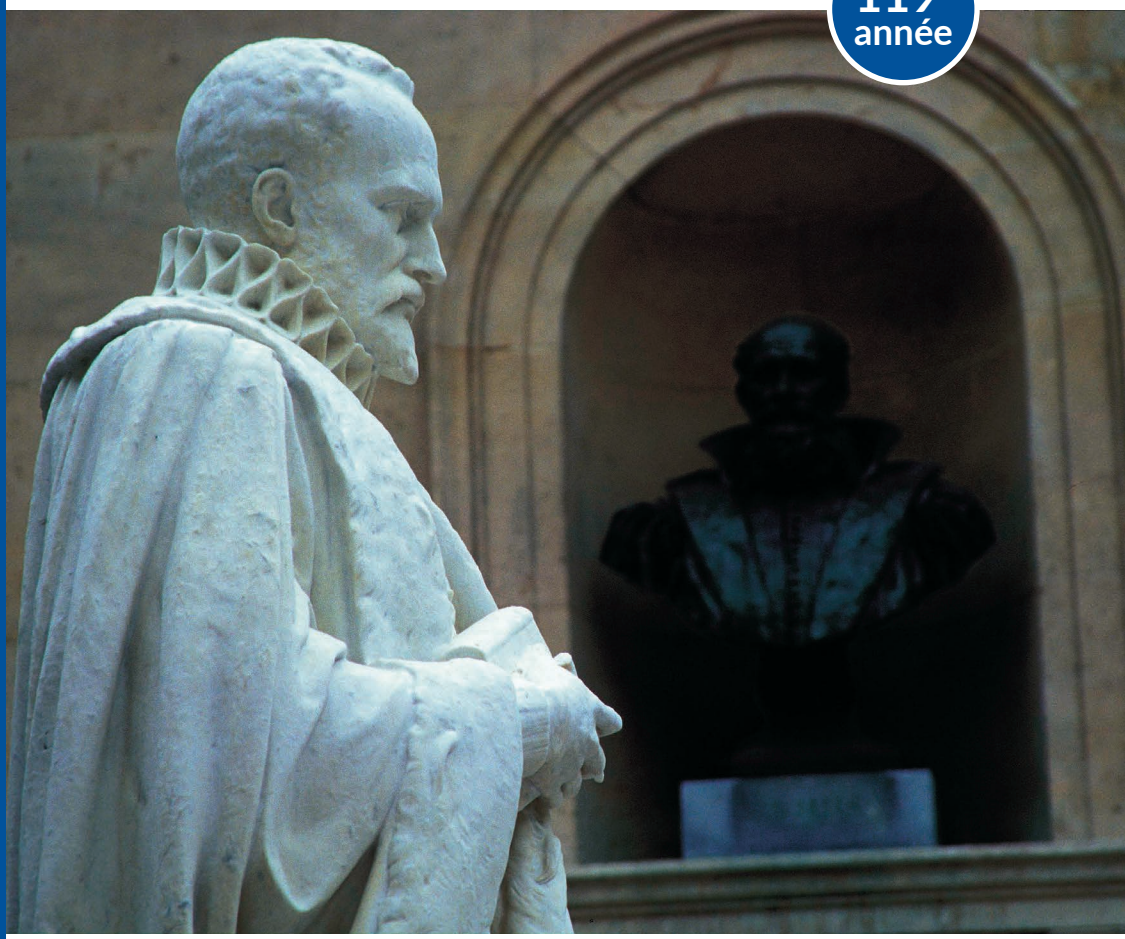
ANNUAIRE du **COLLÈGE DE FRANCE** 2018 - 2019

Résumé des cours et travaux

119e année



COLLÈGE DE FRANCE
1530

Sciences du logiciel

Xavier Leroy
Professeur au Collège de France

Mots-clés : informatique, logiciel, langages de programmation, sémantique, logique, méthodes formelles

La série de cours et de séminaires « Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui » est disponible en audio et vidéo, sur le site internet du Collège de France (https://www.college-de-france.fr/site/xavier-leroy/course-2018-2019.htm), ainsi que la leçon inaugurale « Le logiciel, entre l'esprit et la matière » (https://www.college-de-france.fr/site/xavier-leroy/inaugural-lecture-2018-2019.htm). Celle-ci a également fait l'objet d'une publication : X. Leroy, *Le Logiciel, entre l'esprit et la matière*, Paris, Collège de France/Fayard, coll. « Leçons inaugurales du Collège de France », vol. 284, 2019 ; édition numérique : Collège de France, 2019, https://doi.org/10.4000/books.cdf.7671, http://books.openedition.org/cdf/7671.

Enseignement

Leçon inaugurale – Le logiciel, entre l'esprit et la matière

La leçon inaugurale a rappelé à quel point l'informatique trouve ses racines dans la logique mathématique. Comme Leibniz avec son *calculus ratiocinator*, de nombreux logiciens ont cherché dans le calcul une source de vérités absolues. C'est en travaillant sur le programme de Hilbert, une tentative ambitieuse de refondation des mathématiques, que Church en 1936 et Turing en 1937 ont créé la théorie de la calculabilité. La machine de Turing et le lambda-calcul de Church, apparus à cette occasion, sont les grands ancêtres des microprocesseurs et des langages de programmation contemporains.

L'informatique n'est pas uniquement l'étude de ce qui est calculable : c'est l'étude de ce qui est calculable de manière efficace et effective. D'un côté, l'algorithmique

X. Leroy, « Sciences du logiciel », *Annuaire du Collège de France 2018-2019. Résumé des cours et travaux*, 119e année, Paris, Collège de France, 2022, p. 21-33, https://doi.org/10.4000/annuaire-cdf.16737.

conçoit les algorithmes efficaces et étudie leurs performances ; de l'autre, les sciences du logiciel construisent le chemin qui va des algorithmes abstraits à leur exécution effective et correcte par la machine. Cela passe notamment par des langages et outils de programmation qui, des langages d'assemblage de 1950 aux langages fonctionnels contemporains, ont fait d'immenses progrès.

Alors que nous confions de plus en plus de responsabilités aux ordinateurs, le problème principal n'est plus de développer le logiciel, mais de le vérifier. Comment s'assurer de sa justesse et de son innocuité ? Les méthodes empiriques à base de tests atteignent leurs limites et se complètent de méthodes formelles, où le programme est vu comme une définition mathématique, et sa vérification comme la démonstration, automatisée ou assistée par l'ordinateur, des propriétés attendues du programme. Introduites à la fin des années 1960, les méthodes formelles atteignent aujourd'hui une maturité telle que la vérification de langages et d'outils de programmation comme les compilateurs est possible.

Finalement, le logiciel ne serait-il que de la logique qui s'exécute sur l'ordinateur de manière algorithmiquement efficace ? Souhaitons-le, tant cette rigueur formelle est nécessaire pour faire confiance au logiciel aujourd'hui omniprésent.

Cours – Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui

La leçon inaugurale a rappelé les liens historiques entre logique et informatique. Le cours 2018-2019 de la chaire Sciences du logiciel étudié un autre lien, de nature mathématique celui-là (il s'agit d'un isomorphisme), entre langages de programmation et logiques mathématiques. Dans cette approche, démontrer un théorème, c'est la même chose que d'écrire un programme ; énoncer le théorème, c'est la même chose que de spécifier le type du programme.

Cette correspondance entre démonstration et programmation a d'abord été observée dans un cas simple par deux logiciens : Curry en 1958, puis Howard en 1969. Le résultat semblait tellement anecdotique que Howard ne l'a jamais soumis à une revue, se contentant de faire circuler des photocopies de ses notes manuscrites. Rarement photocopie a eu un tel impact scientifique, tant cette correspondance de Curry-Howard est entrée en résonance avec le renouveau de la logique et l'explosion de l'informatique théorique des années 1970 pour s'imposer dès 1980 comme un lien structurel profond entre langages et logiques, entre programmation et démonstration.

Aujourd'hui, il est naturel de se demander quelle est la « signification logique » de tel ou tel trait de langage de programmation, ou encore quel est le « contenu calculatoire » de tel ou tel théorème mathématique (c'est-à-dire, quels algorithmes se cachent dans sa démonstration ?). Plus important encore, la correspondance de Curry-Howard a débouché sur des outils informatiques comme Coq et Agda, qui sont à la fois des langages de programmation et des logiques mathématiques, et s'utilisent aussi bien pour écrire et vérifier des programmes que pour énoncer et aider à démontrer des théories mathématiques.

Le cours a retracé ce bouillonnement d'idées à la frontière entre logique et informatique, et mis l'accent sur les résultats récents et les problèmes ouverts dans ce domaine. Le séminaire a donné la parole à sept experts du domaine pour des approfondissements et des points de vue complémentaires.

Cours 1 – Les chemins d'une découverte : la correspondance de Curry-Howard, 1930-1970

Le premier cours a retracé les trois chemins qui mènent à la découverte de Curry et Howard : le lambda-calcul, la logique intuitionniste, et les systèmes de types simples.

Vers 1930, Church introduit la lambda-notation $\lambda x.e$ pour désigner la fonction qui à x associe e, et l'utilise pour unifier diverses notations mathématiques faisant intervenir des variables liées (sommes, limites, quantificateurs, etc.). Son premier lambda-calcul, publié en 1932 et 1933, est une logique mathématique qui utilise la lambda-notation pour représenter les ensembles. Cette logique est rapidement abandonnée car incohérente. En revanche, débarrassé de ses connecteurs logiques, le lambda-calcul pur s'impose dès 1936 comme un formalisme fondamental de la calculabilité, en équivalence avec les machines de Turing, puis, dès le langage Lisp (1960), comme noyau des langages fonctionnels de programmation.

La logique intuitionniste, introduite par Glivenko et Heyting vers 1928, formalise la notion de démonstration constructive, qui est un aspect important de l'intuitionnisme, la philosophie des mathématiques fondée par Brouwer. Suivant l'interprétation BHK (Brouwer, Heyting, Kolmogorov) de l'intuitionnisme, les seules propositions vraies en logique intuitionniste sont celles qui sont justifiées par une *construction*, c'est à dire un objet mathématique qui atteste de leur véracité. Ainsi, une construction de $\exists x.P$ est une paire d'une valeur pour x et d'une construction de $P(x)$. Les démonstrations non constructives de $\exists x.P$ sont donc exclues.

Les types sont présents dans de nombreux langages de programmation pour détecter des erreurs de programmation et clarifier le sens des programmes. En logique, ils apparaissent dès 1908 dans les travaux de Russell pour rejeter des termes absurdes comme $x \notin x$ (« x n'appartient pas à lui-même ») et les paradoxes qui en découlent. Partant de la théorie des types de Russell, plusieurs simplifications débouchent sur le lambda-calcul simplement typé publié par Church en 1940.

Tout est en place pour qu'entrent en résonance lambda-calcul typé et logique intuitionniste. Une première observation, due à Curry en 1958, est que si on lit les types de fonctions $\tau \to \sigma$ comme l'implication $P \Rightarrow Q$ des propositions, la règle de typage de l'application est la règle de déduction *modus ponens*, et les types des combinateurs S, K, I du lambda-calcul sont les axiomes définissant l'implication dans une logique à la Hilbert. Mais il faut attendre 1969 pour que Howard construise la correspondance complète entre lambda-calcul typé et logique intuitionniste. Non seulement les types correspondent aux propositions et les termes aux constructions (au sens BHK), mais encore l'évaluation des termes par beta-réductions correspond à l'élimination des coupures dans les démonstrations, du moins pour le fragment implicatif.

Cours 2 – Polymorphisme à tous les étages ! Du système F au calcul des constructions

La correspondance de Curry-Howard n'est-elle qu'une agréable coïncidence entre une logique intuitionniste et un lambda-calcul typé tous deux peu expressifs ? De nombreuses extensions qui apparaissent dans les années 1970 et 1980 montrent que ce n'est pas le cas.

Le *typage polymorphe* est la première de ces extensions. Il apparaît indépendamment dans les travaux de Girard (1971) et de Reynolds (1974). Reynolds est un informaticien et cherche à assouplir le typage du lambda-calcul de manière à pouvoir typer des fonctions *génériques* comme le tri, qui appliquent un même algorithme à des données de différents types. Girard est un logicien qui cherche une interprétation fonctionnelle (comme l'interprétation BHK) pour l'arithmétique du second ordre, une logique suffisamment puissante pour exprimer l'analyse. Ils convergent sur le même système de types, appelé lambda-calcul polymorphe par Reynolds et système F par Girard, qui se révèle extrêmement expressif.

D'autres formes de polymorphisme apparaissent ensuite : les constructeurs de types (types paramétrés par des types) et les types dépendants (types paramétrés par des termes). Ces formes de paramétrisation s'ajoutent à celles du lambda-calcul (termes paramétrés par des termes) et de système F (termes paramétrés par des types), débouchant sur des formalismes inutilement complexes.

Une « grande unification » apparaît dès le début des années 1970 avec la théorie intuitionniste des types de Martin-Löf, suivie en 1986 par le calcul des constructions de Coquand et Huet, puis en 1988 par les *Pure Type Systems* de Berardi. Les types et les termes partagent la même syntaxe ; un seul lambda et une seule application traitent toutes les formes de paramétrisation. Cependant, identifier entièrement types et termes *via* un type de tous les types, comme dans une première version de la théorie des types de Martin-Löf, mène à l'incohérence logique. Il reste nécessaire de stratifier types et termes, ce qui se fait à l'aide d'*univers*. De nombreux systèmes de types intéressants s'obtiennent en variant le nombre et les relations entre univers.

En l'espace de vingt ans, la correspondance de Curry-Howard s'est non seulement étendue à des logiques et des langages typés très expressifs, mais encore a inspiré des systèmes formels utilisables à la fois comme logiques et comme langages de programmation, tels la théorie des types de Martin-Löf et le calcul des constructions.

Cours 3 – Des armes de construction massive : types algébriques, prédicats inductifs, GADT

Le cours a débuté par un historique des types de données dans les langages de programmation : tableaux en Fortran, enregistrements (*records*) en Cobol, unions disjointes en Algol 68, références et pointeurs en Algol W, ainsi que les types universels des S-expressions en Lisp et des termes en Prolog. Les *types algébriques*, introduits dans le langage HOPE et popularisés par ML et Haskell, unifient toutes ces notions. Les types algébriques sont des sommes de produits, récursives et nommées. Ils peuvent exprimer de nombreuses structures de données informatiques, ainsi que la conjonction et la disjonction dans une approche de Curry-Howard.

Les lambda-calculs étudiés dans les cours précédents ne fournissent pas de types de données, car ils peuvent être définis en termes de fonctions, généralisant l'astucieux codage fonctionnel des nombres entiers introduit par Church. Ce codage est bien typé dans les systèmes F et Fω, mais avec des restrictions sur les analyses de cas que l'on peut faire sur les données. Ainsi, on ne peut pas démontrer $0 \neq 1$ avec le codage de Church typé !

Pour dépasser ces restrictions, Paulin-Mohring et Pfenning introduisent en 1989 les *types inductifs*, une variante des types algébriques compatibles avec la théorie des types. En moins, la récursion est restreinte et une condition de positivité est ajoutée afin de préserver la normalisation forte et la cohérence logique. En plus, les types

inductifs peuvent être dépendants. Par la correspondance de Curry-Howard, ils deviennent des prédicats inductifs, avec lesquels on peut définir les connecteurs logiques « et », « ou », « il existe ». Une extension naturelle, les *familles inductives*, permet d'exprimer tous les prédicats définissables par règles d'inférence, y compris l'égalité et les ordres bien fondés.

Le cheminement des idées continue : des types algébriques en programmation vers les types et familles inductifs en logique, puis retour en programmation avec les types algébriques généralisés, *Generalized Algebraic Data Types* (GADT). Imaginés par Augustsson et Petersson en 1994, puis intégrés dans Haskell en 2007 et OCaml en 2012, les GADT sont une extension des types algébriques permettant d'exprimer certaines propriétés des données dans le style des prédicats inductifs, incluant des égalités entre types et des quantificateurs existentiels, sans pour autant nécessiter toute la mécanique des types dépendants.

La recherche autour des types inductifs reste active : quelles sont les bonnes conditions de garde qui garantissent la normalisation ? Peut-on identifier des valeurs d'un type inductif, réalisant ainsi le quotient de ce type par une relation d'équivalence ? Le dixième cours est revenu sur cette dernière question.

Cours 4 – Il faut qu'une porte soit ouverte ou fermée ! Logique classique, continuations, opérateurs de contrôle

On peut opposer logique classique et logique intuitionniste comme deux ennemis irréductibles, comme dans la polémique entre Hilbert et Brouwer en 1927. Plus positivement, on peut, comme Gödel dans les années 1930, utiliser la logique intuitionniste pour mieux comprendre la logique classique. Tout d'abord, la logique classique peut être vue comme la logique intuitionniste à laquelle on ajoute un axiome classique, comme le tiers exclu « P ou $\neg P$ » qui donne son titre à ce cours. Plus surprenant : la logique classique se plonge dans la logique intuitionniste *via* une traduction par *double négation*. Si P est une formule propositionnelle, P est vraie en logique classique si et seulement si $\neg\,\neg P$, « P n'est pas faux », est démontrable en logique intuitionniste. Kolmogorov, Gödel, Gentzen, Kuroda étendent cette traduction doublement négative à des formules avec quantificateurs. Friedman la généralise pour montrer que toute formule Π^0_2 démontrable en arithmétique classique l'est aussi en arithmétique intuitionniste.

Coup de théâtre en 1990 : Murthy et Griffin mettent en correspondance ces approches de la logique classique avec le concept de *continuation* dans les langages de programmation. Dans un langage fonctionnel, la continuation d'une expression a est le calcul qui reste à faire pour terminer le programme une fois a évaluée. On peut voir cette continuation comme une fonction de la valeur de a vers la valeur du programme complet. Un programme peut agir sur ses continuations de deux manières : soit on l'écrit en style à passage de continuation (CPS, *Continuation-Passing Style*), où chaque expression reçoit sa continuation en argument supplémentaire ; soit on ajoute au langage de programmation des opérateurs de contrôle permettant de réifier les continuations, comme l'opérateur `call/cc` du langage Scheme.

Murthy est le premier à observer que les transformations CPS, qui mettent un programme en forme CPS, correspondent, au sens de Curry-Howard, avec les traductions par double négation en logique. Plus précisément, la traduction de Kolmogorov correspond à la transformation CPS en appel par nom, et la traduction

de Kuroda à la transformation CPS en appel par valeur. En parallèle, Griffin observe que certains opérateurs de contrôle admettent des types qui correspondent, au sens de Curry-Howard, avec des axiomes de logique classique. Ainsi, l'opérateur `call/cc` correspond à la loi de Clavius, $(\neg P \Rightarrow P) \Rightarrow P$, et l'opérateur C de Felleisen à l'élimination de la double négation, $\neg\,\neg P \Rightarrow P$.

Ces beaux résultats permettent déjà d'identifier le contenu calculatoire de démonstrations classiques. Cependant, un aspect important de la logique classique est perdu : la dualité entre conjonction et disjonction via la négation, visible dans la loi de De Morgan $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ et magistralement exprimée dans le calcul des séquents classiques de Gentzen (1934). Quel langage de programmation correspond par Curry-Howard à ce calcul des séquents classique ? Les travaux récents de Parigot, Curien, Herbelin, Wadler et d'autres donnent des éléments de réponse à cette question.

Cours 5 – Peut-on changer le monde ? Programmation impérative, effets monadiques, effets algébriques

Jusqu'ici nous avons mis en correspondance des logiques et des langages fonctionnels. Cependant, beaucoup de programmes sont *impératifs* et non purement fonctionnels, car leur exécution a des *effets* sur le monde extérieur : ils consomment des entrées, produisent des sorties, modifient des fichiers, etc.

Les *monades* sont un concept issu de la théorie des catégories et appliqué à la sémantique dénotationnelle par Moggi en 1989. Elles fournissent un cadre élégant pour décrire de nombreuses sortes d'effets allant de l'affectation à la programmation probabiliste. La transformation monadique, qui généralise la transformation CPS du quatrième cours, permet de traduire tout programme avec effets monadiques en programme fonctionnel pur.

Certaines monades ont une signification logique claire. Ainsi, la transformation monadique des continuations est une traduction doublement négative qui injecte la logique classique dans la logique intuitionniste, comme vu dans le quatrième cours ; de même, la transformation monadique des exceptions injecte la logique intuitionniste dans la logique minimale, sans loi *ex falso quod libet*. Sur les monades en général, le contenu logique se réduit à des correspondances avec certaines logiques modales, en particulier la modalité laxiste $\bigcirc$ de Mendler, et une combinaison des modalités classiques $\Box$ et $\Diamond$ observée par Pfenning et Davies.

En combinaison avec des types dépendants, les monades permettent également de raisonner sur les programmes. On peut enrichir la monade d'état pour capturer des invariants sur l'état ainsi que des propriétés d'évolution monotone de l'état. Plus généralement, les monades de Hoare (Nanevsky *et al.*, 2008) et les monades de Dijkstra (Swamy *et al.*, 2013) permettent d'associer aux fonctions des préconditions et des postconditions quelconques, comme dans une logique de programmes. Le langage F* met en pratique cette approche monadique de la preuve de programmes.

De même que les monades décrivent la propagation des effets, la théorie des effets algébriques de Plotkin, Power, Pretnar *et al.* cherche à décrire la génération et le traitement des effets. Elle débouche sur une nouvelle construction pour les langages de programmation, les gestionnaires d'effets (*effect handlers*), qui généralisent les gestionnaires d'exceptions.

Cours 6 – Des théorèmes gratuits : la paramétricité

On s'attend à ce qu'un type abstrait puisse être réalisé par différents types concrets sans que les utilisateurs du type abstrait observent de différences. C'est le principe d'indépendance vis-à-vis des représentations énoncé par Reynolds en 1983. Symétriquement, on s'attend à ce qu'une fonction polymorphe exécute le même algorithme lorsqu'on l'applique à des données de différents types.

La théorie de la paramétricité donne un sens mathématiquement précis à ces intuitions. Elle s'appuie sur la notion de *relations logiques* : des relations entre termes, indexées par des types, et compatibles avec la structure des types. En particulier, deux fonctions sont reliées au type $\tau \to \sigma$ si et seulement si elles envoient des arguments reliés au type τ sur des résultats reliés au type σ. Ces relations logiques ont été introduites par Plotkin pour l'étude du lambda-calcul simplement typé. Reynolds étend les relations logiques au cas des types abstraits, lui permettant de formaliser et de démontrer le principe d'indépendance vis-à-vis des représentations. L'extension au lambda-calcul polymorphe est plus délicate car il est facile de tomber dans des définitions circulaires, mais est finalement résolue vers 1990 par Bruce, Meyer et Mitchell, en s'inspirant de la technique des candidats de réductibilité de Girard.

Une application spectaculaire de la paramétricité est les *Theorems for free* (théorèmes gratuits), titre d'un célèbre article de Wadler (1991). Toutes les fonctions du système F ayant un certain type polymorphe vérifient des équations qui ne dépendent que du type ; il est inutile de redémontrer ces équations pour chaque fonction. Ainsi, toute fonction $f : \forall\alpha.\ \alpha \to \alpha$ vérifie $f\,x = x$ et est donc l'identité. Toute fonction $f : \forall\alpha.\ \alpha\ \texttt{list} \to \alpha\ \texttt{list}$ vérifie $f \circ \texttt{map}\ g = \texttt{map}\ g \circ f$. La paramétricité permet aussi de démontrer l'isomorphisme entre un type inductif et son codage fonctionnel mentionné au troisième cours.

Les travaux récents de Bernardy et de Lasson montrent que la paramétricité non seulement s'étend à la théorie des types, mais encore peut s'internaliser dans cette théorie. Un type A se traduit mécaniquement en une relation R_A qui est la relation logique au type A ; un terme clos $a : A$ se traduit en une preuve que a est relié à lui-même par R_A. L'approche s'étend naturellement aux constructeurs de types et aux types dépendants.

Cours 7 – Le *forcing*, une transformation de programme comme une autre ?

Vers 1870, Cantor démontre que le cardinal de l'ensemble $\mathbb{R}$ des réels est strictement plus grand que celui de l'ensemble $\mathbb{N}$ des entiers ; en d'autres termes, que l'infini continu ($\mathbb{R}$) est plus grand que l'infini dénombrable ($\mathbb{N}$). Il énonce l'*hypothèse du continu* (HC) : il n'existe pas de cardinal intermédiaire entre $\mathbb{N}$ et $\mathbb{R}$. Cette conjecture restera longtemps ouverte, au point que Hilbert en 1901 en fait le premier de sa liste de 20 grands problèmes mathématiques. En 1938, Gödel montre que HC est cohérente avec la théorie des ensembles : on peut la prendre comme axiome car on ne peut pas démontrer sa négation. En 1964, Cohen établit que HC n'est pas démontrable car sa négation est cohérente. L'hypothèse du continu est donc *indépendante* de la théorie des ensembles.

Pour sa démonstration, Cohen a inventé une technique nouvelle de construction de modèles, le *forcing*, qui permet d'adjoindre à un modèle existant de nouveaux ensembles et de contrôler leurs propriétés *via* des approximations finies : les

conditions de forcing. À partir de 2009, les travaux de Krivine, Miquel, Rieg, Jaber, Tabareau et Sozeau débouchent sur une toute autre présentation du *forcing*, internalisée en théorie des types sous forme de transformations de propositions et de leurs termes de preuve, dans le style des traductions doublement négatives du quatrième cours. Ces transformations aident à imaginer le contenu calculatoire de la technique de *forcing*. Ainsi, le *forcing* intuitionniste tel que présenté par Jaber *et al.* peut se voir comme une transformation monadique, au sens du cinquième cours, vers une monade d'entrées asynchrones où chaque terme peut à tout moment recevoir davantage de données provenant de l'extérieur.

En spécialisant les résultats précédents à des conditions de *forcing* qui sont juste des nombres entiers, on obtient une logique modale, appelée logique interne du topos des arbres par Birkedal *et al.*, qui permet de définir des types récursifs généraux (non algébriques), tout en se plongeant par traduction dans la théorie des types. Cela rejoint les idées de *step-indexing*, objet du cours suivant.

Cours 8 – À pas comptés : les techniques de *step-indexing*

Au début de ce cours, nous avons repris et approfondi la notion de *relation logique* introduite au sixième cours, tout d'abord en les reformulant de manière purement opérationnelle : au lieu de se reposer sur une sémantique dénotationnelle du langage, on peut tout définir en termes d'une relation de réduction entre termes.

L'étape suivante consiste à étendre les relations logiques à des traits délicats de langages comme OCaml : les types récursifs généraux et les références (structures de données mutables) pouvant contenir des fonctions. Dans les deux cas, la définition usuelle des relations logiques n'est plus bien fondée par récurrence sur les types. En 2001, Appel et McAllester contournent brillamment ce problème en utilisant un autre critère de bonne fondation : le nombre d'étapes de réduction que l'on se donne pour observer les propriétés des termes. Cette technique, connue sous le nom de *step-indexing* (comptage de pas), est étendue aux relations logiques binaires par Ahmed en 2006.

Dans cette présentation des relations logiques, les démonstrations nécessitent une comptabilité minutieuse et lourde des étapes de réduction. Appel *et al.* (2007) puis Dreyer *et al.* (2011) développent une présentation plus élégante dans une logique modale ayant la modalité $\triangleright P$ (« P est vraie plus tard ») et satisfaisant la loi de Löb $(\triangleright P \Rightarrow P) \Rightarrow P$. Cette logique n'est autre que la logique interne du topos des arbres que nous avons obtenue au septième cours par *forcing* intuitionniste sur les entiers.

Cours 9 – Sisyphe heureux : types infinis, démonstrations par coinduction, et programmation réactive

Le troisième cours a introduit les types et prédicats inductifs, un mécanisme général pour définir et raisonner sur les structures de données finies. Comment faire de même pour les structures de données potentiellement infinies, par exemple les flux (*streams*) ou les graphes vus comme des arbres infinis ? On parle alors de types et de prédicats *coinductifs*.

Dans la présentation ensembliste classique, les types coinductifs correspondent à des plus grands points fixes d'opérateurs croissants, alors que les types inductifs sont des plus petits points fixes. La théorie des types suit une autre présentation, en termes d'arbres bien fondés (toute branche est finie) pour le cas inductif et

possiblement mal fondés (avec des branches infinies) pour le cas coinductif. Un critère de garde garantit la bonne formation des définitions : récursions structurelles pour le cas inductif, corécursions productives pour le cas coinductif.

Une présentation plus moderne et d'inspiration plus algébrique caractérise les types inductifs comme engendrés par leurs constructeurs (`nil` et `cons` pour les listes), et les type coinductifs comme engendrés par leurs observations (projections ; `head` et `tail` pour les flux). Cette présentation débouche sur un élégant style de programmation à base de cofiltrage et de *copatterns* (Abel *et al.*, 2013).

Le cours a présenté deux des nombreuses applications de la coinduction : la monade de partialité (Capretta, 2005) qui permet de définir des récursions générales en théorie des types et de raisonner coinductivement sur leur terminaison ou leur divergence, et la modélisation de la programmation réactive à l'aide de fonctions causales sur les flux de données.

Cours 10 – Qu'est-ce que l'égalité ? De Leibniz à la théorie homotopique des types

Que veut dire le signe « égal » dans un programme informatique ? dans un texte mathématique ? La première définition précise de l'égalité, attribuée à Leibniz, dit que x et y sont égaux si et seulement s'ils satisfont les mêmes propriétés : $\forall P, P(x) \Leftrightarrow P(y)$. Cette définition est équivalente à d'autres définitions plus récentes, en logique du premier ordre ou en théorie des ensembles, ainsi qu'à la définition en théorie des types introduite par Martin-Löf en 1973. Celle-ci se présente comme un type $x =_A y$ des identités (preuves d'égalité) entre $x : A$ et $y : A$, un constructeur qui est l'axiome de réflexivité, et un éliminateur qui est la loi de remplacement entre égaux.

L'égalité de Martin-Löf, ou sa reformulation comme un prédicat inductif en Coq, est parfaite pour raisonner sur les types de premier ordre comme les entiers mais pose quelques difficultés sur les types plus riches. En particulier, on ne peut pas démontrer que deux fonctions égales point à point sont égales, ni que deux identités entre x et y sont égales. Cela conduit beaucoup de développements en Coq à ajouter comme axiomes l'extensionalité des fonctions et l'unicité des identités.

Un nouveau point de vue sur l'égalité en théorie des types est fourni par la théorie de l'homotopie en topologie algébrique et son extension à l'ordre supérieur : la structure de ω-groupoïde. On peut interpréter les identités comme des chemins et les identités entre identités comme des homotopies entre chemins. Sur la base de cette inspiration homotopique et catégorique, Voevodsky, Avodey, Coquand et d'autres ont, à partir de 2005, développé la théorie homotopique des types (*Homotopy Type Theory*, HoTT), une variante de la théorie des types où l'égalité joue un rôle central. Ainsi, HoTT définit les propositions logiques comme les types où tous les éléments sont égaux entre eux, et les ensembles comme les types où l'égalité est une proposition. HoTT introduit également de nouveaux outils : la troncature propositionnelle, les types quotients, les types inductifs d'ordre supérieur, qui ouvrent des possibilités nouvelles tant en logique mathématique qu'en programmation typée.

Cours 11 – Conclusion et discussions

Cinquante ans presque jour pour jour après la diffusion de la note de Howard, le dernier cours de l'année a fait un rapide bilan de la correspondance de Curry-Howard et des nombreuses directions de recherche qu'elle a inspirées, tant en informatique

qu'en logique. Il a ensuite laissé place à une discussion à partir des questions provenant des auditeurs, notamment plusieurs questions sur l'impact de cette correspondance sur la programmation fonctionnelle au quotidien : Curry-Howard fait-il de moi un meilleur programmeur ?

Séminaires : en relation avec le sujet du cours

Séminaire 1 – Les types dépendants : tout un programme !

Pierre-Évariste Dagand (CNRS, laboratoire LIP6, Paris), le 28 novembre 2018

Le premier séminaire a consisté en une défense et illustration de la programmation avec types dépendants dans le langage Agda. Le conférencier est parti d'un problème classique : la sémantique d'un langage d'expressions arithmétiques et sa compilation vers le code d'une machine à pile. Il a montré que de nombreux cas d'erreurs sont à traiter si on n'emploie que des types simples, mais disparaissent si on indexe les expressions et les instructions de la machine par les types des valeurs produites. Cela garantit par construction que le compilateur préserve les types. Les instructions peuvent aussi être indexées par leur sémantique opérationnelle, ce qui garantit par construction que le compilateur est correct (préserve la sémantique), atteignant ainsi un point extrême de la programmation à types dépendants.

Séminaire 2 – Mathématiques assistées par ordinateur

Assia Mahboubi (Inria, équipe Gallinette, Nantes), le 5 décembre 2018

Le deuxième séminaire a été consacré aux utilisations dans la recherche en mathématiques de l'ordinateur en général, et en particulier des assistants de preuve (Coq, HOL, Mizar, etc.). Depuis longtemps employé comme support d'expériences et aide à l'intuition, le calcul sur ordinateur peut aussi contribuer à part entière à la démonstration de grands théorèmes. L'oratrice a développé l'exemple de la démonstration par Helfgott de la conjecture faible de Goldbach, qui fait appel à des encadrements numériques d'intégrales ; elle et ses coauteurs ont pu démontrer formellement ces encadrements *via* une arithmétique d'intervalles vérifiée en Coq. Cependant, la formalisation « sur machine » des structures algébriques usuelles pose des problèmes subtils notamment liés à « l'ambiguïté typique » entre une structure et son ensemble support. Après les premiers succès, où l'assistance de l'ordinateur a permis d'achever des démonstrations longues (théorème des quatre couleurs, conjecture de Kepler), verrons-nous un nouveau style de mathématiques émerger de l'utilisation de ces assistants ?

Séminaire 3 – Programmer avec Coq : récursion et filtrage dépendant

Matthieu Sozeau (Inria, équipe PiR2, Paris), le 12 décembre 2018

Le troisième séminaire a continué l'exploration de la programmation avec types dépendants débutée au premier séminaire. L'orateur a identifié deux écueils majeurs pour programmer dans une théorie des types comme celle de Coq : la partialité et la non-terminaison, et a montré comment l'utilisation de types dépendants permet de rendre les fonctions totales et d'en démontrer la terminaison. Se pose cependant le problème du filtrage dépendant, c'est-à-dire l'analyse par cas sur ces types dépendants, qui est plus complexe que le filtrage non-dépendant usuel.

Séminaire 4 – Peut-on dupliquer un objet ? Linéarité et contrôle des ressources

Guillaume Munch-Maccagnoni (Inria, équipe Gallinette, Nantes), le 19 décembre 2018

Au contraire de la programmation fonctionnelle pure, la programmation système doit gérer des ressources limitées par la physique de la machine : mémoire, fichiers, etc. Il faut en particulier assurer que ces ressources ne sont ni dupliquées ni perdues. Le conférencier a développé le concept de linéarité comme discipline de programmation système, mais aussi sous l'angle de la logique linéaire, en enfin sous forme de systèmes de types comme celui du langage Rust.

Séminaire 5 – Réalisabilité et *forcing*

Alexandre Miquel (Université de la République, Montevideo), le 16 janvier 2019

Ce séminaire a approfondi la technique de *forcing* de Cohen et l'analyse de son contenu calculatoire abordées dans le septième cours. Le conférencier a ensuite expliqué la théorie de la réalisabilité, une méthode de construction de modèles de la logique formalisant et étendant l'interprétation BHK, et montré ses liens formels avec le *forcing* : la réalisabilité peut être vue comme un *forcing* non commutatif. Poursuivant cette unification, il a introduit la notion d'algèbre implicative, une structure qui représente uniformément programmes et types, réalisateurs et formules logiques.

Séminaire 6 – Sémantique des programmes fonctionnels probabilistes, à la lumière de la logique linéaire

Christine Tasson (Université Paris Diderot, laboratoire IRIF), le 23 janvier 2019

Le sixième séminaire a étudié les langages probabilistes. Il s'agit de langages fonctionnels simples comme PCF auxquels on ajoute de l'aléatoire, par exemple une primitive qui s'évalue en 0 avec probabilité ½ et en 1 avec probabilité ½. Ces langages servent notamment à l'étude de l'inférence Bayésienne. La conférencière a développé des sémantiques dénotationnelles pour le langage PCF probabiliste utilisant des espaces cohérents probabilistes. Elle a dégagé des liens avec la logique linéaire et notamment son fragment semi-polarisé, qui correspond, au sens de Curry-Howard, au calcul *call-by-push-value* de Levy.

Séminaire 7 – Du calcul des constructions à la théorie des types univalents

Thierry Coquand (Université de Göteborg), le 30 janvier 2019

Le dernier séminaire a conclu la série par une perspective personnelle sur l'histoire et l'actualité de la théorie des types. Le conférencier est revenu sur son invention avec Huet du calcul des constructions, le formalisme à la base de l'assistant de preuves Coq. Le calcul des constructions est apparu comme une synthèse entre les travaux de de Bruijn sur le système Automath pour la mécanisation des mathématiques, les travaux de Girard sur le système Fω, et les travaux de Martin-Löf sur sa théorie des types. Ces formalismes débouchent aujourd'hui sur une approche nouvelle et prometteuse de la notion mathématique d'identification entre deux collections, généralisant les notions d'isomorphismes entre structures algébriques et de transport de propriétés.

Recherche

Je suis responsable scientifique de l'équipe-projet Inria Gallium, localisée au centre Inria de Paris. Les travaux de l'équipe visent à améliorer la fiabilité et la sécurité du logiciel au moyen de nouveaux langages de programmation, plus expressifs et plus sûrs, et de la vérification formelle de programmes et d'outils de programmation. Les principaux résultats de l'équipe pendant l'année universitaire 2018-2019 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/RA2018/gallium/.

Vérification formelle d'algorithmes et de leur complexité

À l'aide d'une logique de séparation étendue avec des crédits de temps, nous avons vérifié formellement la complexité de plusieurs algorithmes, dont un algorithme particulièrement complexe de détection incrémentale de cycles. Nous avons introduit la notion duale de reçus de temps afin d'établir des bornes inférieures sur les temps d'exécution, ce qui a permis de montrer par exemple l'absence de débordements arithmétiques dans l'algorithme *union-find*.

Vérification formelle de compilateurs et outils de programmation

Nous avons poursuivi le développement de CompCert, le compilateur C formellement vérifié : ajout et vérification d'une passe d'optimisation de type *if-conversion* pour éliminer des branchements conditionnels ; introduction de nouveaux mécanismes pour spécifier les fonctions prédéfinies par le compilateur ; et poursuite de l'effort d'industrialisation en partenariat avec la société AbsInt GmbH. Plus en amont, nous explorons une nouvelle manière de formaliser la sémantique dénotationnelle d'un langage de programmation sous une forme exécutable, s'appuyant sur la monade de partialité de Capretta mentionnée dans le neuvième cours.

Programmation fonctionnelle typée en OCaml

Nous avons poursuivi le développement du langage OCaml et de son implémentation : extensions du filtrage sur les GADT, introduction de nouveaux mécanismes de substitution dans les modules et leurs signatures, améliorations des performances du code compilé et du gestionnaire de mémoire (GC), et refonte des mécanismes d'autoconfiguration et de construction du système.

Modélisation et test de modèles mémoire faiblement cohérents

Notre étude formelle et expérimentale des modèles mémoires fournis par les processeurs multi-cœurs contemporains se poursuit, avec notamment une étude des comportements en mémoire des codes auto-modifiants sur architecture ARM. Plus en amont, nous avons travaillé sur un nouveau formalisme pour décrire les modèles mémoire, inspiré par les sémantiques à jeux et les structures d'événements.

PUBLICATIONS

ACAR U., AKSENOV V., CHARGUÉRAUD A. et RAINEY M., « Provably and practically efficient granularity control », *in* : *PPoPP 2019: 24th ACM Symposium on Principles and Practice of Parallel Programming*, New York, Association for Computing Machinery, 2019, p. 213-228, https://doi.org/10.1145/3293883.3295725.

ACAR U., CHARGUÉRAUD A., GUATTO A., RAINEY M. et SIECZKOWSKI F., « Heartbeat scheduling: Provable efficiency for nested parallelism. », *in* : *PLDI 2018: 39th ACM Conference on Programming Language Design and Implementation*, New York, Association for Computing Machinery, 2018, p. 769-782.

AKSENOV V., *Synchronization Costs in Parallel Programs and Concurrent Data Structures*, thèse de doctorat, université ITMO de Saint-Pétersbourg et université Paris Diderot, sept. 2018.

ALGLAVE J., MARANGET L., MCKENNEY P., PARRI A. et STERN A., « Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel », *in* : *ASPLOS 2018: 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, Association for Computing Machinery, 2018, p. 405-418.

BOZMAN C., CANOU B., DI COSMO R., COUDERC P., GESBERT L., HENRY G., LE FESSANT F., MAUNY M., MOREL C. et PEYROT L., « Learn-OCaml : un assistant à l'enseignement d'OCaml », *in* : *JFLA* (*Journées francophones des langages applicatifs) 2019* (Les Rousses, 2019), hal-01962838.

GUÉNEAU A., CHARGUÉRAUD A. et POTTIER F., « A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification », *in* : *ESOP 2018: 27th European Symposium on Programming*, *Lecture Notes in Computer Science*, vol. 10801, 2018, p. 533-560.

GUÉNEAU A., JOURDAN J.-H., CHARGUÉRAUD A. et POTTIER F., « Formal proof and analysis of an incremental cycle detection algorithm », *in* J. HARRISON, J. O'LEARY et A. TOLMACH (dir.), *ITP 2019: 10th Conference on Interactive Theorem Proving* (Portland, 8-12 septembre 2019), *Leibniz International Proceedings in Informatics*, vol. 141, 2019, p. 18:1-18:20.

KREMER S., MÉ L., RÉMY D. et ROCA V., *Cybersecurity. Current challenges and Inria's research directions*, Rocquencourt, Inria, coll. « White Book », vol. 3, 2019.

KÄSTNER D., WÜNSCHE U., BARRHO J., SCHLICKLING M., SCHOMMER B., SCHMIDT M., FERDINAND C., LEROY X. et BLAZY S., « CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler », *in* : *ERTS 2018: Embedded Real Time Software and Systems* (Toulouse, 31 janvier-2 février 2018), hal-01643290. .

MÉVEL G., JOURDAN J.-H. et POTTIER F., « Time credits and time receipts in Iris », *in* : *ESOP 2019: 28th European Symposium on Programming*, *Lecture Notes in Computer Science*, vol. 11423, 2019, p. 1-27.

SCHOMMER B., CULLMANN C., GEBHARD G., LEROY X., SCHMIDT M. et WEGENER S., « Embedded program annotations for WCET analysis », *in* : *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*, *OpenAccess Series in Informatics (OASIcs)*, vol. 63, 2018, 8:1-8:13.