sur machine) est en plein essor. De nombreux laboratoires de recherche industriels (Intel, Microsoft, ...) sont demandeurs de chercheurs ou ingénieurs ayant une formation large dans le domaine. De plus, cette demande devrait encore grandir, vu que l'informatique est de plus en plus utilisée dans des domaines critiques pour la vie humaine (comme le nucléaire, l'industrie avionique ou ferroviaire), ou bien dans des domaines où les enjeux financiers sont importants (comme le spatial, les télécommunications ou le domaine bancaire).

Notre projet de recherche est *transversal*. Notre objectif est de pouvoir proposer notre expérience à toute équipe intéressée à formaliser des preuves en machine. Comme nous le

verrons plus loin, nous avons déjà des collaborations dans ce sens avec différentes équipes, à l'INRIA comme à l'extérieur.

Notre démarche est essentiellement pragmatique, dirigée par des applications concrètes; notre projet est essentiellement *applicatif*. L'objectif est de développer des méthodes et des bibliothèques de spécifications et de preuves dans le domaine de la certification de logiciel, tout en contribuant au développement d'une des théories sous-jacentes au domaine : la théorie des types.

Nous nous intéressons à la chaîne de développement complète : *spécification, prototypage et preuves*, réalisée en machine en utilisant des systèmes comme le système Coq. Ce système a pour nous une conjugaison unique d'avantages : ses fondements théoriques sont solides, on peut extraire un programme à partir d'une spécification et de plus le noyau du système a lui-même été certifié.

Notre recherche est essentiellement dirigée par les applications à moyen terme et à long terme. Ces applications sont multiples, et peuvent être classées en deux catégories : les preuves de propriétés de langages et les preuves de correction de programmes et programmes certifiés. Nous avons déjà acquis une certaine compétence dans le premier domaine. Nous nous proposons de compléter notre programme de recherche en abordant le second, dans lequel la demande industrielle s'annonce très forte à plus ou moins brève échance, quand elle ne l'est pas déjà.

3 Fondements scientifiques

3.1 Preuve de propriétés de langages

Participants : Joëlle Despeyroux, André Hirschowitz, Guillaume Gillard.

C'est un domaine dans lequel nous avons acquis une compétence reconnue depuis de nombreuses années, aussi bien sur le plan pratique (étude d'exemples, développement d'un logiciel [5]) que sur le plan théorique (élaboration de méthodes [1] et propositions de théories typées [8, 3, 9, 2]).

3.1.1 Étude d'exemples

Participants : Joëlle Despeyroux, Guillaume Gillard.

Deux exemples typiques de preuve de propriétés de langages sont la preuve de conservation des types d'un langage lors de son exécution, et la preuve de correction d'un compilateur. La première propriété assure une certaine cohérence entre le système de vérification de type et les règles d'évaluation d'un langage. Ce type de preuve, essentiel pour les concepteurs du langage, donne par ailleurs au programmeur une certaine confiance dans les outils qu'il utilise.

Nous avons réalisé l'étude de nombreux exemples sur de petits langages impératifs (Imp de Winskel)) ou applicatifs classiques (Mini-ML) [4, 1] sur papier en 86, en machine depuis, principalement dans le système Coq. Ces exemples sont repris dans une présentation récente dans les notes de cours rédigées pour le Dea MDFI de Marseille [6].

Delphine Kaplan-Terrasse, ancien doctorant dans notre projet, a réalisé la certification du

noyau du compilateur d'Esterel en Coq. Cette preuve intéresse plusieurs compagnies industrielles dont Dassault.

Dans le cadre du projet Génie 2, nous collaborons avec les équipes de recherche de Dassault-Aérospatiale sur la certification du compilateur du langage Lustre, langage réactif à flot de données.

Nous avons abordé récemment (1997) l'étude des langages concurrents et à objets, par le biais des calculs (π-calcul de Milner et sigma-calcul de Martin Abadi et Luca Cardelli). L'objectif de ces études est de construire les briques de base qui permettront de formaliser des langages comme OCaml ou Java, par exemple, de manière raisonnable. C'est ce type de langages (les langages concurrents et à objets) qui nous intéresse le plus à court et moyen terme, dans une perspective à la fois de recherche et de contrats avec l'industrie.

Le modèle des langages concurrents est le π-calcul. La difficulté principale, et l'intérêt de cette exemple est la description du phénomène d'extrusion de portée des variables. Nous avons réalisé plusieurs étude formelles (spécifications et preuves) du π-calcul, en suivant différentes méthodes de description des variables (voir section 3.1.2). L'approche donnant les spécifications les plus lisibles et les preuves les plus courtes est une technique qui consiste ici à utiliser directement les fonctions de Coq pour décrire les liaisons des variables du π-calcul (on appelle ce type de description une spécification d'ordre supérieur -3.1.2).

L'objectif de la thèse de Guillaume Gillard, commencée en octobre 97, est de formaliser des calculs ou langages concurrents et à objets, basés sur le π-calcul et le sigma-calcul, dans le système Coq. Guillaume a étudié plusieurs propositions de tels calculs et a choisi de formaliser un calcul proposé récemment par Andrew Gordon, en utilisant la technique de représentation des langages introduite par ce même Andrew Gordon (3.1.2). Cette méthode semble permettre des spécifications relativement proches de ce que l'on écrit sur le papier.

3.1.2 Méthodes

Participants : Joëlle Despeyroux, André Hirschowitz, Guillaume Gillard.

Nous avons proposé une méthode de preuve de correction de traduction en Sémantique Naturelle [4], qui fait toujours référence dans le domaine des preuves de correction de compilateur, plus de dix ans après.

Description des variables dans un langage fonctionnel Aussi surprenant que cela puisse paraître, il n'y a toujours pas de consensus sur le choix de la représentation des variables dans un langage fonctionnel.

On dispose à l'heure actuelle de quatre familles de méthodes pour décrire les variables d'un langage de programmation fonctionnel, et leur manipulation, c'est à dire essentiellement la substitution, à α conversion près (un terme est défini modulo le renommage de ses variables). Nous connaissons bien trois d'entre elles, que nous avons encore expérimentées cette année dans le système Coq, sur le π-calcul.

La technique la plus connue est celle des indices de Bruijn. Les variables sont implémentées par des entiers, représentant leur profondeur dans le terme, i.e. le nombre de liaisons entre leur occurence et la racine du terme. Par exemple, les termes $\lambda x.x$ et $\lambda x.\lambda y.x$ seront respectivement

représentés par les termes $(lam\ 0)$ et $(lam\ (lam\ 1))$. Bien sûr, les termes sont bien définis à α-conversion près, mais on imagine aisément tous les problèmes causés par ce type de représentation. Les termes étant eux-mêmes déjà illisibles, les spécifications le sont naturellement aussi, d'autant plus qu'un tel terme doit être complètement réécrit lorsqu'il est plongé dans un autre terme (les entiers doivent tous être décallés) ! Ce problème rend les preuves arbitrairement compliquées. Dans certains développements (λ et π-calculs, par exemple), les trois quarts du développement (spécification et preuves) concernent la manipulation des codes de Bruijn, et non la sémantique elle-même. La preuve de la propriété de conservation des types (simples) pour un π-calcul polyadique semble déjà déraisonnable, sinon impossible, dans cette approche. Cette technique, encore très utilisée, paraît donc hautement inadaptée à la formalisation des langages de programmation. Cependant, il existe un cas où l'utilisation des codes de Bruijn est tout à fait indiquée : le cas du *boot-strap* d'un langage implémenté en utilisant ces codes. C'est le cas de la formalisation de Coq en Coq, réalisée par Bruno Barras.

Nous explorons en ce moment une technique due à Andrew Gordon. Cette méthode, implémentée dans le système HOL, consiste à spécifier les termes du lambda-calcul à α-conversion près, sur une syntaxe utilisant des codes de Bruijn (qui disparaissent dans la syntaxe finale). La méthode semble permettre des spécifications relativement proches de ce que l'on écrit sur le papier, une fois prouvé un ensemble de lemmes sur les spécifications contenant des codes de Bruijn. À notre connaissance, cette méthode n'avait encore jamais été vraiment utilisée.

James McKinna et Robert Pollack utilisent pour spécifier des règles de typage (typage des théories typées) une technique d'ordre un intéressante qui distingue les variables libres (formalisées par des *paramètres*) des variables liées (formalisées par des *variables*). Les descriptions sémantiques renomment judicieusement les variables d'un terme en paramètres avant tout traitement du terme, pour effectuer la substitution inverse à la fin du traitement. Ainsi, il ne peut pas y avoir de phénomène de capture de variable lors d'une substitution. (L'introduction d'une variable déjà liée dans un terme nécessite normalement son renommage préalable pour éviter toute capture.) James McKinna et Robert Pollack ont ainsi formalisé Lego en Lego (Lego et Coq sont des systèmes très voisins). Leur approche nous semble être aujourd'hui la meilleure pour décrire un langage fonctionnel, ou dont le noyau est fonctionnel, contenant ou non des primitives inspirées du π-calcul, dans l'attente d'un système aussi puissant que Coq permettant la récursion sur des termes de types fonctionnels.

Enfin, justement, il existe une méthode qui permet d'utiliser directement les fonctions du système choisi (Coq par exemple) pour formaliser les notions de variables liées et de substitution du langage à décrire. On appelle ce type de description une spécification d'ordre supérieur. L'idée de base de la méthode est de représenter les variables du langage spécifié (le langage dit objet) par les variables du langage utilisé pour spécifier (le langage dit méta). Ainsi, un constructeur d'un langage L sera défini par des déclarations du type : $lam : (L \Rightarrow L) \Rightarrow L$. Le fils d'un nœud lam est ici une fonction (méta).

Cette technique permet également de formaliser les *side conditions* (variable libre dans une hypothèse) et les changements de contexte (déchargement d'une hypothèse en déduction naturelle) dans les règles d'inférence. Elle fournit souvent la description la plus concise et la plus élégante d'un langage. De plus, par voie de conséquence, les preuves sont aussi grandement facilitées. L'utilisateur peut se concentrer sur les difficultés de son langage, sans devoir réimplémenter les fonctions, que le système a déjà implémentées pour lui.

Par ailleurs, la définition d'un type inductif fournit des schémas de récurrence et des principes d'induction qui permettent de faire des preuves générales sur des données de ce type. En particulier, si l'on représente la syntaxe d'un langage par un type inductif, on peut utiliser les principes associés pour faire des preuves sur la sémantique de ce langage.

Le problème est qu'une syntaxe abstraite d'ordre supérieur, définie de manière naïve, ne forme pas un type inductif. En collaboration avec d'autres chercheurs (notamment Frank Pfenning à CMU), nous travaillons depuis 1992 sur plusieurs solutions à ce problème, qui était ouvert depuis 1987. La solution à long terme passe par la conception d'une nouvelle théorie typée (voir section 3.1.3). La solution à court terme consiste à inventer de nouvelles méthodes de description des langages dans les systèmes existants. Nous avons proposé deux solutions pour cela, dans le système Coq.

En collaboration avec Amy Felty (ATT Bell Labs), nous avons défini une nouvelle notion : la *syntaxe abstraite d'ordre supérieur restreint* qui permet d'utiliser à la fois la syntaxe d'ordre supérieur et les types inductifs. Cette approche est basée sur une idée très simple : définir les fonctions du langage sur un ensemble de variables prédéfini, ce qui conduit à des définitions de constructeurs d'un langage L du type : $lam : (var \Rightarrow L) \Rightarrow L$. La syntaxe étant trop permissive, il est nécessaire de définir un prédicat décrivant les termes légaux sur lesquels on travaille. Il faut aussi spécifier une opération de substitution pour chaque langage étudié. Mais cette substitution est très facile à définir et peut utiliser l'application (méta) du système utilisé.

Dans la seconde approche, appelée *sémantique d'ordre supérieur* [1], l'utilisateur n'a pas besoin de décrire une opération de substitution ; il peut utiliser directement l'application du système choisi. L'idée maîtresse de cette méthode, qui sera reprise dans la définition d'une nouvelle théorie typée (3.1.3), est de considérer un terme ouvert, dépendant d'une liste de variables, comme une fonction de cette liste de variables. Un prédicat, nommé *valide*, différent du prédicat utilisé dans l'approche précédente, restreint l'ensemble des termes décrits aux arbres de syntaxe abstraite. Les sémantiques sont données sur les termes valides, qui sont donc des termes fonctionnels. D'où le nom de sémantique d'ordre supérieur pour cette méthode qui prolonge à la sémantique l'idée de la syntaxe abstraite d'ordre supérieur.

Pour conclure sur ces différentes méthodes de description des liaisons dans les langages de programmation, une seule méthode ne nous intéresse pas du tout : les indices de Bruijn. La technique introduite par Andrew Gordon semble permettre des spécifications relativement proches de ce que l'on écrit sur le papier, après une étape préliminaire de preuves concernant les codes de Bruijn. Comme nous l'avons dit plus haut, la méthode due à James Mc Kinna et Randy Pollack nous semble aujourd'hui la meilleure pour décrire un langage fonctionnel en l'absence d'un système permettant l'utilisation conjointe de la syntaxe abstraite d'ordre supérieur et de l'induction. La technique de spécification d'ordre supérieur est la plus abstraite, la plus élégante ; c'est celle que nous préférerons toujours chaque fois que ce sera possible ; la version restreinte de cette méthode est compatible avec l'induction. Cette technique ne nous semble pas encore avoir fait ses preuves pour les langages impératifs, bien qu'il existe plusieurs expériences dans ces langages. Cependant la recherche est encore active sur ce sujet, que nous avons inclu dans nos objectifs de recherche à court terme. La notion de sémantique d'ordre supérieur que nous avons proposée représente toujours une possibilité intéressante d'utilisation de la syntaxe abstraite d'ordre supérieur dans le système Coq actuel, peut-être la plus intéressante parmi les trois ou quatre méthodes proposées à ce jour. Un excellent défi pour cette méthode semble être

la preuve du théorème de Church-Rosser pour le lambda-calcul simple. Nous avons commencé à implémenter cette preuve dans Coq.

***Logical Frameworks* et systèmes** En ce qui concerne les *Logical Frameworks* et les systèmes, nous pensons utiliser principalement le système Coq, mais pas seulement. Il existe beaucoup d'autres systèmes: Alf, PVS, Twelf, HOL, Isabelle, etc. Nous suivrons attentivement l'évolution du système Alf, très proche de Coq. PVS est un concurrent américain sérieux de Coq ; il serait surement utile d'acquérir un minimum d'expérience dans ce système. Twelf nous intéresse plus particulièrement parce qu'il est le système actuellement le mieux placé pour l'utilisation conjointe de la syntaxe abstraite d'ordre supérieur et de l'induction. HOL est plus connu (et intensivement utilisé) dans le domaine du *hardware* que nous ne comptons pas aborder pour l'instant. Isabelle a beaucoup d'atouts, et en particuler plus d'automatisme (réécriture utilisant l'unification d'ordre supérieur) que Coq, mais ce système n'a pas le langage et les fondements puissants de Coq. Plusieurs approches pour incorporer la réécriture dans Coq sont actuellement en cours de développement.

3.1.3 Théories typées

Participant : Joëlle Despeyroux.

Dans un premier pas vers la définition d'un système de type qui incorpore la méthodologie du système LF (Logical Framework développé à Edimbourg) dans des systèmes comme Coq ou Alf et en collaboration avec Frank Pfenning et Carsten Schürmann (Carnegie Mellon University), nous avons proposé récemment [3] un système préservant l'adéquation des représentations utilisant la syntaxe abstraite d'ordre supérieur. Ce système est un λ-calcul modal étendu avec des opérateurs de raisonnement par cas et d'itération, permettant la récursion sur des termes de type fonctionnel.

Dans son travail de thèse, Pierre Leleu a proposé [2] une variante de ce système, meilleure sur plusieurs points. D'une part, il a changé le noyau modal en reprenant celui de Frank Pfenning et Hao-Chi Wong (1995), plus agréable pour l'utilisateur. D'autre part, il a remplacé les règles d'évaluation du système initial par des règles de réduction, auxquelles il a ajouté l'êta-expansion. Le système a toutes les propriétés souhaitées : préservation du typage par la réduction, confluence, normalisation forte de la réduction, et extension conservative par rapport au λ-calcul simplement typé. Pierre Leleu a aussi proposé une extension partielle de son système aux produits dépendants ; dans une présentation plus proche de celle du Calcul des Constructions Inductives, sans types mutuellement récursifs et avec seulement l'élimination non-dépendante pour le moment.

Ce sujet de recherche très difficile est très prometteur. Malheureusement, nous manquons de moyen en hommes pour continuer à y travailler, du moins pour l'instant. Nous sommes ravis de voir de nouvelles contributions apparaître dans des conférences prestigieuses comme LICS'99. En particulier, le travail de Martin Hofmann, réalisé au niveau des catégories, devrait permettre d'étendre notre système à un système généralisant le Calcul des Constructions Inductives.

3.2 Preuve de correction de programmes

Participants : Joëlle Despeyroux, André Hirschowitz.

Nous n'avons abordé ce domaine que très récemment. La demande industrielle est ici très forte, dans différents secteurs plus ou moins sensibles.

Dans un premier temps, afin d'acquérir une connaissance du domaine, nous avons commencé à étudier plusieurs exemples, dans des domaines différents. Dans un second temps, fort de cette expérience, nous espérons pouvoir dégager de nouvelles méthodes ou/et améliorer l'existant.

Le domaine est relativement nouveau pour nous. Cela dit, ce domaine est connexe au précédent, et de plus, nous nous proposons de l'aborder avec le même outil que précédemment : le système Coq.

3.2.1 Étude d'exemples

Participants : Joëlle Despeyroux, André Hirschowitz.

La preuve de correction de *programmes impératifs* est théoriquement faisable dans le système Coq depuis que Jean-Christophe Filliâtre a montré comment interpréter des preuves en logique de Hoare dans ce système. Nous avons expérimenté cette nouvelle possibilité cette année (5.1.3).

Le *glaneur de cellule-mémoire (gc)* est l'un des outils liés à un langage. La preuve de sa correction est cruciale, et peut s'avérer très difficile, particulièrement dans le cas d'une architecture distribuée.

Le *cache* est une partie importante du système d'exploitation d'une machine. La preuve de sa correction est aussi cruciale, particulièrement avec l'avènement du *co-design* (conception en parallèle du hardware et d'une partie du software).

Plusieurs équipes de recherche, dans le monde académique comme dans l'industrie, s'intéressent au système Coq pour formaliser des preuves de correction de *protocoles de communication*. Notons en particulier l'équipe Dyade de Bull (action VIP), autour de Dominique Bolignano et le CNET.

Notons que la plupart des preuves de correction de programmes se font sur une spécification formelle abstraite (une abstraction) de ce programme et non sur le programme lui-même. C'est notamment le cas pour les preuves de correction de protocoles.

3.2.2 Méthodes

Participant : Joëlle Despeyroux.

Le plus souvent, on ne prouve pas la correction du programme lui-même, mais d'une description formelle de ce programme dans différents formalismes (logique temporelle, Unity, etc). La difficulté principale étant alors de trouver le bon niveau d'abstraction de la description, tout en s'assurant (informellement) que les hypothèses faites sont réalisables ! La seule approche qui ne suit pas ce schéma est l'extraction de programme.

L'*extraction de programme* proposée dans le système Coq consiste en fait à développer en même temps un programme et sa spécification — qui dans un système basé sur la théorie

des types, comme Coq, se trouve être justement une preuve de sa correction. Nous fondons beaucoup d'espoir sur cette technique, encore en développement, qui est l'une des originalité et l'une des grandes forces du système Coq.

Les preuves de correction de protocoles de communication sont souvent réalisées dans un système appelé *Unity*. Or le langage Unity a été récemment partiellement décrit sous le système Coq par Pierre Crégut et Barbara Heyd. Il est donc possible d'aborder les preuves de correction de protocoles dans Coq en utilisant Unity. Nous sommes très curieux d'essayer cette approche.

La *Logique Temporelle (TLA)* est beaucoup utilisée pour formaliser les preuves de correction de protocoles, entre autres domaines. Damien Doligez et Georges Gonthier l'ont notamment utilisée pour formaliser la preuve de correction d'un glaneur de cellule-mémoire pour une architecture multi-processeurs. C'est une approche que nous devrons étudier, et essayer par nous-mêmes.

La méthode B développée par J-R. Abrial dans les années 80, et toujours en développement, est encore l'une des méthodes les plus utilisées dans le monde industriel. Elle a notamment été utilisée, avec succès, pour spécifier le métro parisien. Cette méthode n'est pas aussi formelle que les autres approches dont nous avons parlé. Cependant, il existe plusieurs formalisations de cette méthode, dans différents systèmes, avec différents buts : prouver des propriétés méta-théoriques de la méthode, ou permettre la preuve des obligations de preuves générées par le système. L'une des plus récentes propositions, réalisée pour les systèmes Coq et PVS, permet les deux. C'est une proposition a priori très intéressante, que nous espérons pouvoir étudier. Cela dit, cette étude ne fait pas partie de nos objectifs à court terme.

La technique appellée *proof-carrying code*, présentée à la conférence POPL en 97, permet à un site hôte de vérifier qu'un code produit par un client par hypothèse non fiable vérifie certaines propriétés. Les applications sont nombreuses, et comprennent typiquement le code mobile, mais aussi, par exemple, les compilateurs d'un langage (fiable) de haut niveau pouvant importer du code (non fiable) d'un langage de bas niveau. L'idée clé est que l'implémenteur du code doit fournir une preuve attestant le fait que le code respecte une certaine politique de securité definie par le site hôte. La preuve de correction doit pouvoir être réalisée avec un outil simple et efficace. L'idée est excellente et est actuellement intensivement utilisée pour le compilateur de Standard ML dans l'équipe de Peter Lee à CMU. Il nous faudra l'expérimenter par nous-mêmes.

L'*interprétation abstraite* n'est pas une technique de preuve, mais une technique d'analyse de programme, qui permet de trouver des *bugs*, mais est surtout utilisée pour optimiser du code, typiquement un compilateur. L'analyse porte sur les variables d'un programme: leur point de définition, leurs points d'appel, et les conditions aux bords des variables (dépassement des bornes d'un tableau par exemple). C'est une technique très utilisée que nous devrons apprendre à maîtriser.

3.3 La part de rêve, ou veille technologique: mobilité, réseaux

Participant : Joëlle Despeyroux.

Cette section est hautement prospective... Mais n'est-ce pas là la vocation première de la recherche ?

Une famille de langages de programmation est en train de voir le jour, pour décrire deux phénomènes liés aux réseaux : d'une part la mobilité (processus mobiles), d'autre part la description du réseau lui-même (notion d'ambients de Luca Cardelli). Pour l'instant, il ne s'agit pour nous que de suivre l'évolution de ces recherches : ces notions sont en train de naître ; on ne dispose jusqu'ici que de différentes propositions de modèles, au mieux de calculs, rarement de langages. Toutes les propositions de calcul mobile sont plus ou moins basées sur le π-calcul (join-calcul, calcul bleu, ...) et allient avec plus ou moins de bonheur les notions de concurrence et d'objets. Elles sont relativement difficiles à comparer. La phase de formalisation et de preuve (particulièrement de preuve de correction de programmes – protocoles) n'est donc pas entamée. Cela dit, il existe déjà une communauté internationale très active sur ce sujet : projet ESPRIT CONFER comprenant notamment les projets Meije et Para à l'INRIA, Benjamin Pierce à l'université de Penn et le laboratoire de recherche de Microsoft avec Luca Cardelli et Andrew Gordon. Ce domaine de recherche est probablement l'un des domaines les plus novateurs, et prometteurs, du moment. De plus, le besoin industriel est évident. Il se pourrait que ce qui semble hautement prospectif aujourd'hui soit déjà une réalité demain, c'est à dire dans 2 ou 3 ans...

En tous les cas, ces langages nous offrent l'une des applications les plus intéressantes à plus d'un titre: d'une part ils représentent des défis certains pour nous (nous avons déjà commencé à formaliser le π-calcul), d'autre part ils nous paraissent très prometteurs. D'où une motivation double.

4 Domaines d'applications

4.1 Panorama

Mots clés : énergie nucléaire, internet, santé, télécommunications, transports.

Nos compétences sur les spécifications formelles et les preuves intéressent l'ensemble des domaines industriels où la présence d'erreurs même minimes dans les programmes peut avoir des conséquences graves : danger pour la vie humaine comme dans l'énergie nucléaire ou les transports (avions, automobiles), la chirurgie assistée par ordinateur, ou simplement coût prohibitif comme dans le cas du commerce électronique et des réseaux de télécommunication, ou du spatial.

4.2 Logiciel embarqué

Mots clés : sûreté de programmation, vérification de logiciel, transports, télécommunications.

Résumé : *Les éléments logiciels présents dans les appareils modernes ont une importance cruciale pour le bon fonctionnement de ces appareils. Dans de nombreux cas, il n'est plus possible de recourir uniquement à des campagnes de tests pour assurer leur correction et des techniques de programmation certifiée doivent être développées et mises à la portée des ingénieurs. Notre travail sur les spécifications formelles et les preuves contribue à cet effort.*

Les appareils que nous utilisons dans notre vie quotidienne contiennent une quantité croissante de logiciel. Ce logiciel contribue à la compétitivité de ces appareils en permettant des utilisations plus simples du point de vue de l'utilisateur, mais souvent plus complexes en réalité.

Jusqu'à maintenant, les erreurs de programmation étaient une caractéristique admise et le coût de telles erreurs était réduit par la possibilité de faire intervenir un opérateur humain pour diagnostiquer les pannes et corriger les erreurs manuellement. Lorsque le logiciel est embarqué, ceci n'est plus possible. Le coût d'une erreur pour l'entreprise qui utilise ce logiciel devient beaucoup plus important: rappel de voitures déjà vendues, fusées ou sondes spatiales dont les missions échouent, accidents de véhicules de transport.

De nombreuses équipes de recherche mettent au point des techniques qui permettent de développer des logiciels validés vis-à-vis de spécifications, ce qui permettrait théoriquement de réduire le risque d'erreur à zéro (en pratique des erreurs peuvent également se glisser dans les spécifications). Ces techniques sont plus ou moins coûteuses et plus ou moins générales. Avec le logiciel Coq, fondé sur une théorie typée d'un grand pouvoir d'expression, on se trouve à un bout du spectre : on peut aborder des problèmes très variés et complexes mais avec un coût de développement très important [1]. Les recherches que nous faisons contribuent à diminuer ce coût, d'une part par la recherche de théories typées permettant des spécifications plus concises, d'autre part par le développement de bibliothèques d'exemples. Nos recherches dans ce domaine sont le lieu de collaborations avec un constructeur aéronautique.

4.3 Programmation sur internet

Mots clés : internet, sûreté de programmation, télécommunications, web.

Résumé : *Le langage qui semble s'imposer pour la programmation sur le réseau internet est le langage* Java. *Cependant, une activité de recherche intense s'organise autour des langages mobiles, dédiés à ce type de programmation et devant permettre des développements beaucoup plus fiables. Nous suivons cet effort, en formalisant ces langages sur ordinateur, afin d'être prêts lorsque lesdits langages le seront.*

Ici le besoin de sûreté est d'un type bien particulier, puisqu'il ne s'agit plus de faire fonctionner des programmes dans un environnement naturel qui présente aléatoirement des situations difficiles, mais de les rendre résistants à des attaques systématiques. Certains langages comportent d'ailleurs des primitives codifiant le type d'attaque envisagé : attaque de la couche externe (le logiciel d'exploitation) ou interne (le programme lui-même).

Le langage Java s'est pour l'instant imposé comme le langage de programmation idéal du domaine, faute de mieux, au moins connu des industriels. (Les langages de la famille ML permettent assurément une programmation plus fiable mais ne sont malheureusement connus que dans le monde académique). Les langages basés sur la mobilité, ayant les primitives requises pour la programmation sûre des réseaux, devraient logiquement s'imposer un jour prochain. Espérons, pour la sécurité des transactions sur l'internet et la nôtre, qu'ils sauront le faire. C'est dans cet espoir que nous étudions la formalisation de ces langages en machine.

1. Les techniques de *Model Checking* sont à l'autre bout du spectre.

Par ailleurs, les théories typées, et certaines techniques les utilisant, comme le *proof-carrying code*, ont prouvé leur utilité dans le domaine du développement de programmes sûrs pour l'internet. De nombreuses équipes de recherche, en Europe et aux États-Unis (CMU, par exemple), ont acquis une expérience reconnue dans ce domaine.

5 Résultats nouveaux

5.1 Resultats

5.1.1 Formalisation de langages

Guillaume Gillard a poursuivi sa formalisation d'un calcul concurrent et à objet en Coq, en utilisant une méthode de description des variables due à Andrew Gordon. Cette méthode est intermédaire entre la technique des codes de Bruijn et la syntaxe d'ordre supérieur. Elle conduit à des spécifications et des preuves relativement proches des développements sur papier et bien plus concis et clairs que la technique des codes de Bruijn ne peut le permettre.

Joëlle Despeyroux et André Hirschowitz, en collaboration avec Stefan Berghofer de l'université de Munich, ont repris la preuve du théorème de Church-Rosser pour le λ-calcul simple en utilisant la méthode de description des variables présentée en 94. Stefan Berghofer a réalisé une preuve de ce théorème dans le système Isabelle en suivant la même approche. Il semble que l'on puisse admettre un axiome d'extensionalité plus fort que ce qui avait été choisi dans les précédentes tentatives. La preuve dans le système Coq devrait fournir un élément de comparaison interessant entre les deux systèmes.

5.1.2 Theories typées

Joëlle Despeyroux a présenté son travail commun avec Pierre Leleu sur la récursion primitive pour la syntaxe d'ordre supérieur avec types dépendants au séminaire sur les logiques modales intuitionistes affilié à la conférence internationale FLoC'99, en Italie en juillet 99. Un article présentant ce travail a été soumis à un journal.

5.1.3 Programmation certifiée

Pierre Corbineau, dans un stage de 1ère année de l'ENS Ulm, a réalisé une preuve de correction d'un programme impératif en Coq. Le programme, un algorithme de contrôle de n ascenseurs, est l'un des exemples pour lesquels la méthode B est particulièrement bien adaptée. Nous avons utilisé ici une méthode récemment implémentée dans Coq par Jean-Christophe Filliatre, de l'équipe Coq à Rocquencourt. L'expérience a été enrichisssante pour les deux équipes.

Etienne Lozes, dans un stage de 1ère année de l'ENS Lyon, a réalisé plusieurs certifications d'un nouvel algorithme d'inférence de types en Coq, en utilisant différents codages des variables. Cette expérience a été très enrichissante pour nous. Nous espérons étendre deux des approches testées à des langages plus conséquents, en particulier au calcul concurrent et à objet étudié par Guillaume Gillard.

6 Contrats industriels (nationaux, européens et internationaux)

6.1 Action de développement Génie

Participant : Joëlle Despeyroux.

Dans le cadre de l'action Génie phase 2 entre l'INRIA et Dassault-Aviation, nous sommes intervenus comme expert extérieur dans la démonstration de correction d'un compilateur du langage Scade, dérivé de Lustre, démonstration réalisée par Eduardo Gimenez à Dassault.

7 Actions régionales, nationales et internationales

7.1 Action de Recherche Coopérative XXX

Une proposition d'action de recherche coopérative « vers la certification du langage Objective Caml », en collaboration avec les équipes Cristal et Coq de l'INRIA et le CNET, a été soumise à la Direction Scientifique en novembre 1999. Réponse attendue courant décembre.

7.2 Groupe de travail européen Types

Joëlle Despeyroux est responsable locale (*site leader*) du groupe de travail européen ESPRIT « Types », qui fait suite au projet européen ESPRIT BRA « Types pour les preuves et les programmes ». Ce groupe de travail a commencé le 15 septembre 1996, et s'est terminé cette année, le 14 septembre 1999. Il comprenait 15 sites répartis en Europe (Finlande, France, Allemagne, Grande Bretagne, Italie, Pays-Bas et Suède). L'adresse Web de la page d'accueil de ce groupe est : `http://www.cs.chalmers.se/ComputingScience/Research/Logic/TypesWG/`. Nous avons soumis une proposition de continuation de ce groupe de travail à la communauté européenne fin novembre 99. Cette proposition réunit sensiblement les mêmes sites académiques que précedemment, en y adjoignant des sites industriels, notamment, en France, le CNET, Dassault-Aviation et Trusted Logic.

8 Diffusion de résultats

8.1 Animation de la communauté scientifique

Joëlle Despeyroux était responsable scientifique de l'école d'été Types qui a eu lieu en septembre 1999. (voir `http://www-sop.inria.fr/types-project/types-sum-school.html`).

Joëlle Despeyroux a fait partie du comité de programme du séminaire Logical Frameworks and Meta-languages (LFM'99), affilié à la conférence PLI'99 qui a eu lieu en septembre 99 à Paris. Elle sera *chairman* du comité de programme de LFM'2000, prévu pour juin 2000, affilié à la conférence LICS'2000.

Des membres du projet ont effectué des revues d'articles pour diverses conférences dont LICS'99.

8.2 Enseignement universitaire

Joëlle Despeyroux est responsable locale à Sophia Antipolis du DEA MDFI (Mathématiques Discrètes et Fondements de l'Informatique) de Marseille depuis octobre 1998 (10h de cours en 1998-1999).

Joëlle Despeyroux a proposé un cours de tronc commun sur les « méthodes formelles et la fiabilité du logiciel » au DEA d'informatique à l'université de Nice pour le renouvellement de l'habilitation de ce DEA. Un module optionnel sur la « mécanisation des preuves », enseigné en collaboration avec Emmanuel Kounalis, sera assuré dès cette année, au second semestre.

Guillaume Gillard a donné des TP à un IUT dépendant de l'université de Nice (60 heures).

8.3 Thèses et stages

8.3.1 Thèses en cours dans le projet

1. Guillaume Gillard, « Vers la spécification et les preuves sur les langages concurrents et à objets », université de Paris VII, depuis octobre 1997, dirigée par Joëlle Despeyroux.

8.3.2 Stages effectués dans le projet

1. Pierre Corbineau, « Preuves de correction de programme impératif en Coq », stage 1ère année ENS Ulm.
2. Etienne Lozes, « Un algorithme d'inférence de type simple certifié en Coq », stage 1ère année ENS Lyon.

8.4 Participation à des colloques, séminaires et invitations

Deux membres de l'équipe ont participé au séminaire annuel du groupe de travail européen Types à Lokeberg en Sweden en juin 99. Guillaume Gillard y a exposé ses travaux de thèse.

Joëlle Despeyroux a participé à la conférence FLOC'99 en juillet en Italie, et en particulier au séminaire sur les logiques modales intuitionistes et leurs applications (IMLA'99) où elle a présenté ses travaux sur la récursion primitive pour les syntaxes fonctionnelles, avec types dépendants.

9 Bibliographie

Ouvrages et articles de référence de l'équipe

[1] J. Despeyroux, A. Hirschowitz, « Higher-Order Syntax and Induction in Coq », *in : Proceedings of the fifth Int. Conf. on Logic Programming and Automated Reasoning (LPAR 94)*, F. Pfenning (éditeur), *822*, Springer-Verlag LNAI, p. 159–173, July 1994. Also appears as INRIA Research Report RR-2292 (June 1994).

[2] J. Despeyroux, P. Leleu, « A modal λ-calcul with iteration and case constructs », *in : Proc. of the annual Types for Proofs and Programs seminar, Springer-Verlag LNCS*, mars 1998.

[3] J. Despeyroux, F. Pfenning, C. Schürmann, « Primitive Recursion for Higher-Order Abstract Syntax », *in : Proceedings of the TLCA 97 Int. Conference on Typed Lambda Calculi and*

Applications, Nancy, France, April 2–4, P. de Groote, J. R. Hindley (éditeurs), Springer-Verlag LNCS 1210, p. 147–163, April 1997. An extended version is available as CMU Technical Report CMU-CS-96-172.

[4] J. Despeyroux, « Proof of translation in Natural Semantics », *in : Proceedings of the first Symp. on Logic In Computer Science, LICS'86*, IEEE Computer Society, June 1986. Also appears as INRIA Research Report RR-514, April 1986.

[5] J. Despeyroux, « Theo: an interactive proof development system », *The Scandinavian Journal on Computer Science and Numerical Analysis (BIT), special issue on 'Programming Logic' 32*, 1992, p. 15–29, A preliminary version appears as INRIA Research Report no. 887, August 1988.

[6] J. Despeyroux, « Sémantique Naturelle: Spécifications et Preuves », *Research Report n° RR-3359*, INRIA, février 1998, Lecture notes for a graduate course in the "DEA" "Mathématiques Discrètes et Fondements de l'Informatique" (MDFI), Marseille, 1995-1999 (80 pages, in french), `http://www.inria.fr/RRRT/RR-3359.html`.

[7] D. Terrasse, « Encoding Natural Semantics in Coq », *in : Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, *Springer-Verlag LNCS*, *936*, p. 230–244, July 1995.

Articles et chapitres de livre

[8] J. Despeyroux, P. Leleu, « Metatheoretic results for a modal λ-calcul », *the Journal of Functional and Logic Programming*, 1999, to appear (à paraître).

[9] J. Despeyroux, F. Pfenning, C. Schürmann, « Towards Primitive Recursion for Higher-Order Abstract Syntax », *Theoretical Computer Science*, 1999, Extended and revised version of the TLCA'97 paper, to appear (à paraître).