

## Coq, un outil pour l'enseignement

### Une expérience avec les étudiants du DESS Développement de logiciels sûrs

David Delahaye\* — Mathieu Jaume\*\* — Virgile Prevosto\*\*\*

\* CPR – CEDRIC – CNAM  
292, rue St Martin, F-75141 Paris Cedex 03, France  
David.Delahaye@cnam.fr

\*\* SPI – LIP6 – Université Pierre et Marie Curie  
8 Rue du Capitaine Scott, F-75015 Paris, France  
Mathieu.Jaume@lip6.fr

\*\*\* Max-Planck Institut für Informatik  
Stuhlsatzenhausweg 85, D-66123 Saarbrücken, Allemagne  
INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France  
prevosto@mpi-inf.mpg.de, virgile.prevosto@inria.fr

---

*RÉSUMÉ.* Cet article présente l'emploi de l'outil d'aide à la preuve Coq auprès d'étudiants de DESS (3<sup>e</sup> cycle universitaire). D'abord, dans le cadre d'un cours de sémantique des langages, Coq facilite l'appropriation par les étudiants de notions souvent jugées abstraites en leur permettant de les relier à des termes plus concrets. Ensuite, un projet informatique utilise Coq pour traiter des problèmes de plus grande envergure, faisant apparaître par là-même Coq comme un véritable outil de génie logiciel. Enfin, la réalisation de preuves dans l'atelier Focal a permis de fructueuses interactions avec le développement de ce système.

*ABSTRACT.* In this article, we present the use of the Coq proof assistant with DESS (Master thesis) students. First, in the framework of a course of programming language semantics, Coq greatly helps the students to understand formal and abstract notions, such as induction, by binding them to more concrete terms. Next, a computer science project shows that Coq is also appropriate when dealing with larger problems. Last, we show how proofs developed by means of the Focal toolbox made it possible to get very valuable hints on the development of that system.

*MOTS-CLÉS:* enseignement, sémantique, conception formelle, Coq, Focal.

*KEYWORDS:* teaching, semantics of programming languages, formal conception, Coq, Focal.

---

## 1. Introduction

Dans cet article, nous présentons comment l'assistant à la preuve Coq a été utilisé, durant ces trois dernières années (de 2001 à 2004), comme outil d'enseignement auprès des étudiants du DESS DLS "Développement de Logiciels Sûrs" du CNAM et de l'université Pierre et Marie Curie. Le DESS DLS forme chaque année environ 25 étudiants aux métiers de la sûreté et de la sécurité des systèmes à logiciel prépondérant. Outre une formation classique (sémantique, analyse statique, temps réel, concurrence...), il vise à donner une formation aux méthodes formelles demandées dans les niveaux élevés de certification (EAL 5-7, SIL 3-4) et aux outils associés : atelier B, Coq, outils de "model-checking", Lustre/Esterel... Le public est principalement issu de maîtrise d'informatique.

Les principales motivations pour l'utilisation de Coq dans nos activités d'enseignement résident dans les difficultés (et le manque de motivation !) qu'éprouvent certains étudiants en informatique vis à vis d'exercices de nature formelle. De plus, beaucoup d'étudiants ont l'habitude d'assimiler les enseignements au travers de travaux pratiques et peu d'entre eux ont déjà une autonomie suffisante pour la lecture de textes à caractère théorique. Le choix de Coq n'est pas primordial, les développements présentés ici auraient sans doute pu être réalisés avec un autre système (pour une comparaison entre différents systèmes, on pourra, par exemple, consulter (Jakubiec *et al.*, 1997; Griffioen *et al.*, 1998)).

Coq (Coq Development Team, 2002) est un outil d'aide à la preuve ; il est actuellement développé par le projet LogiCal à l'INRIA et au LRI (université d'Orsay). Il est basé sur la théorie des types (le calcul des constructions inductives pour être plus précis) et utilise une logique de base intuitionniste. Les spécifications, les définitions, les théorèmes et les preuves formelles peuvent être réalisés de manière interactive et il est possible d'en extraire des programmes (isomorphisme de Curry-Howard). Les principales constructions de Coq utilisées en cours sont les types dépendants et les types inductifs. Ces derniers ressemblent aux types somme des langages fonctionnels, mais Coq engendre en outre automatiquement les principes d'induction correspondants. Par exemple, on peut définir les entiers de Peano à l'aide d'un type à deux constructeurs :

```
Inductive peano_nat : Set :=
  zero: peano_nat | succ: peano_nat -> peano_nat.
```

Coq définit alors un terme `peano_nat_ind`, de type

```
forall P:peano_nat->Prop,
  P zero ->
  (forall n:peano_nat, P n -> P (succ n)) ->
  forall n: peano_nat, P n
```

Ce terme indique que si un prédicat  $P$  est vérifié par `zero` et si pour tout naturel  $n$  vérifiant  $P$ , `succ(n)` vérifie  $P$ , alors  $P(n)$  est vrai pour tout  $n$ .

Par ailleurs,  $P$  illustre la puissance des types dépendants.  $P$  définit en fait une famille de types paramétrée par les termes de `peano_nat`. Ainsi on peut distinguer les termes (c'est à dire les preuves) de type  $P(\text{zero})$ ,  $P(\text{succ zero})$ , etc.

L'emploi de Coq est décliné dans trois contextes différents.

Coq est tout d'abord utilisé dans le cadre d'un cours d'introduction à la sémantique des langages de programmation. Ce cours est souvent jugé difficile par les étudiants du fait de l'utilisation d'un formalisme mathématique permettant d'énoncer et de prouver des propriétés. Nous avons donc fait implanter dans Coq par les étudiants la quasi-totalité du cours de sémantique. Coq s'est révélé être un excellent vecteur de compréhension : implanter les notions étudiées permet de se les approprier en leur donnant forme et en pouvant les expérimenter. La section 2 présente les grandes lignes de ce cours de sémantique.

Parallèlement, Coq a été, pour la première fois cette année (2003/2004), utilisé dans le cadre d'un projet. D'habitude, le projet est plutôt orienté programmation avec notamment l'utilisation d'OCaml (OCaml Development Team, 2004) (qui est présenté aux étudiants brièvement et de manière essentiellement pratique en début d'année sous la forme de cours de mise à niveau), mais même si elle se révèle plus subtile, l'activité de preuve n'est pas fondamentalement différente de l'activité de programmation (au moins dans le système Coq où l'isomorphisme de Curry-Howard ne permet même plus de distinguer les deux activités). L'objectif de ce projet est de confronter les étudiants à une application réelle (mais d'envergure raisonnable de manière à ne pas trop sous-spécifier le problème) et non plus à des "exemples d'école" comme c'est volontairement le cas dans les quelques cours d'introduction à Coq dont les étudiants ont bénéficié en début d'année (afin qu'ils puissent, en particulier, commencer les preuves du cours de sémantique). Pour les étudiants, il est important de retirer de cet exemple réel la distinction entre raisonnement rigoureux et raisonnement formel. Notamment, il s'agit de bien comprendre le fossé existant entre une preuve mathématique (plus ou moins abstraite) et une preuve formelle (concrète et détaillée). Cet effort de concrétisation aboutit généralement à des questions finalement très nouvelles pour ces étudiants "contraints", par exemple, d'écrire des programmes qui terminent ou de choisir une représentation appropriée pour faciliter les preuves de correction. Le déroulement de ce projet est décrit en détail dans la section 3.

Dans le cadre du cours de sémantique et du projet, les preuves demandées sont parfois longues et requièrent de temps à autre des récurrences un peu difficiles. En revanche, l'environnement dans lequel elles sont développées est complètement explicite : les définitions et les propriétés s'ajoutent de manière incrémentale et toutes les preuves peuvent s'appuyer sur les résultats précédemment établis. Prouver des propriétés dans cette architecture "linéaire" correspond à une situation "idéale". Le contexte de développement est parfois plus complexe. C'est notamment le cas lorsque l'on souhaite établir des propriétés dans le cadre d'une architecture hiérarchique (approche objet, approche par raffinements...). Afin de sensibiliser les étudiants à ces problèmes, nous avons encadré des projets (associés au cours de conception formelle) consistant à utiliser l'atelier Focal (Prevosto *et al.*, 2002a; Prevosto *et al.*, 2002b; Focal development team, 2004; Prevosto, 2003). Il s'agit d'un atelier intégré de conception modulaire de logiciels certifiés. Chaque module de Focal peut être constitué d'un ensemble de déclarations, définitions, énoncés et preuves. On peut passer d'une spécifi-

cation abstraite (ne contenant que des déclarations et des énoncés) à une implantation concrète (où tout est défini et prouvé), à l'aide de mécanismes d'héritage et d'instanciation de paramètres. Lors d'une étape d'héritage, des déclarations peuvent être raffinées en définitions et des énoncés recevoir une preuve. En outre, une fonction déjà définie peut recevoir une nouvelle définition, plus efficace dans le contexte du nouveau module. Le compilateur Focal effectue un certain nombre d'analyses garantissant la cohérence de ces opérations d'héritage (Prevosto *et al.*, 2002a). En plus du code OCaml et de la documentation automatique, la compilation d'un programme Focal produit du code Coq dans lequel des preuves peuvent être développées. Le code Coq engendré explicite alors complètement les environnements de définitions et d'hypothèses dans lesquels les preuves peuvent être établies. Ces environnements sont calculés à la suite des analyses menées par le compilateur. Le but des projets est de présenter la sémantique des constructions d'une architecture hiérarchique (héritages, redéfinitions, dépendances...) en montrant comment une telle architecture peut être "dépliée". Ici le compilateur Focal permet, via la génération de structures Coq, d'expérimenter une telle sémantique. La section 4 présente l'utilisation de Coq et Focal dans le cadre de ce cours.

## 2. Implantation d'un cours de sémantique

Trop souvent encore, l'enseignement des notions de base de sémantique des langages de programmation est perçu par les étudiants en informatique comme un exercice de style formel difficile, laborieux et peu attrayant. Plus encore, l'utilisation d'un formalisme mathématique qui fait largement appel à des notations exprimées à partir de symboles "exotiques" est à la source d'une véritable appréhension de certains étudiants. L'alphabet grec fait peur ! Pourtant, à partir d'un certain niveau, les étudiants en informatique sont familiers avec les objets étudiés en sémantique : tous ont une idée (plus ou moins précise) de ce qu'est un langage de programmation, un arbre de syntaxe abstraite, ou un environnement d'exécution, et connaissent au moins de manière informelle les mécanismes d'exécution d'un programme. L'objectif d'un cours d'introduction à la sémantique des langages de programmation est double. D'une part, il s'agit de donner une description complète et détaillée du sens que l'on souhaite donner à un programme (sémantique opérationnelle, dénotationnelle, axiomatique...) en explicitant tous les détails laissés dans l'ombre par la "sémantique informelle (ou intuitive)". D'autre part, il s'agit d'exprimer cette sémantique dans un formalisme mathématique afin de pouvoir établir avec rigueur certaines propriétés essentielles. Les étudiants adhèrent volontiers à la nécessité d'explicitier tous les détails d'une sémantique : il est bien clair que l'écriture d'un compilateur ou d'un programme ne peut se faire que si l'on connaît le sens associé à chacune des constructions du langage considéré. En revanche, la formalisation complète de cette sémantique leur apparaît généralement plus difficile. C'est pourtant le "prix à payer" pour pouvoir non seulement implanter le langage mais aussi raisonner sur ce langage. À partir de cette formalisation, les étudiants sont amenés à prouver certaines propriétés classiques vérifiées par le langage considéré et par les programmes écrits dans ce langage (équivalence,

terminaison, déterminisme...). Ces preuves s'établissent la plupart du temps en considérant des arbres d'inférence et en raisonnant par récurrence (sur une expression, sur un arbre, en utilisant un ordre bien-fondé...). Ici encore, la "saveur" formelle de ces concepts laisse croire, à tort, qu'il s'agit d'un exercice difficile.

En fait, la difficulté rencontrée par les étudiants réside dans l'explicitation formelle de connaissances qu'ils utilisent (implicitement) lors de l'écriture d'un programme et dans l'application de techniques de base de démonstration. Pourtant, la plupart d'entre eux n'éprouve aucune appréhension à écrire des programmes ! En effet, écrire un programme est un exercice assisté par des outils logiciels : l'écriture se fait via un éditeur permettant souvent une première mise en forme qui aide à structurer le programme, la compilation aide à corriger les erreurs de syntaxe et, selon les fonctionnalités offertes par le compilateur utilisé, à détecter certaines "fautes" de programmation (non-initialisation de variables, erreurs de type...), l'utilisation d'un debugger permet de tracer pas à pas l'exécution d'un programme et certains outils de documentation "automatique" fournissent une présentation plus lisible d'un programme que son code source. La richesse des environnements de développement actuels guide alors l'étudiant en instaurant avec lui un "dialogue" lui permettant de "valider" ou de corriger son développement. L'absence de cet "outillage" lors d'exercices plus formels conduit l'étudiant à une confrontation beaucoup plus périlleuse (et parfois rebutante) avec la "page blanche". Initialement ce cours était abordé de manière "traditionnelle" et n'était pas toujours très bien vécu. Aussi, le cours de sémantique que nous avons mis en place fait un large usage de l'assistant à la preuve Coq : tout comme avec un programme, l'étudiant peut tester ses définitions et ses preuves. Nous présentons dans cette section les grandes lignes de cet enseignement.

Après de brefs rappels sur les systèmes d'inférence et le raisonnement par récurrence, le cours présenté aux étudiants (en 16h) aborde en détail les mécanismes d'évaluation des expressions arithmétiques, puis, de manière beaucoup plus rapide, traite les expressions booléennes et enfin définit la sémantique opérationnelle des constructions de base d'un langage impératif. Toutes ces notions sont formalisées à l'aide de fonctions et de systèmes d'inférence.

## 2.1. Expressions

La syntaxe abstraite de l'ensemble  $E_A$  des expressions arithmétiques est définie à partir de l'ensemble  $\mathbb{Z}$  des entiers relatifs (type `Z` de Coq), d'un ensemble  $X$  de variables, et des opérations de base ( $+$ ,  $-$ ,  $*$ ,  $/$ ). Elle est introduite par un système d'inférence accompagné de la définition inductive EA correspondante dans le langage de Coq (similaire à une définition en OCaml).

```
Inductive EA : Set :=
  itg : Z -> EA | var : X -> EA | plus : EA -> EA -> EA | ...
```

Dès la première définition, l'utilisation de Coq est fructueuse :

- elle permet de montrer la correspondance entre règles d'inférence (concept jugé abstrait) et constructeurs d'un type somme (concept souvent familier),
- l'écriture en Coq d'un terme de type EA permet alors d'illustrer la notion d'arbre de syntaxe abstraite d'une expression arithmétique,
- enfin, puisque pour chaque type inductif, Coq engendre un schéma d'induction, l'étudiant peut examiner le type de ce schéma et vérifier qu'il retrouve bien la forme générale d'un raisonnement par récurrence structurelle, associé à un système d'inférence, utilisé pour raisonner sur les théorèmes de ce système :

```
EA_ind: forall P : EA -> Prop,
  (forall z : Z, P (itg z)) ->
  (forall x : X, P (var x)) ->
  (forall e : EA, P e -> forall e0 : EA, P e0 -> P (plus e e0)) ->
  ...
  forall e : EA, P e
```

L'évaluation des expressions arithmétiques est ensuite abordée sous des angles différents. Auparavant, l'ensemble  $\mathbb{V}$  des valeurs pouvant être associées aux expressions arithmétiques est définie (il s'agit de l'ensemble  $\mathbb{Z}$  des entiers relatifs auquel est ajouté une valeur spéciale Err correspondant à une expression dont l'évaluation conduit à une division par zéro) et la présence de l'ensemble  $X$  des variables dans les expressions conduit à introduire la notion d'environnement d'évaluation (qui se modélise ici par une fonction de  $X$  dans  $\mathbb{Z}$ ). La première approche considérée consiste à définir une fonction d'interprétation des expressions arithmétiques  $\mathcal{A}[\_]\_ : (X \rightarrow \mathbb{Z}) \rightarrow E_A \rightarrow \mathbb{V}$ . Cette approche permet de revisiter les notions de termes du premier ordre et de schéma d'interprétation des termes sur un exemple concret. S'agissant d'une fonction, le déterminisme est garanti (puisque les opérateurs primitifs sont déterministes). Cette fonction est codée en Coq par une fonction récursive. La construction `Fixpoint` de Coq qui permet ce codage garantit la terminaison de cette fonction.

La sémantique opérationnelle à grands pas d'évaluation des expressions arithmétiques est ensuite définie via un système d'inférence manipulant des jugements de la forme  $\langle a, \sigma \rangle \rightsquigarrow v$  où  $a \in E_A$ ,  $\sigma$  est un environnement d'évaluation et  $v \in \mathbb{V}$ . Ici encore, le codage en Coq de la définition inductive de cette relation est présenté : chaque règle d'inférence correspond à un constructeur, le schéma d'induction associé à cette définition permet d'envisager un raisonnement par récurrence sur l'arbre d'inférence du jugement  $\langle a, \sigma \rangle \rightsquigarrow v$  et la construction d'un arbre d'inférence de ce jugement revient à définir un terme Coq. L'étudiant peut ainsi tester ses constructions à l'aide de Coq et disposer des messages de Coq lui indiquant ses éventuelles erreurs.

Le jeu de tactiques nécessaires pour mener à bien l'essentiel des preuves demandées étant assez restreint, son apprentissage peut se faire facilement au travers d'exemples. C'est aussi l'occasion de prouver les premiers résultats du cours, non pas avec un bâton de craie sur un tableau noir, mais directement sur un ordinateur en utilisant un vidéo-projecteur (le tableau reste tout de même utile pour dessiner la forme de l'arbre de preuve en cours de construction afin de pouvoir identifier l'état courant

de la preuve et l'origine des hypothèses disponibles). Les trois résultats ainsi prouvés sont les suivants :

– *Terminaison.*  $\forall a \in E_A \quad \forall \sigma \quad \exists v \in \mathbb{V} \quad \langle a, \sigma \rangle \rightsquigarrow v$

– *Déterminisme.*

$$\forall a \in E_A \forall \sigma \forall v_1, v_2 \in \mathbb{V} \quad (\langle a, \sigma \rangle \rightsquigarrow v_1 \wedge \langle a, \sigma \rangle \rightsquigarrow v_2) \Rightarrow v_1 = v_2$$

– *Equivalence entre sémantiques.*  $\forall \sigma \forall e \in E_A \forall v \in \mathbb{V} \quad \langle e, \sigma \rangle \rightsquigarrow v \Leftrightarrow \mathcal{A}[[e]]_\sigma = v$

Ces preuves permettent d'illustrer la différence entre relations et fonctions (rien ne garantit *a priori* l'existence et le déterminisme de l'évaluation avec la sémantique à grands pas, il faut prouver ces propriétés), et de montrer l'équivalence des deux approches présentées. L'utilisation de Coq pour établir ces preuves aide les étudiants à comprendre la forme des hypothèses de récurrence et montre clairement l'importance de la forme de l'énoncé : l'ordre des variables quantifiées universellement au début de l'énoncé conditionne la forme des hypothèses de récurrence. Coq fournit alors un outil d'expérimentation qui permet de visualiser les hypothèses de récurrence disponibles avec différents énoncés. Par exemple, lors d'un raisonnement par récurrence sur une expression arithmétique  $a$ , les énoncés suivants sont associés à des hypothèses de récurrence différentes :

Énoncé	Hypothèse de récurrence (cas de l'addition)
$\forall \sigma \forall a \forall v P(a)$	$\forall v P(a_1) \Rightarrow P(a_2) \Rightarrow P(a_1 + a_2)$
$\forall v \forall \sigma \forall a P(a)$	$P(a_1) \Rightarrow P(a_2) \Rightarrow P(a_1 + a_2)$
$\forall a \forall v \forall \sigma P(a)$	$\forall v \forall \sigma P(a_1) \Rightarrow P(a_2) \Rightarrow P(a_1 + a_2)$

On voit donc bien que selon la forme de l'énoncé, l'hypothèse de récurrence est plus ou moins générale. En sensibilisant les étudiants à ces problèmes, l'emploi de Coq pour effectuer ces tests leur permet (presque à leur insu) d'apprendre à maîtriser la conduite d'un raisonnement par récurrence : ce n'est plus l'hypothèse de récurrence qui est difficile à trouver mais plutôt comment l'utiliser.

A l'issue de cette présentation, dans laquelle tout symbole mathématique a été remplacé par un identificateur Coq, les exercices consistant à prouver des propriétés revêtent un aspect plus "pratique", pour ne pas dire plus ludique. La suite du cours ne comporte plus aucune référence à Coq. Néanmoins, le contrôle des connaissances comporte, pour un tiers de la note finale, un projet consistant à formaliser avec Coq l'intégralité de la suite du cours (définitions, propriétés et preuves), ce travail étant réparti en binômes à l'ensemble des étudiants. Comprendre une preuve revient alors à comprendre comment elle va pouvoir se coder dans Coq : les étudiants ne "subissent" plus les preuves "sur le papier" mais cherchent à en extraire les ingrédients qui leur permettront de les formaliser. Nous avons alors constaté que la qualité et la rigueur des preuves rédigées (sur papier) lors de l'examen final écrit par les étudiants est considérablement augmentée grâce à ces projets.

La suite du cours sur les expressions arithmétiques aborde les notions d'expressions équivalentes, et d'indépendance du résultat de l'évaluation d'une expression arithmétique vis à vis des valeurs associées aux variables n'apparaissant pas dans cette expression. Pour finir, la sémantique opérationnelle à petits pas d'évaluation des expressions arithmétiques est présentée et l'équivalence de cette sémantique avec la sémantique opérationnelle à grands pas est prouvée. Toutefois, seules les définitions utilisées pour la sémantique à petits pas sont formalisées dans Coq. En effet, une telle sémantique fait intervenir une notion de séquence de transitions de calcul qui peut être infinie (la terminaison de l'évaluation d'une expression arithmétique dans ce cadre se prouve) contrairement à la notion d'arbre d'inférence qui, par définition, est fini. La spécification en Coq des séquences de transitions, éventuellement infinies, fait intervenir des types coinductifs et les preuves de terminaison nécessitent de manipuler des ordres bien fondés, ce qui rend beaucoup plus technique la formalisation des preuves (nous ne disposons pas du temps nécessaire pour aborder ces techniques).

Enfin, les expressions booléennes sont très rapidement présentées. La démarche suivie est exactement la même que pour les expressions arithmétiques. Néanmoins, nous ne donnons plus aucune preuve : toutes doivent être formalisées par les étudiants à l'aide de Coq. Nous insistons toutefois sur la gestion "paresseuse" des opérateurs de conjonction et de disjonction et de l'incidence des choix faits sur le résultat de l'évaluation des expressions booléennes. Certains choix invalident en effet l'équivalence entre sémantiques. Par exemple le résultat de l'évaluation de  $(x = 0) \vee (10/x \leq 5)$  dépend de ces choix.

## 2.2. Constructions impératives

La dernière partie du cours présente la sémantique opérationnelle à grands pas des constructions de base d'un langage impératif. Sont abordées les notions de terminaison des programmes (récurrence bien fondée), de programmes équivalents et de déterminisme de l'exécution des programmes. L'introduction d'une construction conduisant à une exécution potentiellement infinie (la construction *while*) conduit pour la première fois à mener des raisonnements par récurrence non plus sur la structure d'un programme, mais sur l'arbre d'inférence d'un jugement de la forme  $\langle P, \sigma \rangle \rightarrow \sigma'$  exprimant que l'exécution d'un programme  $P$  dans un état  $\sigma$  conduit à un état  $\sigma'$ . Ici encore, l'utilisation de Coq permet de forcer l'étudiant à expliciter sur quel objet porte la récurrence et de constater que, selon ses choix, la forme de l'hypothèse de récurrence dont il dispose lui permet ou ne lui permet pas de mener à bien sa preuve. C'est typiquement le cas lors de la preuve du déterminisme de l'exécution des instructions :

$$\forall \sigma, \sigma_1, \sigma_2 \quad \forall P \quad (\langle P, \sigma \rangle \rightarrow \sigma_1 \text{ et } \langle P, \sigma \rangle \rightarrow \sigma_2) \Rightarrow \sigma_1 = \sigma_2$$

Un raisonnement par récurrence sur  $P$  ne permet pas de conclure. En effet, dans ce cas, lorsque  $P$  est le programme  $\langle \text{while } e \text{ do } P_0 \rangle$ , et lorsque l'expression booléenne  $e$

s'évalue à true, on souhaite établir l'égalité  $\sigma_2 = \sigma'_2$  à partir des deux arbres d'inférence suivant :

$$(C_7) \frac{\frac{(B_i) \frac{\vdots}{\langle e, \sigma \rangle \rightsquigarrow \text{true}} \quad (C_j) \frac{\vdots}{\langle P_0, \sigma \rangle \rightarrow \sigma_1} \quad (C_k) \frac{\vdots}{\langle \text{while } e \text{ do } P_0, \sigma_1 \rangle \rightarrow \sigma_2}}{\langle \text{while } e \text{ do } P_0, \sigma \rangle \rightarrow \sigma_2}}{\frac{(B_i) \frac{\vdots}{\langle e, \sigma \rangle \rightsquigarrow \text{true}} \quad (C_{j'}) \frac{\vdots}{\langle P_0, \sigma \rangle \rightarrow \sigma'_1} \quad (C_{k'}) \frac{\vdots}{\langle \text{while } e \text{ do } P_0, \sigma'_1 \rangle \rightarrow \sigma'_2}}{\langle \text{while } e \text{ do } P_0, \sigma \rangle \rightarrow \sigma'_2}}$$

Un raisonnement par récurrence sur la structure de  $P$  permet de disposer d'une hypothèse de récurrence sur  $P_0$  à partir de laquelle on peut déduire l'égalité  $\sigma_1 = \sigma'_1$ . Mais on ne peut pas aller plus loin, car  $\langle \text{while } e \text{ do } P_0, \sigma_1 \rangle \rightarrow \sigma_2$  a la même structure while ... do que le programme  $P$  pour lequel on cherche à faire la preuve (et n'est donc pas structurellement plus petit). L'étudiant peut alors facilement vérifier avec Coq qu'une hypothèse supplémentaire, correspondant à celle qui lui manquait, est disponible s'il met en oeuvre un raisonnement par récurrence sur l'arbre d'inférence du jugement  $\langle P, \sigma \rangle \rightarrow \sigma_1$ . Il lui faudra pour cela préciser quelle propriété  $Q$  il cherche à établir :

$$Q(\langle P, \sigma \rangle \rightarrow \sigma_1) \Leftrightarrow \forall \sigma_2 \langle P, \sigma \rangle \rightarrow \sigma_2 \Rightarrow \sigma_1 = \sigma_2$$

et l'hypothèse d'induction dont il disposera alors s'exprimera par :

$$\begin{aligned} & \forall P_0 \forall \sigma, \sigma', \sigma'' \forall e \\ & (\langle e, \sigma \rangle \rightsquigarrow \text{true} \text{ et } Q(\langle P_0, \sigma \rangle \rightarrow \sigma') \text{ et } Q(\langle \text{while } e \text{ do } P_0, \sigma' \rangle \rightarrow \sigma'')) \\ & \Rightarrow Q(\langle \text{while } b \text{ do } P_0, \sigma \rangle \rightarrow \sigma'') \end{aligned}$$

Les preuves de déterminisme et d'équivalence de certains programmes ne peuvent s'obtenir qu'en menant un raisonnement par récurrence sur un arbre d'inférence. Comme on le voit, l'étudiant est amené à préciser de manière explicite l'objet sur lequel porte la récurrence et la propriété qu'il cherche à établir. On constate alors que certaines preuves classiques, rarement réussies sur le papier par les étudiants, sont obtenues correctement dans Coq (c'est par exemple le cas de la preuve d'équivalence entre les constructions while et repeat).

Pour finir, la sémantique opérationnelle à petits pas est présentée et l'équivalence de cette sémantique avec la sémantique opérationnelle à grands pas est prouvée. Mais, ici encore, seules les définitions sont formalisées en Coq.

Signalons aussi, que comme pour les expressions, il est bien sûr possible de définir une sémantique dénotationnelle pour les programmes. Cette sémantique est présentée

aux étudiants mais ne fait pas l'objet d'une formalisation avec Coq (pour plus de détails sur ce sujet, on pourra consulter (Bertot *et al.*, 2002)).

### 3. Projet Coq

Le but du projet Coq donné cette année (2003/2004) aux étudiants de DESS a pour objectif de compléter la formation Coq de début d'année (environ 4 séances de 3h pendant lesquelles Coq est présenté mais aussi manipulé en pratique), ainsi que l'utilisation de Coq faite au cours de l'enseignement de sémantique (voir section 2). L'idée est de faire travailler les étudiants sur un exemple concret et raisonnablement élaboré (sachant qu'ils ont un délai de 3 mois pour le réaliser en travaillant en moyenne de 3 à 6h par semaine) pour leur montrer les problèmes que peuvent occasionner un changement d'échelle de développement. Le sujet a consisté à réaliser une bibliothèque de polynômes à une variable sur un corps. Plus précisément, il s'agissait d'implanter les opérations d'addition, de multiplication, d'opposé puis d'établir les propriétés d'anneau. Il fallait ensuite munir l'anneau d'une norme euclidienne pour obtenir la division euclidienne. Enfin, le pgcd était d'abord introduit de manière abstraite (par la notion de divisibilité), puis il fallait coder l'algorithme d'Euclide en Coq en montrant le lemme de correction correspondant. De manière annexe, il était également demandé d'écrire une tactique permettant de montrer automatiquement qu'un polynôme était le pgcd de deux autres polynômes. Enfin, il fallait montrer une version généralisée du théorème de Bezout. Une spécification de base a été fournie aux étudiants fixant notamment la représentation des polynômes et donnant les déclarations des fonctions, tactiques ainsi que des lemmes à compléter. Aucune consigne n'a été donnée sur le respect de cette spécification, il était donc possible de la modéliser à souhait pour réaliser cette implantation. Compte-tenu du travail assez conséquent qui était demandé, le projet a été subdivisé en tâches qui ont été distribuées aux étudiants. En particulier, un groupe a pris en charge la direction du projet, avec pour objectif de plonger davantage les étudiants dans "l'ambiance" d'un projet en grandeur nature.

#### 3.1. *Choix d'une représentation : types inductifs et types dépendants*

Comme nous l'avons souligné plus haut, un des points forts de l'outil Coq est probablement la notion primitive de type inductif, ainsi que le support fourni par le système (génération de schémas d'induction, terminaison, tactiques d'induction...). En effet, l'utilisateur (en l'occurrence, l'étudiant) est systématiquement guidé dans toutes les "phases" de l'induction :

– d'abord dans l'écriture du type inductif lui-même : Coq impose le respect de conditions syntaxiques de positivité. Intuitivement, il s'agit d'éviter des termes de la forme Inductive  $t$ : Set := Cons : ( $t \rightarrow t$ )  $\rightarrow$   $t$ , qui peuvent conduire à des incohérences ;

– ensuite dans l'implantation des fonctions définies par récurrence (un appel récursif ne peut être fait que sur un terme structurellement plus petit que l'argument de départ, ce qui assure la terminaison de la fonction) ;

– enfin dans les preuves par récurrence (application automatique des schémas d'induction au moyen de tactiques génériques).

Même si l'utilisateur n'est pas toujours en mesure de comprendre pourquoi telle ou telle contrainte est imposée (il faut en effet un certain bagage théorique pour pouvoir les appréhender), un contrôle automatique est assuré (même s'il peut se révéler parfois trop restrictif), garantissant la terminaison et donc la cohérence du développement. Dans le cadre du projet, la représentation des polynômes est fixée au préalable et est naturellement basée sur une liste de couples coefficient/degré, implantée au moyen d'un type inductif. Cette implantation peut ressembler à :

```
Inductive pol : Set := PList : list (A * nat) -> pol.
```

où A représente le corps des coefficients (de type Set) et list est le type inductif (prédéfini) des listes polymorphes.

Cependant, cette représentation a pour inconvénient de ne pas être canonique. En effet, la liste correspondant à un polynôme peut contenir des coefficients nuls mais aussi plusieurs monômes de même degré. Ainsi, il est impossible de comparer deux polynômes en utilisant la simple égalité syntaxique (appelée égalité de Leibniz en Coq). Ceci peut ne pas être très gênant dans l'optique d'un langage de programmation mais dans un outil comme Coq, basé de manière rigide sur l'égalité de Leibniz, il est préférable d'avoir une forme canonique pour pouvoir utiliser pleinement certaines tactiques de preuves. L'idée est donc de contraindre très fortement la représentation en utilisant, pour cela, un autre point fort de Coq (qui n'est pas présent dans tous les outils d'aide à la preuve), à savoir les types dépendants. Ainsi, dans un premier temps, les étudiants ont considéré un polynôme comme une liste de couples coefficient/degré telle qu'aucun des coefficients n'est nul et la liste est triée en ordre décroissant strict (il n'y a qu'une occurrence d'éléments égaux) selon le degré. Pour ce faire, un prédicat "canonique" est défini, puis utilisé dans la représentation des polynômes, comme suit :

```
Inductive canonical : list (A * nat) -> Prop :=
  can_nil : canonical nil
| can_zro : forall (a : A) (n : nat) (l : list (A * nat)),
  ~(a = Azero) -> canonical (cons (a, n) nil)
| can_ord : forall (a1 a2 : A) (n1 n2 : nat) (l : list (A * nat)),
  n1 > n2 -> a1 <> Azero -> a2 <> Azero ->
  canonical (cons (a2, n2) l) ->
  canonical (cons (a1, n1) (cons (a2, n2) l)).
Inductive pol : Set :=
  PList : forall l : list (A * nat), canonical l -> pol.
```

où Azero représente le 0 du corps A et nil/cons sont les constructeurs de list.

Ainsi, dans cette nouvelle version, un polynôme est un couple dépendant composé d'une liste de couples coefficient/degré et d'une preuve que cette liste est canonique. Toutefois, les étudiants ont trouvé cette représentation trop rigide rendant notamment l'écriture des opérations sur les polynômes (addition, multiplication...) complexe. Ils ont donc décidé astucieusement (en accord avec leur enseignant) de scinder la difficulté en deux en imposant bien une liste triée pour la construction de polynômes mais en repoussant la comparaison de deux polynômes modulo les coefficients nuls au niveau de l'égalité (quitte à devoir montrer des lemmes de compatibilité incombant à cette nouvelle égalité de setoïde<sup>1</sup>). Ainsi, l'écriture des opérations devient plus simple, mais la preuve des propriétés plus difficile (au passage, la complexité des tâches gagnait en uniformisation). Par ailleurs, cela permet également d'isoler clairement les parties potentiellement indécidables du développement. En effet, si trier la liste des monômes selon le degré est décidable, savoir si un coefficient est nul ou non ne l'est pas forcément (par exemple, si A représente l'ensemble des nombres réels, il n'est pas possible de décider si un coefficient est nul). Dans cette nouvelle représentation, montrer que la liste est canonique est donc toujours décidable et la construction des polynômes peut être automatisée, alors que dans l'ancienne version et de manière générale, il fallait montrer à la main que la liste était canonique avant de pouvoir former un polynôme. Finalement, voici la dernière option qui a été retenue :

```

Inductive sorted : list (A * nat) -> Prop :=
| sorted_nil : sorted nil
| sorted_one : forall (a : A) (n : nat), sorted (cons (a, n) nil)
| sorted_cns : forall (a0 a1 : A) (n0 n1 : nat) (l : list (A * nat)),
    sorted (cons (a1, n1) l) -> n0 > n1 ->
    sorted (cons (a0, n0) (cons (a1, n1) l)).
Inductive pol : Set :=
| PList : forall l : list (A * nat), sorted l -> pol.

```

puis, la définition de l'égalité sur les polynômes :

```

Inductive eql : list (A * nat) -> list (A * nat) -> Prop :=
| eql_nil : forall p : list (A * nat), eql nil nil
| eql_cns : forall (l0 l1 : list (A * nat)) (m0 m1 : A * nat),
    m0=m1 -> eql l0 l1 -> eql (cons m0 l0) (cons m1 l1)
| eql_0lt : forall (l0 l1 : list (A * nat)) (n : nat),
    eql l0 l1 -> eql (cons (Azero, n) l0) l1
| eql_0rt : forall (l0 l1 : list (A * nat)) (n : nat),
    eql l0 l1 -> eql l0 (cons (Azero, n) l1).
Definition eqp := fun (p1 p2 : pol) =>
    let (l1, _) := p1 in let (l2, _) := p2 in (eql l1 l2).

```

En outre, les lemmes suivants ont été introduits afin de faciliter l'automatisation dans la décidabilité de cette égalité suivant que l'égalité sur A est décidable ou non :

1. Un setoïde est un ensemble muni d'une relation d'équivalence. De manière générale, cette relation n'est pas une congruence.

```

Lemma eql_cns_dec :
  forall (l0 l1 : list (A * nat)) (a0 a1 : A) (n : nat),
    a0 = a1 -> eql l0 l1 -> eql (cons (a0, n) l0) (cons (a1, n) l1).
Lemma eql_0lt_dec : forall (l0 l1 : list (A * nat)) (a : A) (n : nat),
  a = Azero -> eql l0 l1 -> eql (cons (a, n) l0) l1.
Lemma eql_0rt_dec : forall (l0 l1 : list (A * nat)) (a : A) (n : nat),
  a = Azero -> eql l0 l1 -> eql l0 (cons (a, n) l1).

```

Au-delà de cette représentation (dont le choix est fondamental pour l'écriture des fonctions et la construction des preuves), les étudiants n'ont pas eu à introduire d'autres types inductifs (que ce soit pour des types de données ou pour des prédicats). Par contre, les types dépendants ont du être encore largement utilisés (norme euclidienne, théorème de Bezout...).

### 3.2. Fonctions et preuves

Un autre avantage de Coq, toujours relié à l'induction, est le support qu'il offre dans l'écriture des fonctions récursives. En effet, les étudiants doivent systématiquement écrire des fonctions qui terminent. Ceci n'est évidemment pas véritablement nouveau et l'on peut légitimement attendre de ces mêmes étudiants qu'ils se donnent la même contrainte dans tout autre langage de programmation. Cependant, la différence est qu'ici, ils doivent "convaincre" le système que leurs fonctions terminent. Pour ce faire, Coq repose par défaut sur un ordre bien fondé prédéfini, à savoir l'ordre sous-terme (on parle alors de récursion structurelle) qui s'avère extrêmement naturel à appréhender. De ce fait, les étudiants n'ont pas trop de mal à se plier à cette nouvelle contrainte et comprennent assez rapidement qu'ils ne peuvent pas toujours se contenter d'écrire leurs fonctions naïvement. Ils apprennent donc à décomposer leurs fonctions de manière à bien identifier les différents appels récursifs nécessaires. A titre d'exemple, voici la version de l'addition de deux polynômes qu'ils ont réalisée :

```

Fixpoint UPplus_monom_list (a : A) (n : nat) (l : list (A * nat))
  {struct l} : list (A * nat) :=
  match l with
  | nil => cons (a, n) nil
  | (cons (b, m) tl) =>
    (match (lt_eq_lt_dec n m) with
    | (inleft (left _)) => cons (b, m) (UPplus_monom_list a n tl)
    | (inleft (right _)) => cons ((Aplus a b), n) tl
    | (inright _) => cons (a, n) l
    end) end.
Definition UPplus_monom (a : A) (n : nat) (p : pol) : pol :=
  let (l, q) := p in
  PList2 (UPplus_monom_list a n l) (UPplus_monom_proof a n l q).
Fixpoint UPplus_list (p0 : pol) (l : list (A * nat)) {struct l} : pol :=
  match l with
  | nil => p0

```

```

| (cons (a,n) t1) => UPplus_list (UPplus_monom a n p0) t1
end.
Definition UPplus (p0 p1 : pol) : pol :=
  let (l, _) := p1 in (UPplus_list p0 l).

```

L'idée générale des étudiants est d'itérer une fonction d'ajout de monôme dans un polynôme (UPplus\_monom) sur le deuxième polynôme (argument de l'addition). Tous les appels récursifs (de chaque sous-fonction) sont bien fondés (selon l'ordre sous-terme) et cette fonction a été testée (également par les étudiants) avec succès.

Finalement, la plus grande difficulté rencontrée par les étudiants dans l'écriture de leurs fonctions récursives provient de la représentation des polynômes même et notamment du type dépendant utilisé. Ceci n'est pas particulièrement surprenant car la programmation avec types dépendants peut s'avérer fastidieuse (des termes peuvent intervenir dans les types et il peut être difficile de les convertir). Ainsi, les étudiants ont dû écrire non seulement la fonction qui ajoute un monôme dans un polynôme mais aussi la fonction de correction, qui produit une preuve que la liste obtenue est toujours triée. Les étudiants ont compris la difficulté de l'écriture directe de cette fonction et ils ont pris l'initiative de construire cette fonction par la preuve. Cette idée est d'autant plus étonnante que Coq ne leur avait pas été véritablement introduit théoriquement. En particulier, l'isomorphisme de Curry-Howard n'avait pas été formellement présenté mais simplement évoqué. Le lemme auxiliaire introduit est le suivant (qui apparaît dans la fonction UPplus\_monom vue plus haut) :

```

Lemma UPplus_monom_proof: forall (a : A) (n : nat) (l : list (A * nat)),
  sorted l -> sorted (UPplus_monom_list a n l).

```

Poser ce lemme s'avère d'autant plus nécessaire que sa preuve utilise, à plusieurs reprises, une tactique (écrite par les étudiants) pour montrer automatiquement qu'une liste est triée et Coq ne permet pas l'utilisation de tactiques directement dans les termes. C'est probablement un point que l'on peut regretter dans son utilisation dans la mesure où une tactique n'est pas si différente d'un terme puisqu'elle décrit la façon d'en construire un. De manière annexe, (Delahaye, 2001) présente un langage de preuves permettant une telle possibilité dans le cadre de Coq.

Concernant les preuves, les étudiants ont d'abord éprouvé de nettes difficultés (notamment ceux chargés des preuves sur les opérations d'addition et de multiplication), mais ont finalement pour la plupart bien réussi à appréhender le système de preuves. Parmi les difficultés initiales, on retrouve bien sûr le problème de formalisation, à savoir qu'une preuve formelle est bien plus fine qu'une preuve mathématique (certes rigoureuse mais pleine d'imprécisions et de justifications implicites). Cette difficulté est d'ailleurs exacerbée par une très faible granularité des preuves Coq, où l'automatisation ne prend pas toujours le relais comme on pourrait le souhaiter. Par contre, le langage de preuves procédural de Coq (même s'il possède un certain nombre d'inconvénients comme la lisibilité par exemple) soutenu par une boucle interactive très intuitive s'est avéré complètement adéquat dans le cadre de ce projet. Les étudiants ont pu le prendre en main très rapidement et sont parvenus à construire des preuves

non triviales assez vite. Toutefois, les étudiants ont parfois tendance à se laisser trop “porter” par cette boucle interactive qui, si elle les aide à construire une preuve, ne leur permet pas forcément de la trouver. Ainsi, il faut insister régulièrement sur l'utilisation du papier/crayon avant de se lancer dans une preuve que l'on estime non triviale.

### 3.3. Automatisation et métalangage

Parmi les objectifs du projet, il était également demandé aux étudiants d'écrire de petites (voire même plus conséquentes) automatisations. Pour ce faire, Coq s'avère complètement adapté notamment grâce à son langage de tactiques à toplevel  $\mathcal{L}_{tac}$  (Delahaye, 2001), dédié justement à de telles utilisations. En effet, compte-tenu du temps imparti, il n'était pas question de faire écrire une quelconque automatisation en OCaml (le langage d'implantation mais aussi et surtout le métalangage de Coq), une telle approche nécessitant une trop grande connaissance du code de Coq. Avec  $\mathcal{L}_{tac}$ , les étudiants peuvent s'abstraire complètement du côté technique et se concentrer sur l'automatisation en profitant notamment des itérateurs de haut niveau sur les preuves. Les automatisations demandées consistaient à montrer qu'une liste (de monômes) est triée (suivant le degré), puis à montrer l'égalité entre deux polynômes (la représentation retenue implique une égalité de setoïde, voir sous-section 3.1) et enfin à montrer qu'un polynôme est le pgcd de deux autres. Voici, à titre d'exemple, la version de la première automatisation demandée (liste triée) telle que donnée par les étudiants :

```
Ltac is_sorted := repeat
  match goal with
  | [|- sorted nil] => apply sorted_nil
  | [|- sorted (cons (_, _) nil)] => apply sorted_one
  | [|- sorted (cons (_, _) (cons (_, _) _))] =>
    (apply sorted_cns); [idtac | progress auto] end.
```

où `sorted_nil`, `sorted_one` et `sorted_cns` sont les constructeurs du prédicat inductif `sorted` donné précédemment (voir sous-section 3.1).

Cette tactique fonctionne parfaitement (elle a été testée sur plusieurs exemples par les étudiants). Elle s'arrête lorsque les deux monômes de tête ne sont pas en ordre décroissant strict (échec de la tactique `progress auto` qui se charge de montrer les conditions sur les degrés de tête). Concernant les autres automatisations, les étudiants ont eu besoin d'un peu d'aide. Par exemple, pour l'égalité, l'idée est d'utiliser l'égalité sur les corps pour les coefficients (d'où l'introduction des lemmes `eql_cns_dec`, `eql_0lt_dec` et `eql_0rt_dec`, de la sous-section 3.1 et engendrant des égalités sur lesquelles la tactique prédéfinie `field`, prouvant automatiquement des égalités sur des corps, peut s'appliquer). Quant à la tactique sur le pgcd, elle s'est avérée un peu trop technique (la formation Coq concernant les tactiques utilisateurs avait été très courte) car elle nécessite en réalité d'implanter l'algorithme d'Euclide (même si dans le métalangage, la situation est bien plus simple puisque la terminaison n'est pas imposée).

### 3.4. Curry-Howard et extraction

Un peu comme un aboutissement au projet, une des tâches consistait à montrer aux étudiants comment il est possible d'extraire un programme d'une preuve. Pour cela, il s'agissait de "simuler" l'algorithme d'Euclide en prouvant l'existence de  $\text{pgcd}(p_1, p_2)$ , pour deux polynômes  $p_1$  et  $p_2$ . Cette tâche a mis en évidence les limites d'une approche de Coq purement utilisateur dégagée de toute méta-théorie. En effet, les étudiants n'ont pu achever ce travail : il faut être un bon technicien Coq pour parvenir à mettre en pratique le leitmotiv "prouver = programmer" et les étudiants n'avaient pas encore atteint un tel niveau. Devant cette impasse, ils ont tenté une approche plus classique en écrivant directement le code de la fonction `pgcd` puis en montrant un lemme de correction. Toutefois, ce lemme se prouve, non pas par induction structurelle sur les polynômes, mais par induction sur leur degré. On ne peut donc plus utiliser l'induction par défaut de Coq. Si la preuve n'est pas mathématiquement compliquée, une telle induction est assez technique à mettre en œuvre en Coq. Si on ne peut pas reprocher à Coq de ne pas être plus aidant dans des approches à la Curry-Howard, on peut cependant regretter la faiblesse du support quand on utilise d'autres schémas d'induction que ceux engendrés par le système (d'autant plus que cette situation n'est pas si rare).

## 4. Conception formelle : utilisation de l'atelier Focal

Le cours de conception formelle présente les méthodologies de développement par raffinements et est illustré par l'utilisation de l'atelier B (Abrial, 1996). Afin de compléter et de mettre en pratique les notions introduites dans ce cours, les étudiants ont à réaliser un projet utilisant l'atelier Focal. Celui-ci fournit un langage construit sur OCaml et Coq pour spécifier, programmer et prouver des unités de bibliothèques qui sont traduites par le compilateur Focal vers du code source OCaml pour aboutir à des programmes exécutables, et vers du code source Coq afin de vérifier les preuves. Focal permet donc à la fois d'écrire des programmes et de prouver leurs propriétés.

Focal autorise un passage progressif d'une spécification à son implantation grâce aux traits orientés-objet dont il dispose (héritage, redéfinition et instanciation). Si on fait abstraction des preuves, les mécanismes de Focal permettent également d'illustrer les principes énoncés par Meyer (Meyer, 1992) dans le cadre du "Design by Contract", les énoncés de propriétés jouant le rôle des pré- et post-conditions de ce paradigme. Plus généralement, la présentation de Focal permet de sensibiliser les étudiants aux méthodes formelles dans le cadre particulier des langages objets (même si le projet ne prévoyait pas d'intervenir directement sur la hiérarchie de la bibliothèque manipulée). L'atelier Focal et la méthodologie sous-jacente ont principalement été appliqués jusqu'à présent au calcul formel. C'est en fait ce domaine qui a servi de modèle et donné les lignes directrices du développement de l'atelier. Grâce à l'atelier Focal, Renaud Rioboo a pu construire une bibliothèque de calcul formel assez conséquente (Rioboo, 2002). Les projets soumis aux étudiants portent sur cette bibliothèque dans laquelle beaucoup de preuves restent à faire. Au niveau des structures de base de

la bibliothèque, les preuves ne présentent pas de difficulté mathématique intrinsèque (il s'agit de propriétés de niveau DEUG en algèbre). En revanche, le contexte de définitions et d'hypothèses dans lequel elles peuvent être établies est plus difficile à cerner.

Faire ces preuves a ainsi permis aux étudiants de mieux appréhender les dépendances entre les différentes méthodes d'un langage orienté-objet : pour prouver qu'une fonction se comporte correctement, il faut disposer, outre des préconditions sur ses arguments, des spécifications des fonctions appelées, qui peuvent être héritées, et donc se trouver à un tout autre endroit que le théorème que l'on veut prouver. Par ailleurs, travailler sur un "vrai" projet, dont les résultats sont destinés à être utilisés par la suite, plutôt que sur un sujet destiné uniquement à illustrer le cours constitue pour une majorité d'étudiants un facteur de motivation puissant.

#### 4.1. L'atelier Focal

En Focal, les structures sont représentées par des *espèces* qu'on peut en première approche assimiler aux classes d'un langage orienté-objet. L'implantation des bibliothèques peut être effectuée étape par étape grâce à la notion d'héritage qui relie les espèces entre elles. Une espèce est définie par un ensemble de méthodes. Certaines méthodes peuvent être uniquement déclarées (donc non définies), ce qui permet de les manipuler sans connaître leur implantation : l'espèce correspond alors à une classe virtuelle en langage orienté-objet. L'héritage permet de définir une espèce à partir d'autres espèces déjà définies. Une espèce hérite de toutes les méthodes de ses parents. Les collections permettent de coder des domaines mathématiques tels que les entiers, les polynômes, etc. Les espèces permettent donc de définir une famille de collections comme les classes permettent de décrire une famille d'objets en programmation orienté-objet. Toutes les méthodes d'une collection doivent être définies *et toutes ses obligations de preuve déchargées*. Le tableau suivant dresse un parallèle entre les notions usuelles du langage, celles de l'algèbre et des langages orientés-objet :

ALGÈBRE	FOCAL	L.O.O.
structure	espèce	classe
élément	entité	
loi, constante	méthode	méthode
propriété, preuve	méthode	
domaine de définition	signature	type de méthode
ensemble, domaine	collection	objet

Les entités sont les éléments (les objets mathématiques) manipulés à l'intérieur des espèces. Le type support d'une espèce est le type de ses entités. Chaque espèce contient un unique type support, éventuellement hérité. Les méthodes représentent les opérations de base ou les propriétés des structures mathématiques. Il y a quatre catégories de méthodes :

1) le *type support* (déclaré sous forme d'un type abstrait, ou bien lié à un type de donné (`int`, `list(string)`,...));

2) les *signatures* représentant les opérations *déclarées* (introduction du nom et du type de l'opération)

3) les *méthodes* représentant les opérations *définies* ; (introduction du nom, du type et de la définition de l'opération)

4) les *propriétés* (déclarées) et les *théorèmes* (prouvés, donc définis) vérifiés par l'espèce et impliquant des obligations de preuve ;

Les interfaces permettent de décrire les structures mathématiques abstraites (comme les anneaux, les algèbres libres, etc) en énumérant les propriétés que vérifient ces structures. À toute espèce est implicitement associée une interface, obtenue en effaçant le corps des méthodes définies (fonctions ou théorèmes).

#### 4.2. Utilisation de Coq pour l'analyse de code Focal

Comme nous l'avons déjà dit, les preuves à faire par les étudiants ne présentent pas de difficulté particulière mais nécessitent une bonne compréhension des mécanismes mis en œuvre dans une structure hiérarchique. Aussi, ces projets visent à aider les étudiants à maîtriser ces concepts en leur fournissant un environnement dans lequel ils peuvent expérimenter et observer la sémantique des traits objets de Focal (héritage et résolution de conflits). Cette observation se fait en examinant le code Coq produit par le compilateur Focal. En effet, afin de permettre la preuve des propriétés spécifiées en Focal, le code Coq "met à plat" la hiérarchie Focal en explicitant les propriétés et les définitions disponibles pour établir la preuve cherchée. Bien sûr, ces projets sont accompagnés d'une présentation détaillée de la traduction de code Focal en code Coq (Focal development team, 2003). Les points importants pour cet article concernent la traduction des espèces et des théorèmes qui y sont prouvés. Brièvement, une espèce *s* est représentée en Coq par une section, ou – ce qui est presque équivalent du point de vue de Focal – un module. Au sein de cette structure, tout théorème prouvé au niveau de *s* est encapsulé dans une section, où sont données les hypothèses (sous forme de paramètre de la section) et les définitions (sous forme de variable locale, utilisant éventuellement les paramètres précédents) dont on a besoin pour faire la preuve. Cette forme permet la réutilisation de la preuve dans les espèces qui héritent de *s*, du moins tant qu'elle n'est pas invalidée par la redéfinition d'une méthode dont la preuve dépend explicitement (auquel cas le compilateur Focal exige une nouvelle preuve utilisant la nouvelle définition). Les dépendances des théorèmes vis-à-vis des autres méthodes et leur traduction en Coq sont décrites de manière plus détaillée dans (Prevosto, 2003; Boulmé *et al.*, 1999).

### 4.3. Ingénierie de la preuve

La tâche demandée aux étudiants (par groupe de 2 ou 3) consistait à donner une preuve, sous forme d'un script Coq, de certaines propriétés des premières espèces de la librairie. En premier lieu, il s'agissait pour eux d'identifier les méthodes (propriétés et définitions) dont ils allaient avoir besoin dans la preuve, afin que Focal leur génère un environnement adapté. La consigne était d'éviter autant que possible le recours direct aux définitions des fonctions, car toute preuve dépendant explicitement de la définition d'une fonction  $f$  dans une espèce  $s$  donnée devient invalide dans une espèce où  $f$  est redéfinie. Une fois l'environnement trouvé, la preuve proprement dite était en général relativement simple. Les étudiants ont obtenu des preuves qui ont pu facilement être incorporées dans la librairie Focal (les preuves originales figurent dans la version 0.1). D'une part, les étudiants ont pu ainsi se familiariser avec l'utilisation de méthodes formelles dans le contexte d'un langage objet. En effet, il leur a fallu rechercher au sein de la hiérarchie les définitions, mais surtout les spécifications dont ils avaient besoin afin d'obtenir l'environnement de preuve adéquat. D'autre part, l'obligation de définir le contexte avant de se lancer dans la preuve s'est révélé bénéfique au-delà du cadre de Focal : il est nettement plus facile de démontrer un théorème en Coq lorsqu'on a une idée claire des propriétés qu'on va utiliser.

Un "effet de bord" des projets réalisés par les étudiants a porté sur des questions de méthodologie en ingénierie de la preuve. En effet, la redéfinition d'une méthode au cours d'un héritage peut conduire à invalider des preuves utilisant l'ancienne définition. Ces preuves doivent donc être refaites en utilisant la nouvelle définition. Considérons par exemple l'espèce représentant un ordre partiel : cette espèce comporte en particulier la spécification d'une relation d'équivalence  $eq$  et d'une relation d'ordre partiel  $leq$ , et la définition de la relation d'ordre strict associé  $lt$ . Plus précisément, la définition est de  $lt$  est :

$$lt(x, y) \stackrel{def}{=} andb(leq(x, y), notb(eq(x, y)))$$

Les étudiants ont prouvé le théorème  $leq\_lt$  :

$$(\forall x, y \ lt(x, y) \Rightarrow (leq(x, y) \wedge \neg eq(x, y))) \wedge (\forall x, y \ leq(x, y) \Rightarrow (lt(x, y) \vee eq(x, y)))$$

L'espèce représentant un ordre total hérite de cette espèce et ajoute la déclaration de la propriété  $total\_order$  caractérisant un ordre total ( $\forall x, y \ leq(x, y) \vee leq(y, x)$ ). La présence de cette propriété permet de proposer une nouvelle définition de  $lt$ , plus simple que celle de  $s_1$  :  $lt \stackrel{def}{=} notb(leq(y, x))$ . Cependant, cette redéfinition invalide la preuve du théorème  $leq\_lt$ , qui doit être reprouvée. Or la librairie actuelle ne définit quasiment que des ordres totaux : la preuve effectuée par les étudiants ne se retrouve dans aucune implantation.

Plus généralement, se pose un problème d'ingénierie de la preuve : quand et où faire les preuves ? Etant donnée une propriété  $P$  dépendant de la définition d'une méthode  $m$ , il s'agit de trouver le "meilleur endroit" dans le graphe d'héritage pour établir

la preuve de  $P$ . Cette question a été développée dans (Prevosto *et al.*, 2003) à partir des résultats de l’incorporation des preuves des étudiants dans la librairie standard de Focal. Cette expérience a ainsi conduit à proposer des méthodes de programmation adaptées au développement incrémental proposé par Focal, illustrant les rapports fructueux que peuvent entretenir enseignement et recherche.

## 5. Conclusion

Un cours est étiqueté “abstrait” ou “théorique” (pour ne pas dire “difficile”) lorsque son contenu est perçu comme une succession de définitions et de propriétés accompagnées de leurs preuves (même lorsque ces concepts portent sur des notions connues des étudiants en informatique, à savoir les langages de programmation et les programmes). A l’inverse, un cours est “pratique” ou “appliqué” (pour ne pas dire “populaire”) lorsqu’il est accompagné de nombreux travaux pratiques permettant la mise en œuvre des notions présentées sur un ordinateur. Nous sommes partis du constat que les étudiants en informatique écrivaient volontiers des programmes alors qu’ils éprouvaient de grandes difficultés à définir, spécifier et prouver les concepts de base des objets manipulés en informatique. Aussi, nous avons choisi de leur faire implanter et manipuler les définitions, propriétés et preuves à l’aide de l’assistant à la preuve Coq, ramenant ainsi cet exercice à un exercice de programmation. Alors que la page blanche est une maigre source d’inspiration pour les étudiants et ne réagit pas aux erreurs (une feuille de papier accepte toujours l’écriture de preuves fausses), l’utilisation d’un logiciel instaure un certain dialogue. En particulier, conduire une preuve à l’aide d’un assistant interactif permet de tester ses idées, et d’obtenir un certain retour, tout en garantissant au final la correction de la preuve établie.

Il est important de noter que les étudiants auxquels nous nous sommes adressés n’avaient suivi aucune présentation approfondie de la méta-théorie de Coq (calcul des constructions inductives), nous supposons juste une connaissance pratique du langage OCaml. Cela démontre qu’il est donc parfaitement possible de réaliser des développements avec Coq sans en maîtriser la théorie sous-jacente. La présentation d’un sous-ensemble du langage de spécification (typiquement, le calcul des prédicats et l’induction) et du langage de tactiques de Coq au travers de petits exemples permet un apprentissage rapide et efficace de Coq. Ensuite, les différentes documentations de Coq (le manuel de référence (Coq Development Team, 2002) ou l’ouvrage (Bertot *et al.*, 2004)) peuvent et doivent prendre le relais pour compléter la formation.

Dans le cadre du cours d’introduction à la sémantique des langages de programmation, cette expérience s’est avérée positive. L’ensemble des étudiants est parvenu à formaliser avec Coq l’intégralité du cours. Mais plus fondamentalement (et c’était là l’objectif), ils ont pu appréhender les notions jugées initialement trop abstraites du cours puisqu’ils ont du développer (modulairement) des preuves de correction. Concernant le projet Coq réalisé, on a pu s’apercevoir qu’un enseignement Coq utilisateur (c’est-à-dire par la pratique) était plus que satisfaisant pour développer des applications réelles d’envergure moyenne, même si la non-connaissance de la théorie

sous-jacente impose quelques limitations très ponctuelles (relatives notamment à l'extraction ou à la récursion bien fondée). Enfin, les travaux réalisés dans l'atelier Focal ont permis de mettre en évidence la faisabilité de preuves dans un environnement plus complexe que dans le cours de sémantique ou même le projet. Les étudiants ont également pu s'approprier la sémantique d'un certain nombre de concepts orienté-objet présents dans le langage Focal. En outre, cette expérience a constitué un test important pour l'utilisation du langage, qui a conduit à une réflexion poussée sur les méthodes de développement d'un programme Focal. Dans tous les cas, nous avons pu constater que l'utilisation de Coq dans ces cours a beaucoup accru l'intérêt des étudiants. D'autre part, et c'est pour nous l'essentiel, nous avons remarqué une très nette amélioration du niveau des copies de l'examen écrit (qui ne fait pas du tout appel à Coq), ce qui montre le réel apport pédagogique de l'utilisation de Coq pour la compréhension des preuves.

Les résultats positifs de ces expériences nous conduisent à étendre l'utilisation de Coq dans d'autres cours. Par exemple, cette année (2004-2005) dans le cours de logique de première année de master d'informatique, Coq est utilisé pour illustrer les notions de preuves, d'élimination des coupures et de normalisation de lambda-termes.

#### Remerciements

Nous remercions vivement Véronique Donzeau et Thérèse Hardin, responsables du DESS "Développement de Logiciels Sûrs", pour nous avoir permis de réaliser ces enseignements dans le cadre du DESS, ainsi que pour toutes les discussions fructueuses que nous avons eues ensemble au sujet de ces cours. Nous remercions également Damien Doligez, Catherine Dubois et Renaud Rioboo qui ont participé à ces discussions, ainsi que les relecteurs anonymes d'une première version de cet article pour leurs précieux commentaires. Enfin, nous remercions bien sûr l'ensemble des étudiants du DESS qui ont accepté de "jouer le jeu" !

## 6. Bibliographie

- Abrial J. R., *The B-Book : Assigning Programs to Meanings*, Cambridge University Press, 1996.
- Bertot Y., Capretta V., Barman K. D., « Type-theoretic functional semantics », in V. A. Carreno, C. A. Munoz, S. Tahar (eds), *Theorem Proving in Higher Order Logics*, vol. 2410 of LNCS, Springer, p. 83-97, 2002. 15th International Conference, TPHOLS 2002, Hampton, VA, USA, August 20-23, 2002.
- Bertot Y., Castéran P., *Le Coq'Art*, Springer-Verlag, 2004.
- Boulmé S., Hardin T., Hirschhoff D., Ménissier-Morain V., Rioboo R., « On the way to certify Computer Algebra Systems », *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conf.)*, vol. 23 of ENTCS, Elsevier, 1999.
- Coq Development Team, *The Coq Proof Assistant Reference Manual Version 7*, INRIA-Rocquencourt. 2002.

- Delahaye D., Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve : une Étude dans le cadre du système Coq, Thèse de doctorat, Université Paris 6, 2001.
- Focal development team, *A brief FoC tutorial*, LIP6 – INRIA – CNAM. March, 2003.
- Focal development team, *Focal reference manual, version 0.2beta*, LIP6 – INRIA – CNAM. October, 2004, <http://focal.inria.fr>.
- Griffioen D., Huisman M., « A Comparison of PVS and Isabelle/HOL. », in J. Grundy, M. C. Newey (eds), *Theorem Proving in Higher Order Logics, 11th Int. Conf., TPHOLS'98 Proceedings*, vol. 1479 of LNCS, Springer, p. 123-142, 1998.
- Jakubiec L., Coupet-Grimal S., Curzon P., « A comparison of the Coq and HOL proof systems for specifying hardware », *Supplementary Proc. of the 10th Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLS'97*, p. 63-78, 1997.
- Meyer B., « Applying "Design by Contract" », *Computer*, 1992.
- OCaml Development Team, *OCAML Reference Manual Version 3.08*, INRIA-Rocquencourt. 2004.
- Prevosto V., Conception et Implantation du langage FoC pour le développement de logiciels certifiés, Thèse de doctorat, Université Paris 6, September, 2003.
- Prevosto V., Doligez D., « Algorithms and Proof Inheritance in the FoC language », *Journal of Automated Reasoning*, vol. 29, n° 3-4, p. 337-363, December, 2002a.
- Prevosto V., Doligez D., Hardin T., « Algebraic Structures and Dependent Records », in C. Muñoz, S. Tahar, V. Carreño (eds), *TPHOLS : 15th Int. Conf. on Higher Order Logic Theorem Proving*, vol. 2410, LNCS, Springer-Verlag, August, 2002b.
- Prevosto V., Jaume M., « Making Proofs in a hierarchy of Mathematical Structures », *Proceedings of the 11th Calculemus Symposium*, September, 2003.
- Rioboo R., Programmer le Calcul Formel, des Algorithmes à la Sémantique, Mémoire d'habilitation, Université Paris 6, 2002.

Article reçu le 28 mai 2004  
Version révisée le 26 avril 2005

**David Delahaye** est maître de conférences au CNAM. Ses activités de recherche portent sur les preuves formelles en théorie des types et la conception des outils d'aide à la preuve.

**Mathieu Jaume** est maître de conférences au LIP6. Ses activités de recherche portent sur les spécifications/preuves formelles, la sémantique des langages et la sécurité.

**Virgile Prevosto** est polytechnicien (X96) et docteur en informatique. Ses recherches actuelles au Max-Planck Institut portent sur la conception d'outils pour les méthodes formelles de développement.