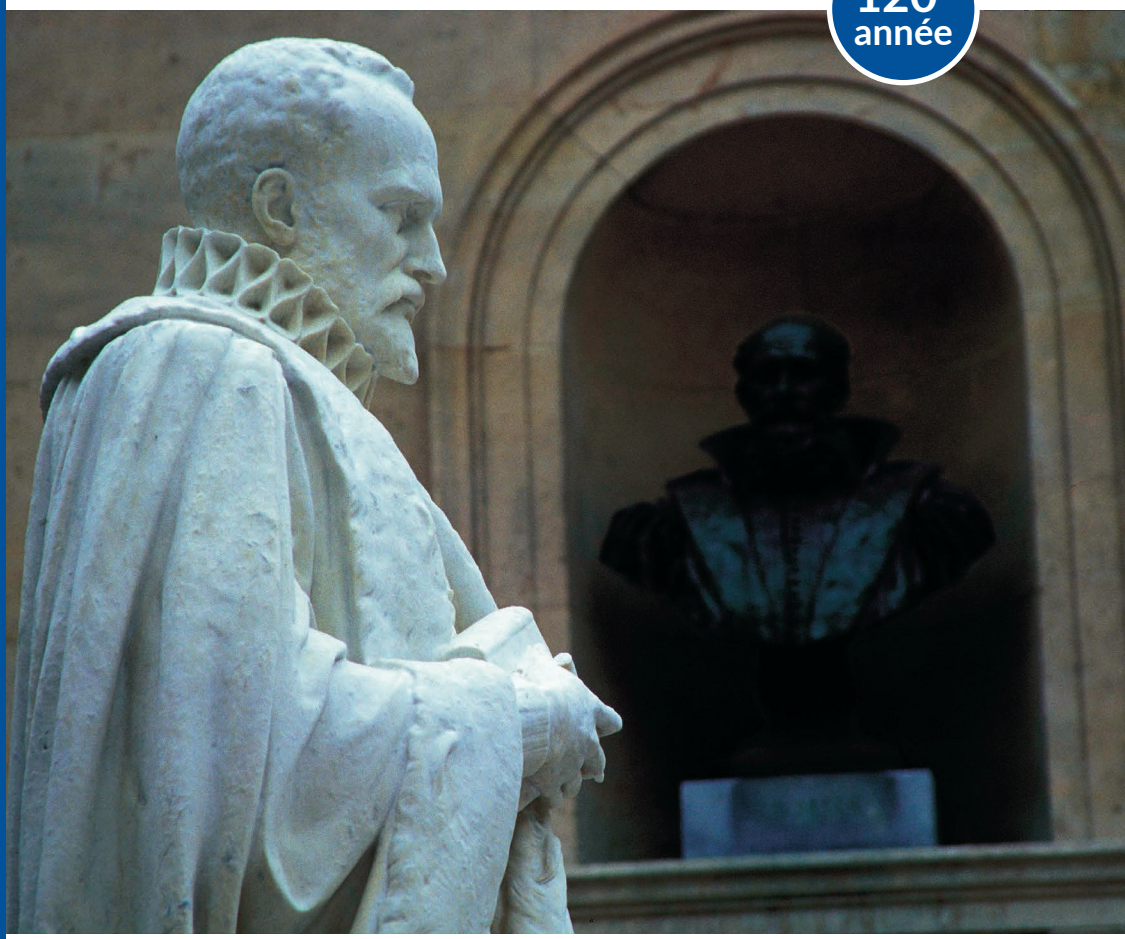
ANNUAIRE du **COLLÈGE DE FRANCE** 2019 - 2020

Résumé des cours et travaux

120e année



COLLÈGE DE FRANCE
1530

Sciences du logiciel

Xavier Leroy
Professeur au Collège de France

Mots-clés : informatique, logiciel, langages, sémantique, types, logique, démonstration, vérification, méthodes formelles

La série de cours et de séminaires « Sémantiques mécanisées : quand la machine raisonne sur ses langages » est disponible en audio et vidéo, sur le site internet du Collège de France (https://www.college-de-france.fr/site/xavier-leroy/course-2019-2020.htm).

Cours – Sémantiques mécanisées : quand la machine raisonne sur ses langages

« Que fait ce programme, au juste ? » Pour répondre à cette question avec la précision des mathématiques, il nous faut une sémantique formelle du langage dans lequel ce programme est écrit. Plusieurs approches de la sémantique formelle sont bien maîtrisées aujourd'hui : sémantiques dénotationnelles, qui interprètent le programme comme un élément d'une structure mathématique ; sémantiques opérationnelles, qui décrivent les étapes successives des exécutions du programme ; sémantiques axiomatiques, qui décrivent les assertions logiques satisfaites par ces exécutions. Ces sémantiques formelles ont de nombreuses applications : non seulement définir les langages de programmation avec une précision bien plus grande qu'un manuel de référence écrit en français ou en anglais, mais aussi vérifier la correction des algorithmes et des outils qui opèrent sur les programmes, comme les compilateurs, les analyses statiques, et les logiques de programmes.

La sémantique d'un langage de programmation peut être volumineuse et complexe, rendant les démonstrations « sur papier » utilisant ces sémantiques pénibles et peu

X. Leroy, « Sciences du logiciel », *Annuaire du Collège de France 2019-2020. Résumé des cours et travaux*, 120e année, Paris, Collège de France, 2023, p. 13-22, https://doi.org/10.4000/annuaire-cdf.18052.

fiables. Mais nous pouvons nous adjoindre l'aide de l'ordinateur ! Les assistants à la démonstration (Coq, HOL, Isabelle, etc.) fournissent un langage rigoureux pour définir les sémantiques, énoncer leurs propriétés, écrire des démonstrations, et vérifier automatiquement la cohérence de ces démonstrations. Cette mécanisation est un puissant levier pour faire passer les approches sémantiques à l'échelle des langages et des outils de programmation réalistes.

Le cours a présenté cette approche de sémantique mécanisée sur l'exemple de petits langages impératifs ou fonctionnels, avec des applications aux logiques de programmes et à la vérification de compilateurs et d'analyseurs statiques. Toutes ces notions ont été entièrement mécanisées avec l'assistant Coq. Le développement Coq est disponible à l'adresse https://github.com/xavierleroy/cdf-sem-meca.

Le séminaire a approfondi l'approche dans plusieurs directions, allant de la mécanisation de langages « du monde réel » comme Javascript et Rust à l'utilisation d'assistants à la démonstration pour enseigner les fondements des langages de programmation.

Cours 1 – Des expressions et des commandes : la sémantique d'un langage impératif

Le premier cours a débuté par un bref historique des sémantiques formelles pour les langages de programmation. Les premiers langages de programmation de haut niveau font apparaître des points délicats de sémantique : le passage d'arguments aux fonctions en Algol 60, « par nom » ou bien « par valeur », et la liaison des variables en Lisp, avec sa sémantique de liaison dynamique qui sera remplacée plus tard par la liaison statique.

En partie pour éclaircir ces points délicats, en partie pour faciliter le raisonnement sur les programmes, les premières sémantiques formelles apparaissent dès le milieu des années 1960, suivant trois styles différents qui sont toujours bien vivants aujourd'hui : les sémantiques opérationnelles, avec les travaux précurseurs de Landin en 1964 et beaucoup d'utilisations depuis 1990 ; les sémantiques axiomatiques, introduites par Floyd en 1967, et développées par Hoare, Dijkstra et coauteurs pendant les années 1970 ; les sémantiques dénotationnelles, introduites sous une forme simplifiée par Strachey vers 1965 puis mathématiquement fondées sur la théorie des domaines par Scott et coauteurs entre 1970 et 1990.

Ensuite, le cours a introduit le petit langage de programmation IMP qui a servi d'objet d'étude dans la plupart des cours de cette année. C'est un langage impératif, structuré en expressions arithmétiques, expressions booléennes, et commandes qui modifient des variables de type entier. Les expressions se prêtent merveilleusement à une sémantique dénotationnelle « naïve » à la manière de Strachey. Les commandes pouvant ne pas terminer, leur sémantique dénotationnelle est plus délicate ; c'est pourquoi nous avons adopté des sémantiques opérationnelles, à réductions comme dans le λ-calcul, ou bien naturelles, ou encore par interpréteur de référence.

Enfin, le cours a motivé le besoin de mécaniser les sémantiques de langages réalistes, c'est-à-dire de s'aider de la puissance de la machine pour écrire ces sémantiques et raisonner à leur propos. Comme exemple de cette démarche, nous avons montré comment mécaniser la syntaxe abstraite et les sémantiques opérationnelles du langage IMP à l'aide de l'assistant à la démonstration Coq. Nous avons finalement démontré des résultats d'équivalence entre la sémantique à réduction, la sémantique naturelle, et l'interpréteur de référence.

COURS 2 – *TRADUTTORE, TRADITORE* : VÉRIFICATION FORMELLE D'UN COMPILATEUR

Le deuxième cours a porté sur la compilation : la traduction automatique d'un langage de programmation de haut niveau en code exécutable par la machine. Accompagnant l'évolution des langages de programmation, de nombreux algorithmes de compilation et d'optimisation des programmes ont été développés, transformant le code pour en augmenter les performances. Toutes ces transformations ne sont pas sans risque : une erreur dans le compilateur peut lui faire produire du code exécutable faux à partir d'un programme source correct ; on parle alors de *mécompilation*.

Pour éviter toute mécompilation, il suffit de vérifier formellement le compilateur lui-même, en démontrant que la sémantique du code compilé est fidèle à la sémantique du programme source. C'est un domaine de recherche déjà ancien, avec les premiers travaux de McCarthy et Painter en 1967 et la première vérification mécanisée par Milner et Weyrauch en 1972, mais encore très actif aujourd'hui.

Le cours a exploré cette démarche de vérification sur un compilateur simple (sans optimisations) pour le langage IMP, produisant du code pour une machine virtuelle utilisant une pile. Une première démonstration, utilisant la sémantique naturelle du langage IMP, montre sans trop de peine la préservation de la sémantique des programmes IMP qui terminent.

Pour étendre ce résultat à tous les programmes IMP, y compris ceux qui divergent, nous avons étudié la technique des diagrammes de simulation entre deux sémantiques à réductions. Cependant, la sémantique à réductions d'IMP étudiée au premier cours se prête difficilement à la construction d'un tel diagramme pour notre compilateur. Nous avons donc introduit une autre sémantique à transitions pour IMP, à base de continuations et de réductions sous contexte, en partie inspirée par le *focusing* en théorie de la démonstration. Cette sémantique permet la construction d'un diagramme de simulation qui montre la préservation sémantique de notre compilateur.

COURS 3 – COMPILER MIEUX : OPTIMISATIONS, ANALYSES STATIQUES, ET LEUR VÉRIFICATION

Dans le troisième cours, nous avons étudié les optimisations dans les compilateurs. Ce sont des transformations de programmes que le compilateur applique automatiquement afin d'améliorer les performances du code engendré. Certaines optimisations s'appliquent sans conditions, mais beaucoup d'autres nécessitent un travail préalable d'analyse statique du programme à compiler, afin d'établir des propriétés vraies de toutes les exécutions du programme et sans lesquelles l'optimisation ne s'appliquerait pas. L'analyse par flux de données (*dataflow analysis*) est un formalisme classique permettant d'exprimer plusieurs analyses comme des résolutions d'équations de flux guidées par le graphe de contrôle du programme.

Nous avons illustré ces concepts sur une analyse importante, l'analyse de vivacité des variables, et son utilisation pour deux optimisations, l'élimination de code mort et l'allocation de registres. Nous avons ensuite mécanisé en Coq cette analyse, ces deux optimisations et leurs démonstrations de correction sémantique. La caractérisation sémantique de l'analyse de vivacité est intéressante car il s'agit d'une hyper-propriété qui relie deux exécutions d'un même programme dans des états différents.

Finalement, nous avons approfondi le calcul de points fixes de fonctions croissantes, une opération centrale dans de nombreuses analyses statique. Le théorème de Knaster-Tarski montre l'existence de tels points fixes, et une présentation constructive de ce théorème débouche naturellement sur un algorithme de calcul de points fixe.

Cours 4 – Des logiques pour raisonner sur les programmes

Comment montrer qu'un programme est correct (toutes ses exécutions calculent le résultat attendu) ou, *a minima*, qu'il est sûr (aucune exécution ne produit un « plantage » ou une faille de sécurité) ? On peut raisonner directement sur la sémantique du programme, mais c'est laborieux. Les logiques de programmes fournissent des principes de plus haut niveau pour raisonner sur les propriétés d'un programme.

Cette approche déductive apparaît dès 1949 dans une brève communication de Turing lui-même, puis est redécouverte et généralisée par Floyd en 1967, sur des programmes de type « organigramme ». Mais c'est Hoare, en 1969, qui introduit le concept de « logique de programmes » et donne les règles de base d'une telle logique pour les langages à contrôle structuré.

Dans ce quatrième cours, nous avons défini puis mécanisé en Coq une logique de Hoare pour le langage IMP. La mécanisation fait clairement apparaître les deux manières de définir les « triplets de Hoare » $\{P\}\ c\ \{Q\}$ qui sont valides : axiomatiquement, par des règles de déduction, ou sémantiquement, en termes des exécutions du programme c. L'équivalence entre des deux définitions démontre la correction et la complétude de la logique de Hoare.

La logique de Hoare s'étend assez facilement aux programmes manipulant des tableaux, mais plus difficilement aux programmes avec pointeurs manipulant des structures de données chaînées. Une réflexion profonde sur les notions de localité, d'alias, et de ressources conduisent O'Hearn, Reynolds et Yang à inventer, de 1999 à 2001, la logique de séparation, avec ses notions d'empreinte mémoire pour chaque assertion logique et de conjonction séparante entre ces assertions. La logique de séparation est très efficace pour vérifier les structures de données chaînées sans partage. Son extension au parallélisme à mémoire partagée, la logique de séparation concurrente, est l'objet de nombreux travaux de recherche récents.

Cours 5 – Un art abstrait : l'analyse statique par interprétation abstraite

L'interprétation abstraite est un formalisme très général, introduit par Patrick Cousot et Radhia Cousot en 1977 pour décrire et implémenter des analyses statiques plus précises que celles réalisables par analyse de flot de données. Au-delà des optimisations dans les compilateurs, l'interprétation abstraite a permis la réalisation d'outils de vérification montrant automatiquement l'absence d'erreurs à l'exécution dans de grands programmes.

Une interprétation abstraite est traditionnellement formalisée par deux fonctions, une fonction d'abstraction α et une fonction de concrétisation γ, qui forment une correspondance de Galois entre un domaine abstrait correspondant aux résultats de l'analyse statique et les ensembles d'états concrets correspondant aux exécutions du programme. De manière élégante, ces deux fonctions permettent de « calculer » les opérateurs abstraits qui implémentent l'analyse statique, c'est à dire de dériver systématiquement des définitions de ces opérateurs qui sont correctes et relativement complètes par construction.

Cependant, cette présentation classique pose problème dans le cadre d'une mécanisation en Coq ou autre logique fondée sur une théorie des types : les fonctions d'abstraction α sont généralement non calculables, et parfois mal définies.

Le cinquième cours a développé une autre présentation des domaines abstraits qui n'utilise pas de fonctions d'abstraction α, mais uniquement des prédicats de concrétisation γ reliant objets abstraits et états concrets de manière entièrement définissable en théorie des types. Les opérateurs abstraits ne sont plus « calculés » mais donnés *a priori* puis vérifiés *a posteriori*.

Nous avons suivi cette approche pour construire, en Coq, un analyseur statique générique (un « interprète abstrait ») pour IMP, et un domaine abstrait très simple pour l'analyse de constantes. Nous avons ensuite amélioré la précision de notre analyseur grâce à deux techniques typiques de l'interprétation abstraite : l'analyse « en arrière » des expressions booléennes dans les conditionnelles et les boucles, et le calcul de points fixes par itération avec élargissement. Cela nous a permis d'obtenir une analyse assez précise des intervalles de variation des variables d'un programme IMP.

Cours 6 – L'éternité, c'est long : divergence, théorie des domaines, approches coinductives

Le sixième cours est revenu sur la question de donner une sémantique aux programmes qui ne terminent pas. Cette question a une importance historique (elle a donné un coup d'arrêt à la sémantique dénotationnelle naïve de Strachey), mais aussi pratique (les programmes réactifs comme les serveurs réseau ne doivent pas terminer !).

Une première approche, de nature topologique, voit la sémantique d'un programme comme la limite de ses comportements finis. Nous l'avons illustrée sur le langage IMP, où il suffit d'écrire un interpréteur borné par une profondeur maximale de calcul, puis de faire tendre cette profondeur maximale vers l'infini, pour obtenir une sémantique dénotationnelle d'IMP qui caractérise à la fois terminaison et divergence. Pour des langages manipulant des types de données plus complexes, comme le lambda-calcul, la sémantique dénotationnelle nécessite une structure topologique plus riche, comme les domaines de Scott.

D'autres approches, qui se prêtent mieux à la mécanisation en théorie des types, reposent sur des définitions et des raisonnements coinductifs. Le cours a montré comment définir et mécaniser une sémantique naturelle pour les programmes IMP qui divergent sous forme d'un prédicat coinductif qui structure les suites infinies de réductions. Ensuite, nous avons introduit la monade de partialité, inventée par Capretta en 2005, qui fournit une représentation coinductive et constructive des calculs pouvant terminer ou diverger. Nous avons étudié comment écrire un interpréteur de référence pour IMP dans la monade de partialité. À équivalence observationnelle près, cet interpréteur satisfait les équations de la sémantique opérationnelle d'IMP, fournissant ainsi une alternative constructive aux approches topologiques.

Cours 7 – Des fonctions et des types : la sémantique d'un langage fonctionnel

Après six cours consacrés aux langages impératifs, où l'opération de base est la modification d'un état, le septième cours a changé de paradigme et braqué le projecteur sur les langages fonctionnels, où les opérations de base sont l'abstraction et l'application de fonctions, comme dans le lambda-calcul. Pour faciliter la programmation, les langages fonctionnels ajoutent au lambda-calcul une stratégie de

réduction ainsi que des types de données primitifs. On leur donne des sémantiques opérationnelles soit sous forme de réductions sous contextes, soit dans le style naturel. Nous avons mécanisé en Coq la syntaxe et la sémantique d'un petit langage fonctionnel, appelé « FUN », avec pour seule difficulté l'absence d'alpha-conversion, nous restreignant à l'évaluation faible de programmes clos.

Lorsqu'un langage offre plusieurs types de données, comme les fonctions et les booléens en FUN, des programmes absurdes apparaissent, comme `true false` (appliquer le booléen `true` comme si c'était une fonction). Le typage est un moyen de rejeter de tels programmes absurdes, dynamiquement (pendant l'exécution) ou statiquement (par analyse statique). Nous avons ajouté au langage FUN un système de types simples statiques, similaire au lambda-calcul simplement typé de Church. Pour démontrer la *sûreté* d'un système de types, c'est à dire le fait qu'aucun programme bien typé n'est absurde, plusieurs techniques existent suivant le style de sémantique utilisé : dénotationnelle, naturelle, ou à réductions. Dans ce dernier cas, la démonstration s'appuie sur deux propriétés reliant typage et réductions : la *préservation* (du typage par une réduction) et la *progression* (un programme bien typé est une valeur ou se réduit).

Le cours et la mécanisation ont suivi jusqu'ici une approche « extrinsèque » du typage, où le système de types s'applique comme un filtre sur un langage non typé. Une approche « intrinsèque » est également possible, où la syntaxe abstraite et la sémantique du langage ne sont définies que pour les termes bien typés. Il faut pour cela définir la syntaxe abstraite à l'aide de types dépendants, ou pour le moins de types algébriques généralisés (GADT). Le bénéfice de cette approche « intrinsèque » est qu'il est possible de définir une sémantique dénotationnelle du langage qui garantit par construction non seulement la sûreté du typage mais aussi la normalisation (tous les programmes bien typés terminent). Étendre cette approche à des langages et systèmes de types riches est un sujet de recherche encore actif.

Cours 8 – Coq en Coq ? Mécaniser la logique d'un assistant à la démonstration

Le dernier cours de l'année a pris la forme d'une introspection. Tout au long du cours, nous avons utilisé un assistant à la démonstration (Coq) comme langage et comme outil de vérification. Comment formaliser et mécaniser la correction sémantique d'un tel outil ? Se pose immédiatement la question de la *cohérence* de la logique implémentée par l'outil. Une logique est cohérente si elle ne peut pas dériver de paradoxes ou autres énoncés absurdes.

Pour une logique fondée sur la théorie des types, comme celle de Coq, la cohérence correspond, au sens de Curry-Howard, à l'existence d'un type qui n'est pas habité (aucun terme clos n'appartient à ce type). Nous avons donné les grandes lignes d'une démonstration de cohérence dans le style de Curry-Howard, qui étend la démonstration de sûreté vue au cours précédent avec un nouvel impératif : la propriété de normalisation, « tout terme bien typé termine ». La normalisation se démontre à l'aide de relations logiques. Cependant, les systèmes de types *imprédicatifs* comme le système F débouchent facilement sur des relations logiques mal définies car circulaires. La technique des « candidats de réductibilité » de Girard évite cette circularité.

Du système F à Coq et Agda, le chemin est long et passe par l'ajout de types dépendants, d'opérateurs de types, de types inductifs ou coinductifs, et d'une hiérarchie d'univers. Des fragments significatifs de la logique de Coq ont été formalisés et vérifiés en Coq, parfois en démontrant la normalisation, souvent en l'admettant. De même, des fragments d'Agda ont été formalisés en Agda en suivant une approche de syntaxe intrinsèquement typée.

Arriverons-nous un jour à « vérifier Coq en Coq », c'est à dire à montrer la cohérence logique de Coq par une démonstration mécanisée en Coq ? Le second théorème d'incomplétude de Gödel nous dit que c'est impossible à strictement parler. Mais il semble plausible que l'on puisse vérifier un grand fragment de Coq ou d'Agda en utilisant un fragment à peine plus grand, ce qui constituerait un magnifique « auto-amorçage » de ces assistants à la démonstration.

Séminaire 1 – *Lambda, the ultimate teaching assistant (Agda version)*

Philip Wadler (université d'Édimbourg), le 12 décembre 2019

Dans le premier séminaire, le conférencier a partagé son expérience d'utilisation d'assistants à la démonstration pour enseigner les fondations des langages de programmation. Initialement, il a utilisé *Software Foundations*, le cours interactif en Coq de Benjamin Pierce et coauteurs. Insatisfait par la présentation des démonstrations, il a développé son propre cours, *Programming Language Foundations in Agda*, qui formalise le lambda-calcul simplement typé avec une syntaxe intrinsèquement typée et des démonstrations explicites sous forme de fonctions exécutables. Ainsi, la démonstration de la propriété de progression donne directement un évaluateur pour le langage. Une extension de l'approche au système F est en cours de développement dans le cadre du langage Plutus de *smart contracts*.

Séminaire 2 – L'arithmétique des ordinateurs et sa formalisation

Sylvie Boldo (Inria, Saclay), le 19 décembre 2019

Le deuxième séminaire était consacré à la sémantique formelle des calculs numériques. La conférencière a rappelé les principes de la représentation approchée des nombres réels par des nombres en virgule flottante à précision limitée, ainsi que les spectaculaires « bugs » qui se produisent lorsque l'arithmétique à virgule flottante est mal utilisée ou mal implémentée. Elle a ensuite décrit le projet Flocq de mécanisation en Coq de l'arithmétique à virgule flottante, et ses utilisations pour vérifier formellement des routines de calcul numérique aussi bien que des algorithmes de génération de code et d'optimisation dans les compilateurs.

Séminaire 3 – Sémantique formelle de JavaScript : les enjeux du passage à l'échelle

Alan Schmitt (Inria, Rennes), le 9 janvier 2020

La mécanisation des sémantiques, comme étudiée dans le cours sur les mini-langages IMP et FUN, passe-t-elle à l'échelle de langages « du monde réel » ? Le troisième séminaire a abordé cette question dans le cas du langage Javascript. Après avoir rappelé l'importance pratique et la grande complexité de ce langage, le conférencier a décrit le projet JSCert de mécanisation en Coq du standard ECMAscript, incluant JSRef et JSExplain, des formes exécutables de la sémantique

qui aident à la valider et la comprendre. Avec environ 900 règles inductives, JSCert arrive aux limites de capacité de Coq. Le conférencier et son équipe développent une forme particulière de sémantique opérationnelle, les sémantiques « squelettiques », pour mieux gérer cette complexité et faciliter l'écriture d'interprètes abstraits.

Séminaire 4 – Logique de séparation en Coq : théorie et pratique

Arthur Charguéraud (Inria, Strasbourg), le 16 janvier 2020

Le quatrième séminaire a approfondi la mécanisation d'une logique de séparation, sujet que le quatrième cours avait juste effleuré. Le conférencier a montré comment construire une telle logique en Coq à partir d'une sémantique naturelle pour un petit langage de type « mini-ML », c'est à dire un langage fonctionnel étendu avec des références mutables. Il a ensuite décrit plusieurs outils, toujours intégrés dans Coq, qui facilitent les vérifications de programmes : un calcul de plus faible précondition, des prédicats d'encodage de structures de données, et des tactiques Coq. Tous ces éléments se retrouvent dans son outil CFML de vérification de programmes impératifs.

Séminaire 5 – Interpréteurs abstraits mécanisés

David Pichardie (ENS Rennes), le 30 janvier 2020

Le cinquième séminaire a donné la parole à un des pionniers de l'interprétation abstraite mécanisée pour approfondir la vision simplifiée donnée dans le cinquième cours. Le conférencier a montré comment construire un interpréteur abstrait « correct par construction » à partir d'une sémantique collectrice pour le langage analysé et d'une bibliothèque compositionnelle de treillis complets. Il a ensuite décrit l'analyseur statique Verasco, qui est un interpréteur abstrait écrit et vérifié en Coq pour le langage intermédiaire C# du compilateur C vérifié CompCert, mettant en œuvre des domaines abstraits complexes pour les états mémoire et pour les nombres.

Séminaire 6 – *Understanding and evolving the Rust programming language*

Derek Dreyer (MPI SWS, Sarrebruck), le 6 février 2020

Toujours dans la problématique du passage à l'échelle des sémantiques mécanisées, le sixième séminaire a décrit des travaux récents sur la formalisation du langage Rust. Après avoir rappelé les concepts de base du système de types de Rust, notamment la possession et l'emprunt temporaire de structures de données mutables, l'orateur a décrit leur formalisation dans le système λ-rust. Les types y sont interprétés sémantiquement comme des formules de logique de séparation, ce qui permet de montrer la sûreté du typage même en présence de code « *unsafe* » non typable, pourvu que ce dernier soit vérifiable en logique de séparation. Ce travail débouche sur une suggestion d'amélioration de la sémantique de Rust : un modèle de partage de pointeurs appelé *stacked borrows* et qui permet de nouvelles optimisations lors de la compilation.

Séminaire 7 – *What's in a name?* Représenter les variables et leurs liaisons

Xavier Leroy (Collège de France), le 13 février 2020

Le dernier séminaire a approfondi un point délicat de la mécanisation de langages de programmation : la représentation des variables liées et l'équivalence des termes

à renommage près des variables liées (règle d'alpha-conversion). Nous avons passé en revue quatre approches : les indices de de Bruijn, où les variables sont notées par positions au lieu de noms ; la syntaxe abstraite d'ordre supérieur (HOAS, *Higher-Order Abstract Syntax*), où les fonctions du métalangage représentent les lieurs du langage objet ; les logiques nominales, où les notions de noms de variables, d'invariance par renommage et de quantification sur un nom nouveau sont primitives dans la logique ; et l'approche « sans noms locaux » (*locally nameless*), qui combine indices de de Bruijn et approches nominales. Comme l'a montré l'étude *POPLmark challenge*, chacune de ces approches a ses forces et faiblesses, et aucune ne s'est imposée comme solution universelle.

Recherche

Les activités de recherche de la chaire Sciences du logiciel s'effectuent dans le cadre de l'équipe-projet Inria Cambium, commune au Collège de France et à l'Inria Paris et dirigée par François Pottier, directeur de recherche Inria. Cette équipe, créée en août 2019, fait suite à l'équipe-projet Inria Gallium que je dirigeais.

La recherche de l'équipe Cambium vise à améliorer la fiabilité et la sécurité du logiciel en faisant progresser les langages de programmation et les méthodes formelles de vérification de logiciel. Les principaux résultats de l'équipe pendant l'année universitaire 2019-2020 sont listés ci-dessous. Une description plus détaillée est disponible dans le rapport annuel d'activité Inria de l'équipe, https://raweb.inria.fr/rapportsactivite/RA2019/cambium/.

Vérification déductive de programmes

Les travaux sur la vérification formelle de résultats de complexité utilisant une logique de séparation étendue avec des crédits de temps ont débouché cette année sur la vérification du meilleur algorithme connu pour la détection incrémentale de cycles dans un graphe. Armaël Guéneau a soutenu sa thèse sur ces sujets en décembre 2019. Nous avons vérifié formellement un autre algorithme ambitieux : un « solveur générique local » pour le calcul incrémental de plus petits points fixes. Enfin, nous développons une logique de programmes parallèles adaptée au langage Multicore OCaml et à son intéressant modèle mémoire faiblement cohérent.

Programmation fonctionnelle typée en OCaml

Nous avons intensifié le développement du langage OCaml et de son implémentation, avec l'ajout d'un mécanisme efficace de profilage des allocations reposant sur un échantillonnage statistique, l'ajout d'un allocateur mémoire de type *best-fit*, et le portage du compilateur pour l'architecture de processeurs RISC-V. Nous avons entamé une révision en profondeur de l'environnement d'exécution d'OCaml afin de préparer l'intégration de l'extension Multicore OCaml qui y ajoute le parallélisme à mémoire partagée.

Vérification de compilateurs

Nous avons poursuivi le développement de CompCert, le compilateur C formellement vérifié : portage pour l'architecture AArch64 (ARMv8 en mode

64 bits) ; ajout de mécanismes pour définir la sémantique de fonctions prédéfinies (*built-in functions*) et permettre leur optimisation ; et amélioration des performances de l'analyseur syntaxique.

Modélisation et test de modèles mémoire faiblement cohérents

Nous continuons la recherche de formalismes axiomatiques et sémantiques pour décrire les modèles mémoires fournis par les processeurs multi-cœurs contemporains, notamment dans le cadre de la thèse de Quentin Ladevèze qui a débutée cette année. Du côté expérimental, nous avons étendu la suite d'outils DIY de test des modèles mémoires pour traiter le code automodifiant ARM ainsi que l'architecture x86 64 bits.

Publications

Charguéraud A. et Pottier F., « Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits », *Journal of Automated Reasoning*, vol. 62, nº 3, 2019, p. 331-365, https://doi.org/10.1007/s10817-017-9431-7.

de Vilhena P.E. et Paulson L.C., « Algebraically closed fields in Isabelle/HOL », *in* : *IJCAR 2020: Automated Reasoning, 10th International Joint Conference* (*Lecture Notes in Computer Science*, vol. 12167), 2020, p. 204-220, https://doi.org/10.1007/978-3-030-51054-1_12.

de Vilhena P.E., Pottier F. et Jourdan J.-H., « Spy game: Verifying a local generic solver in Iris », *Proceedings of the ACM on Programming Languages*, vol. 4, nº POPL, 2020, art. 33, p. 1-28, https://doi.org/10.1145/3371101.

Guéneau A., *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*, thèse de doctorat, Université de Paris, décembre 2019.

Mével G., Jourdan J.-H. et Pottier F., « Cosmo: A concurrent separation logic for Multicore OCaml », *Proceedings of the ACM on Programming Languages*, vol. 4, nº ICFP, 2020, art. 96, p. 1-29, https://doi.org/10.1145/3408978.

Radanne G., Saffrich H. et Thiemann P., « Kindly bent to free us », *Proceedings of the ACM on Programming Languages*, vol. 4, nº ICFP, 2020, art. 103, p. 1-29, https://doi.org/10.1145/3408985.

Simner B., Flur S., Pulte C., Armstrong A., Pichon-Pharabod J., Maranget L. et Sewell P., « ARMv8-A system semantics: Instruction fetch in relaxed architectures », *in* : *ESOP 2020: 29th European Symposium on Programming* (*Lecture Notes in Computer Science*, vol. 12075), 2020, p. 626-655, https://doi.org/10.1007/978-3-030-44914-8_23.