Pierre Courtieu

Né le 14 juillet 1973 (29 ans)
Libéré des obligations militaires

Adresse professionnelle	Adresse personnelle
Projet Lemme	130 route de la reine
INRIA Sophia-Antipolis	92100 Boulogne Billancourt
2004 Route des Lucioles	
BP 93, 06902 Sophia Antipolis Cedex	
Tél. : (+33/04) 92 38 71 97	06 22 34 42 38
Mél : `courtieu@lri.fr`	

Position actuelle

Stage post-doctoral 2001-2003 INRIA Sophia-Antipolis, projet *LEMME*

Cursus

ATER (demi-poste)	2000-2001	Université Paris sud	
Doctorat en Informatique	1997-2001	Université Paris sud	mention très honorable

Au sein de l'équipe *démonstration, programmation* du L.R.I.
Sous la direction de Laurence Puel
"Représentation d'algèbres non libres en théorie des types"
Soutenue le 14 décembre 2001

Service national	1996-1997		
DEA d'informatique	1995-1996	Université Paris sud	mention AB
Maîtrise d'informatique	1994-1995	Université Paris sud	mention B
Licence d'informatique	1993-1994	Université Paris sud	mention AB
DEUG A Math & Info	1991-1993	Université Paris sud	mention AB

Enseignement

J'ai participé, en qualité d'enseignant vacataire de 1997 à 2000 et comme (demi-)ATER de 2000 à 2001, aux différents modules d'enseignement ci-dessous. Au total j'ai effectué environ 400H heures d'enseignement sur 4 ans. J'ai participé à la rédaction des sujets de TD, de projets et d'examens. J'ai bien entendu également participé au travail de correction des copies.

Lors du premier semestre 1999, j'étais chargé de l'unique groupe du module "travail d'étude et de recherche[1]" de la licence d'informatique d'Orsay (le cours était assuré par Laurent Rosaz), ce qui m'a laissé une certaine autonomie concernant la gestion du calendrier, du suivi des étudiants, des soutenances et de la correction des rapports. J'ai également assumer la responsabilité d'un groupe d'étudiants de MIAGE dans leur module "étude de cas", en faisant les cours, TD et TP, sous la responsabilité de Ralf Treinen.

Tous ces enseignements ont été effectués à l'université Paris-sud (Orsay), sauf l'analyse, effectuée actuellement à l'université de Nice.

Principes d'interprétation des langages (DEUG, 2 semestres, TP, 30H).

Introduction à l'analyse syntaxique, analyse sémantique, Lex, Yacc.

Étude de cas Langage et génie logiciel (MIAGE 1ère année, Cours+TD+TP, 1 semestre, 50H).

Introduction à l'analyse syntaxique, analyse sémantique, Lex, Yacc, réalisation d'un navigateur HTML.

T.E.R. Algorithmique[1] (Licence d'informatique, T.E.R., 5 semestres, 200H)

Apprentissage de l'algorithmique, introduction à la notion de complexité, programmation en langage C, gestion des pointeurs et de la mémoire, réalisation d'un programme de traitement et d'affichage d'images.

Logique (MIAGE 2ème année, TD, 1 semestre, 32H)

Formule, interprétation, régle de déduction, système de déduction, Calcul des séquents, déduction naturelle.

T.E.R. Génie Logiciel[1] (Licence d'informatique, T.E.R., 1 semestre, 42H)

Introduction à la notation UML, réalisation d'un programme de simulation autoroutière en JAVA.

Analyse (Licence, 1 semestre, TD-TP, 50H).

Introduction à l'analyse syntaxique, analyse sémantique, Lex, Yacc.

[1] T.E.R. : Travail d'étude et de recherche, il s'agit d'un projet réalisé tout au long du semestre, le suivi des étudiants est effectué au cours de réunions hebdomadaires avec le chargé de T.E.R.

Recherche

Les travaux de recherche décrits ici ont été réalisés dans l'équipe démons du L.R.I. à Orsay, sous la direction de Laurence Puel, et depuis septembre 2001 à l'I.N.R.I.A Sophia-Antipolis dans le projet LEMME, sous la direction de Gilles Barthe.

À l'occasion d'un stage de DEA[Cou97], nous avons étudié et étendu le travail de Val Tannen, Delia Kesner et Laurence Puel sur le *λ-calcul typé avec motif* [BTKP93]. L'extension consistait à ajouter au langage des termes et au typage de ce calcul une quantification universelle correspondant au polymorphisme. Ensuite au cours de ma thèse [Cou01b] nous avons abordé un thème radicalement différent ayant trait à la preuve formelle en théorie des types et au système d'aide à la preuve Coq. Ce sont ces aspects plus récents de notre travail qui sont exposés ci-dessous.

Nous nous sommes intéressé de manière générale à la preuve formelle assistée par ordinateur. Plus particulièrement (1) nous avons fourni un travail théorique sur l'outil d'aide à la démonstration Coq [Coq] et son système logique : la théorie des types ; (2) nous avons appliqué Coq à différents domaines de certification d'algorithmes et de programmes. Nous donnons ici une liste de nos principales contributions :

- **Extension du calcul des constructions inductives** ($\mathcal{CCI}$) [PM96, CPM90]. Un travail théorique sur le système de types utilisé dans Coq pour représenter les preuves. Cette extension, présenté dans ma thèse, a pour but de permettre une représentation des quotients plus satisfaisante que dans le système actuel, à l'aide d'une nouvelle notion : les *types normalisés*. Un type normalisés se construit à l'aide d'un type A et d'une fonction nf sur ce type. Intuitivement le type ainsi construit, noté $\mathsf{Norm}(A, nf)$, représente le quotient du type A par la relation R_{nf} définie par $(R_{nf}\ x\ y) \Leftrightarrow (nf\ x) = (nf\ y)$. Nous avons montré que cette égalité peut être incluse dans la conversion du système sans compromettre les propriétés indispensables de $\mathcal{CCI}$: normalisation forte, cohérence logique, décidabilité de la conversion et donc du typage, etc.
- **Preuves d'auto-stabilisation dans Coq**. Également dans ma thèse, il s'agit d'un développement d'une nouvelle théorie dans l'outil Coq lui-même. Plus précisément nous avons formalisé la notion d'algorithme auto-stabilisant ainsi qu'une méthode de preuve d'auto-stabilisation . Un algorithme auto-stabilisant est un algorithme réparti ayant la propriété suivante : à partir d'une configuration quelconque du réseau, l'algorithme le ramène dans une configuration «légitime» (correcte) au bout d'un temps fini.
- **Certification de la plate-forme JavaCard**. Plus récemment au cours de mon stage post-doctoral, commencé en septembre 2001 et prévu jusqu'à août prochain, nous avons développé de nouveaux outils dans le cadre du projet européen VerifiCard [IST] sur la certification *formelle* de la plate-forme Javacard. Javacard est un sous-ensemble de Java utilisé pour la programmation des cartes à puces de prochaine génération et dont la machine virtuelle est embarquée sur les cartes. Dans cette perspective nous avons fourni plusieurs outils :
  - De nouvelles tactiques de preuve pour Coq permettant notamment d'effectuer des démonstrations par analyse de cas et par récurrence en suivant le schéma donné par la définition d'une fonction. Cette contribution a été intégrée à la distribution de Coq.
  - Une technologie de certification (génération automatique de lemmes intermédiaires, preuve automatique de ces lemmes) dédiés à la preuve de propriétés de spécifications écrites dans les langages de bas niveau (c'est-à-dire du type machine virtuelle Javacard). Cette technologie

est basée sur l'outil *Jakarta* [BDHdS01] développé dans le projet LEMME et dont nous avons par ailleurs repris le développement.
Jakarta est un environnement dédié à la certification de *vérificateurs de bytecode* (BCV), en particulier pour le bytecode Javacard. Le BCV est un élément clé de la sécurité de l'architecture Javacard, et doit faire l'objet de certification de très haut niveau. Jakarta permet de spécifier une machine virtuelle, de générer semi-automatiquement le vérificateur de bytecode associé et la preuve de sa correction (dans Coq par exemple). Associée aux tactiques citées plus haut, elle permet de fournir à l'utilisateur un cadre très automatisé pour la certification de machines virtuelles/BCV.

Ces contributions sont à présent intensivement utilisées dans le projet *Certicartes*.

- **ProofGeneral**. Enfin on m'a confié depuis quatre ans la responsabilité du développement de la partie Coq de `ProofGeneral` [Asp], un mode `Emacs` permettant d'utiliser plusieurs assistants de preuve comme Coq, `Isabelle`, `Lego`, etc, et qui est l'interface la plus utilisée aujourd'hui pour Coq. Ce travail consiste à synchroniser ce mode avec les différentes versions de Coq, à implanter de nouvelles fonctionnalités, corriger les bugs éventuels, etc.

Les trois premiers points ci-dessus sont détaillés dans la suite.

Les structures non-libres Une algèbre de termes non-libre est une algèbre dans laquelle il existe des relations d'égalité entre les constructeurs. Les méthodes de raisonnement en théorie des types sur de telles structures sont encore mal comprises. Elles posent en effet un problème délicat en présence de types inductifs : introduire des égalités entre constructeurs d'un type inductif entraîne systématiquement dans le calcul des constructions inductives ($\mathcal{CCI}$) une incohérence. Par conséquent c'est une opération interdite dans Coq. La définition de quotients en Coq se limite donc aux types non inductifs, ce qui ne permet pas un comportement calculatoire satisfaisant.

Plusieurs solutions ont déjà été proposées pour ce problème, notamment par Gilles Barthe et Herman Geuvers [BG95], ainsi que par Martin Hofmann [Hof95b, Hof95a] et Samuel Boutin [Bou97]. Ces différentes propositions ne résolvent que partiellement le problème du comportement calculatoire. Dans ma thèse il est proposé une extension du calcul des constructions inductives permettant un traitement plus agréable et surtout plus puissant de la notion de quotient.

L'incohérence engendrée par les équations entre constructeurs est causée par le principe de calcul associé aux types inductifs. Ce principe de calcul est caractérisé par deux notions complémentaires : le *schéma de récurrence* et la réduction associée, appelée ι-réduction. La définition de la ι-réduction est telle que la seule sémantique cohérente d'un type inductif correspond à une structure *libre*. La seule manière de définir des structures non libres est donc d'utiliser une *axiomatisation* des quotients. Cette solution présente deux inconvénients : premièrement elle exclut que l'égalité du quotient soit ajoutée à la conversion du système, et deuxièmement elle ne permet pas d'exploiter les propriétés calculatoires des types inductifs.

Pour permettre une définition plus "calculatoire" des quotients, nous définissons une nouvelle notion : les *types normalisés*. Un type normalisés se construit à l'aide d'un type A et d'une fonction nf sur ce type. Intuitivement le type ainsi construit, noté $\mathsf{Norm}(A, nf)$, représente le quotient du type A par la relation R_{nf} définie par $(R_{nf}\ x\ y) \Leftrightarrow (nf\ x) = (nf\ y)$. Nous avons montré que cette égalité peut être incluse dans la conversion du système sans compromettre les propriétés de $\mathcal{CCI}$: normalisation forte, cohérence logique, décidabilité de la conversion et donc du typage, etc.

Une des caractéristiques originale de notre formalisme est que la conversion n'est pas, comme c'est le cas en général, la clôture congruente de la relation de réduction. On justifie ce principe par l'exemple du quotient $\mathbb{Z}/3\mathbb{Z}$: dans ce quotient, 0 et 3 doivent être considérés comme convertibles, mais il n'est pas souhaitable que 3 se réduise vers 0. Pour rendre opérationnelle cette notion de conversion, on définit néanmoins une *interprétation* des termes, qui permet de décider si deux termes sont convertibles en calculant leurs images par l'interprétation, puis en comparant les résultats à l'aide d'une réduction, confluente, plus classique.

On rejoint par là l'idée initiale de Per Martin-Löf et sa théorie des types [ML84], dans laquelle on distingue l'égalité de jugement de l'égalité définitionnelle. On rejoint également l'idée de Martin Hofmann [Hof95a] d'une égalité après interprétation.

Nous avons présenté ce travail au Workshop TYPES'2000, puis à la conférence CSL'2001 [Cou01a].

Démonstration d'auto-stabilisation d'algorithmes répartis Le deuxième thème abordé dans ma thèse est un nouveau champ d'application de l'assistant de preuve Coq : les algorithmes répartis. Plus précisément nous avons implanté dans Coq une méthode de preuve d'algorithme auto-stabilisant tirée de la littérature récente. Un algorithme auto-stabilisant est un algorithme *réparti* (c'est-à-dire s'exécutant sur un réseau) ayant la propriété suivante : à partir d'une configuration initiale quelconque du réseau, l'algorithme fait revenir celui-ci dans une configuration correcte (légitime) au bout d'un temps fini et ne quitte plus les configurations correctes ensuite. On entend par *configuration correcte* une configuration correspondant à un déroulement valide de l'algorithme. Par exemple dans le cas d'un algorithme de passage d'un jeton dans un réseau, les configurations légitimes sont celles dans lesquelles il n'y a qu'un, et un seul, jeton sur le réseau.

La méthode de preuve d'auto-stabilisation formalisée est dûe à Beauquier, Bérard, Fribourg et Magniette, et permet de prouver d'une manière semi-systématique l'auto-stabilisation d'une certaine classe d'algorithmes s'exécutant sur un réseau linéaire ou un anneau, c'est-à-dire sur un réseau dans lequel un nœud a au plus deux voisins. Cette topologie permet de représenter les configurations du réseau par des mots sur l'alphabet des états des machines, afin d'utiliser des techniques de réécriture, notamment le "narrowing", pour prouver la convergence des algorithmes considérés.

Pour démontrer la convergence de toutes les exécutions possibles d'un algorithme vers l'ensemble des configurations légitimes, on va s'intéresser à un sous-ensemble des réductions possibles, dites *réductions actives*. On voit ici apparaître une méthode en deux parties :

- Premièrement on montre une fois pour toutes que sous certaines conditions cette restriction aux réductions actives se fait sans perte de généralité. C'est la partie *générique* de la preuve ;
- Deuxièmement pour chaque algorithme réparti considéré on va démontrer la convergence des réductions actives. C'est la partie *spécifique* de la preuve.

Nous avons formalisé la partie générique de la preuve dans Coq sous la forme d'une librairie réutilisable pour chaque algorithme étudié. Nous avons dû, pour cela, d'une part développer puis utiliser un ensemble de résultats sur la réécriture des mots et d'autre part définir formellement dans Coq la notion d'algorithme réparti et d'auto-stabilisation.

Nous avons également formalisé la partie spécifique sur un algorithme auto-stabilisant standard. Cette partie consistait donc à démontrer la convergence des réductions actives vers l'ensemble des configurations légitimes, puis à appliquer le résultat prouvé dans la partie générique.

Ce développement est le premier pas vers un environnement de démonstration certifié dédié à la preuve d'auto-stabilisation, sujet du projet BQR auquel j'ai participé avec Laurence Puel, Joffroy Beauquier et Frédéric Magniette.

Ce travail a été présenté à la conférence FMPPTA'2002 [Cou02].

Cartes à puces - Formalisation, vérification, preuves Actuellement (septembre 2001-août 2003) en post-doc à l'I.N.R.I.A. Sophia-Antipolis dans le projet LEMME (groupe de Loïc Pottier) sous la direction de Gilles Barthe, je participe au projet européen *VerifiCard*[IST] dont le but est la certification par des méthodes formelles de la plate-forme *JavaCard*. *JavaCard* est le langage de programmation, dérivé de Java, qui sert à la programmation de la prochaine génération de cartes à puces.

Sur ces cartes seront embarquées une *machine virtuelle Javacard*, ainsi qu'une ou plusieurs applications : des *applets* Javacard permettant, par exemple, de gérer un porte-monnaie électronique, un service de paiement sécurisé, un accès sécurisé à un réseau d'assurance médicale, etc. Ces applications pourront éventuellement être téléchargées après achat de la carte.

La présence de la machine virtuelle et la cohabitation de différentes applications sur une même carte posent de nombreux problèmes de sécurité, de confidentialité des données, de détection d'applets frauduleuses... Le milieu industriel est demandeur d'une méthodologie permettant de s'assurer de certaines propriétés de la plate-forme et/ou des applets.

Vu la complexité des propriétés demandées et le nombre de configurations possibles, il est rapidement apparu que la preuve formelle était la meilleure approche pour nombre de ces problèmes. C'est le thème du projet VerifiCard. Parmi les différents travaux issus de ce thème, le projet CertiCartes consiste à formaliser entièrement la machine virtuelle Javacard et un *vérificateur de bytecode* (BCV) dans l'assistant de preuve Coq, et de prouver des propriétés de correction du vérificateur par rapport à la machine virtuelle. Ce vérificateur permet de s'assurer, au moment d'un téléchargement d'applet, de certaines propriétés du code téléchargé (bon typage, initialisation, confinement etc).

Dans le cadre de ce projet, nous avons conçu une tactique de preuve Coq permettant d'effectuer des raisonnements par cas et par récurrence automatiquement, en suivant le schéma donné par une fonction récursive. On peut ainsi effectuer des raisonnements sur le résultat d'une fonction de manière efficace et concise. Cette tactique est maintenant utilisée dans les démonstrations du projet CertiCartes dans lequel il est fait une utilisation massive de fonctions par cas pour formaliser la machine virtuelle. Ce travail a été présenté à la conférence TPHOLs 2002[BC02], et a été intégré dernièrement à la distribution standard de Coq (version non encore distribué mais disponible par CVS anonyme).

Dans ce même projet, nous participons à l'élaboration de l'outil Jakarta[BDHdS01, Sou03], dédié à la certification de BCV et basé sur la méthodologie utilisé dans CertiCartes. Jakarta permet de spécifier une machine virtuelle (*de référence*) dans un langage simple et exécutable, puis d'engendrer à partir de celle-ci :

- Le BCV associé pour une propriété P donnée, obtenu par des techniques d'abstraction syntaxique [BDHdS01] à partir de la machine de référence puis par des techniques d'analyse de flot de donnée, décrites par exemple dans [Ler01].
- La preuve formelle (dans Coq) de la correction de ce BCV. C'est-à-dire la preuve que si le BCV accepte une applet, alors celle-ci a bien la propriété P voulue. Nous avons conçu et implémenter un mécanisme de génération de lemmes intermédiaires fonctionnant en collaboration avec le moteur d'abstraction. Les résultat obtenu sont très concluant : environ 80% des preuves sont maintenant complètement automatiques, notamment grâce aux tactiques de raisonnement décrites plus haut. La plus grande part des preuves nécessaires concerne la correction de l'abstraction, qui se résume par la commutation des machines de référence et abstraite selon le diagramme ci-dessous (où α dénote l'abstraction syntaxique permettant de construire le BCV) :

$$\begin{tikzcd}
\texttt{état} \arrow[r, "\texttt{exécution de référence}"] \arrow[d, "\alpha"] & \texttt{état} \arrow[d, "\alpha"] \\
\texttt{état abstrait} \arrow[r, "\texttt{exécution abstraite}"] & \texttt{état abstrait}^{\star}
\end{tikzcd}$$

J'ai participé, à l'implémentation du moteur d'abstraction de Jakarta et j'en suis aujourd'hui le principal développeur, et j'ai conçu le mécanisme de génération et de preuve des lemmes intermédiaires. Ces travaux font l'objet d'un article de journal en cours de rédaction.

Responsabilités collectives

je suis le responsable depuis trois ans de la maintenance de la partie `Coq` de `ProofGeneral` [Asp], un mode `Emacs` permettant d'utiliser plusieurs assistants de preuves comme `Coq`, `Isabelle`, `Lego`... Mon travail consiste à synchroniser ce mode avec les différentes versions de `Coq`, à implanter de nouvelles fonctionnalités, etc.

Perspectives de recherche

Le thème de recherche que je me propose de suivre est globalement de développer l'usage des assistants de preuve pour la certification de systèmes. L'objectif à moyen ou long terme est de faire de l'assistant de preuve un outil standard de l'ingénieur-informaticien pour la conception de système sûrs. Différents obstacles entravent encore le chemins, certains théoiques, d'autres pratiques.

Il convient pour cela d'appliquer les assistants de preuve à des domaines nouveaux, améliorer les assistants de preuve d'un point de vue théorique (logique, λ-calcul...) et pratique (interface, tactiques de raisonnement...), et enfin développer des outils dédiés à des domaines particulier de certification, comme Jakarta, connecté à un assistant de preuve.

Nous énumérons rapidement ci-dessous différents thèmes de recherche que nous envisageons. Il ne s'agit bien évidemment pas d'une liste exhaustives ou exclusive, mais plutôt d'une description de directions qui semblent prometteuses.

- *Certification d'algorithmes, de logiciels* (Javacard et autres), sémantique des langages. Au niveau du code source (logique de Hoare...) ou au niveau du bytecode comme nous l'avons déjà fait dans le projet VerifiCard. En particulier participer au développement de CertiCartes et Jakarta en collaboration avec le projet LEMME à l'INRIA Sophia-Antipolis. Rechercher de nouveaux domaines d'applications des assistants de preuve, notamment en relation avec d'autres méthode de spécification/vérification (spécification algébriques, test, model checking).
- *Nouveaux développements théoriques autour du calcul des constructions inductives* ($\mathcal{CCI}$) permettant une meilleure représentation des structures mathématiques (et informatiques) dans les assistants de preuve de type `Coq`. En effet les choix de représentation des structure conditionnent le déroulement des preuves de manière trop sensible aujourd'hui, rendant cette activité très difficile à un non expert. Des techniques du type de celles que j'ai utilisées dans ma thèse semble pouvoir s'appliquer à d'autre extension de $\mathcal{CCI}$. L'idée principale étant d'expérimenter ces différentes extensions dans des versions modifiées de `Coq`.
- *Développement de `Coq`, tactiques de preuves complexes.* Il semble que des tactiques "à grands pas" – c'est-à-dire effectuant des étapes de raisonnement complexes mais *ne résolvant pas nécessairement complètement une preuve* – permettraient d'automatiser les démonstrations de manière satisfaisante. C'est dans cet esprit que nous avons déjà développé des tactiques de récurrence dans le projet *Verificard*, nous comptons continuer dans cette voie.

– *Ergonomie, interface des systèmes de preuve.* Notre expérience des deux environnements de preuve `Proof General` et `Pcoq` nous permet de penser que l'association de leurs principes peut mener à un outil plus puissant et ergonomique. Adjoindre des modalités d'interaction issues des recherches récentes en interaction homme-machine semble également très prometteuse. Notamment l'utilisation des modalités d'interaction de *zoom*, d'*interaction bimanuelle*, et l'exploitation plus systématique de l'*hypertexte* semblent pouvoir améliorer de manière très significative l'ergonomie des système de preuve. Ces modalités, et d'autres qui restent à inventer, devront être adaptées à la tâche très spécifique de construction de preuve, comme l'a été avec succès l'utilisation de la souris pour donner le célèbre *proof by pointing*.
– *Création d'un outil de génération automatique de lemmes*, dans le but de permettre l'utilisation d'outils externes standardisés de génération d'obligations de preuve. Ceci dans la perspective d'un usage des assistant de preuve dans les preuves de grande taille. Les outils externes de génération d'obligations de preuves sont déjà nombreux (`Jakarta`, `why` [Fil]), et le besoin d'un outil générique adapté à cette tâche se fait aujourd'hui pressant. Il s'agira de définir un langage permettant (1) d'exprimer les obligations de preuve, (2) de décrire éventuellement comment ces obligations doivent être prouvées afin que les obligations de preuve soient accompagnées d'un script de preuve, (3) décrire comment les obligations, une fois prouvées, doivent être utilisées dans les preuves suivantes. Cet outil devra également être indépendant de l'assistant de preuve utilisé.

Références

[Asp] David Aspinall. ProofGeneral. `http ://www.proofgeneral.org`.

[BC02] G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.

[BDHdS01] G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta : a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.

[BG95] G. Barthe and H. Geuvers. *Congruence types.* In H. Kleine Buening, editor, *Proc. Conf. Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 36–51, 1995.

[Bou97] Samuel Boutin. *Réflexions sur les quotients.* Thèse d'université, Paris 7, April 1997.

[BTKP93] Val Breazu-Tannen, Delia Kesner, and Laurence Puel. A typed pattern calculus. In *Proceedings of the Eigth Annual IEEE Symposium on Logic in Computer Science*, pages 262–274. IEEE Comp. Soc. Press, 1993.

[Coq] The Coq Proof Assistant. `http ://coq.inria.fr/`.

[Cou97] Pierre Courtieu. Extension au second ordre du λ-calcul typé avec motifs. Master's thesis, Université Paris XI, 1997. `http ://www.lri.fr/~courtieu/biblio.html`.

[Cou01a] Pierre Courtieu. Normalized types. In *Proceedings of CSL2001*, volume 2142 of *LNCS*, 2001.

[Cou01b] Pierre Courtieu. *Représentation de structures non libres en théorie des types.* Thèse de doctorat, Université Paris-sud, Laboratoire de Recherche en informatique, December 2001. `http ://www.lri.fr/~courtieu/biblio.html`.

[Cou02] Pierre Courtieu. Proving self-stabilization with a proof assistant. In *Proceedings of IPDPS'2002*, IEEE CS Press, 2002.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[Fil] J.-C. Filliatre. *why*. `http ://why.lri.fr/index.fr.html`.

[Hof95a] M. Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.

[Hof95b] M. Hofmann. A simple model for quotient types. In *proc TLCA'95*, volume 902 of *lncs*. Springer-Verlag, 1995.

[IST] IST. Projet Européen Verificard. `http ://www.verificard.com/`.

[Ler01] X. Leroy. Java bytecode verification : an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.

[ML84] Per Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[PM96] Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. thèse d'habilitation, Université Claude Bernard Lyon I, Décembre 1996.

[Sou03] S. Melo de Sousa. *Outils et techniques pour la vérification formelle de la plateforme JavaCard*. PhD thesis, Université de Nice Sophia-Antipolis, February 2003.