Annuaire du Collège de France

121e année

2020
2021

Résumé des cours et travaux

COLLÈGE
DE FRANCE
1530

Annuaire du Collège de France
Cours et travaux du Collège de France

121 | 2024
2020-2021

Sciences du logiciel

Xavier Leroy

Édition électronique
URL : https://journals.openedition.org/annuaire-cdf/19165
DOI : 10.4000/12kth
ISBN : 978-2-7226-0778-1
ISSN : 2109-9227

Éditeur
Collège de France

Édition imprimée
Date de publication : 18 novembre 2024
Pagination : 19-29
ISBN : 978-2-7226-0777-4
ISSN : 0069-5580

Ce document vous est fourni par Collège de France

Référence électronique
Xavier Leroy, « Sciences du logiciel », *L'annuaire du Collège de France* [En ligne], 121 | 2024, mis en ligne le 01 octobre 2024, consulté le 28 novembre 2024. URL : http://journals.openedition.org/annuaire-cdf/19165 ; DOI : https://doi.org/10.4000/12kth

SCIENCES DU LOGICIEL

Xavier Leroy
Professeur au Collège de France

La série de cours « Logiques de programmes : quand la machine raisonne sur ses logiciels » est disponible en audio et en vidéo sur le site internet du Collège de France (https://www.college-de-france.fr/agenda/cours/logiques-de-programmes-quand-la-machine-raisonne-sur-ses-logiciels), ainsi que le séminaire du même nom (https://www.college-de-france.fr/agenda/seminaire/logiques-de-programmes-quand-la-machine-raisonne-sur-ses-logiciels).

ENSEIGNEMENT

COURS – LOGIQUES DE PROGRAMMES : QUAND LA MACHINE RAISONNE SUR SES LOGICIELS

De même qu'une logique mathématique permet de démontrer des propriétés des objets mathématiques, une logique de programme permet de démontrer des propriétés d'un programme informatique et de toutes ses exécutions possibles. Apparues dans les années 1960 avec les travaux fondateurs de Floyd et de Hoare, les logiques de programmes se sont énormément développées depuis les années 2000, avec des avancées conceptuelles comme les logiques de séparation et de belles applications à la vérification déductive de logiciels critiques.

Le cours a retracé cette évolution des idées et présenté de nombreux résultats récents dans ce domaine.

X. Leroy, « Sciences du logiciel », *Annuaire du Collège de France 2020-2021. Résumé des cours et travaux*, 121e année, Paris, Collège de France, 2024, p. 19-29, https://doi.org/10.4000/annuaire-cdf.19165.

Le séminaire a approfondi l'approche dans plusieurs directions, allant de la mise en pratique dans des outils de vérification déductive de qualité industrielle à l'émergence de nouveaux problèmes et de nouvelles techniques de vérification.

Cours 1 – Comment raisonner sur un logiciel ? La naissance des logiques de programmes

Comment s'assurer qu'un logiciel fait ce qu'il est censé faire ? Les méthodes classiques de vérification et de validation du logiciel, reposant sur le test, les revues et les analyses, ne suffisent pas toujours. La vérification déductive permet d'aller plus loin en établissant des propriétés vraies de toutes les exécutions possibles d'un programme, *via* des raisonnements formels au sein d'une logique appropriée : une logique de programmes.

Le premier cours a illustré cette approche par la vérification déductive d'une fonction de recherche dichotomique dans un tableau trié, un algorithme très utilisé et souvent implémenté de manière incorrecte.

Ensuite, le cours a retracé l'émergence de la vérification déductive et des logiques de programmes *via* trois publications fondatrices. La brève communication d'Alan Turing en 1949, intitulée « Checking a large routine », introduit deux idées majeures : les assertions logiques et les ordres de terminaison, et les illustre sur la vérification d'un petit programme exprimé par un organigramme. Trop en avance sur son temps et jamais publié formellement, ce texte précurseur tombe dans l'oubli jusqu'en 1984.

L'article de Robert W. Floyd en 1967, « Assigning meanings to programs », réinvente l'approche de Turing et l'approfondit considérablement, avec une formalisation complète des conditions de vérification qui garantissent qu'un programme est correctement annoté, et l'observation, révolutionnaire pour l'époque, qu'une telle formalisation constitue une sémantique formelle du langage de programmation utilisé.

Enfin, l'article de C.A.R. Hoare en 1969, intitulé « An axiomatic basis for computer programming », constitue le manifeste de la vérification déductive moderne. L'article étend les résultats de Floyd à un langage à contrôle structuré (séquences, conditionnelles, boucles) et introduit des notations toujours utilisées aujourd'hui (les « triplets de Hoare »). Au-delà des contributions techniques, cet article est visionnaire tant par son approche purement axiomatique de la vérification déductive que par son analyse lucide de l'intérêt pratique de cette approche.

Cours 2 – Variables et boucles : la logique de Hoare

Le deuxième cours a été consacré à l'étude approfondie des « logiques de Hoare », c'est-à-dire des logiques de programmes qui suivent l'approche introduite par C.A.R. Hoare en 1969. Nous avons défini une telle logique de programmes pour le langage IMP, un petit langage impératif à contrôle structuré que nous avions déjà

étudié dans le cours 2019-2020 « Sémantiques mécanisées ». Nous avons ensuite développé diverses extensions de cette logique : règles de raisonnement dérivées ou admissibles, non-déterminisme, erreurs dynamiques, contrôle non structuré, etc.

La suite du cours a étudié les liens entre la logique de programmes et la sémantique opérationnelle du langage IMP. Nous avons défini et démontré la correction sémantique de la logique : toutes les propriétés d'un programme dérivables par les règles de la logique sont bien vérifiées par toutes les exécutions concrètes du programme. Plusieurs techniques de démonstration ont été esquissées : inductives, coinductives, ou encore par comptage de pas (*step-indexing*).

La complétude est la propriété réciproque de la correction sémantique : toute propriété vraie de toutes les exécutions d'un programme peut-elle s'exprimer comme un triplet de Hoare et se dériver des règles de la logique ? Une logique complète permettrait de décider le problème de l'arrêt. La complétude absolue est donc impossible pour un langage Turing-complet. En revanche, la logique de Hoare satisfait une propriété de complétude relative montrant qu'elle est aussi expressive qu'un raisonnement direct sur les exécutions des programmes, à logique ambiante fixée.

Enfin, nous avons discuté des possibilités d'automatiser une vérification déductive à base de logique de Hoare. À condition de fournir manuellement les invariants des boucles, un calcul de plus faible précondition ou de plus forte postcondition permet de réduire la vérification d'un programme en logique de Hoare à la vérification d'un ensemble d'implications en logique du premier ordre, tâche qui peut être confiée à des démonstrateurs automatiques ou assistés.

Cours 3 – Pointeurs et structures de données : la logique de séparation

Dans le troisième cours, nous avons étudié les structures de données et la vérification de programmes qui les manipulent. Les tableaux sont la plus ancienne des structures de données. Une extension simple de la logique de Hoare avec une règle pour l'affectation à un élément d'un tableau permet de spécifier et de vérifier de nombreux programmes utilisant des tableaux, comme les tris en place.

Les pointeurs, aussi appelés références, permettent de représenter de nombreuses structures de données : graphes, listes chaînées, arbres... Un codage des pointeurs en termes de tableaux globaux, comme proposé par R. Burstall (1972) et développé par R. Bornat (2000), se révèle efficace pour vérifier des algorithmes opérant sur des graphes, mais très lourd pour les algorithmes sur les listes et autres structures chaînées. En effet, il est difficile d'éviter les situations de partage, d'alias et d'interférence qui peuvent invalider ces structures.

C'est en cherchant à décrire et maîtriser ces phénomènes d'interférence que J.C. Reynolds, rejoint ensuite par P. O'Hearn et H. Yang, invente entre 1997 et 2001 la logique de séparation. Cette logique met en avant l'importance du raisonnement

local et des règles d'encadrement associées, le besoin d'associer une empreinte mémoire à chaque assertion, et le concept de conjonction séparante entre assertions.

Nous avons illustré l'approche en développant une logique de séparation pour un petit langage fonctionnel et impératif doté de variables immuables et de pointeurs vers des cases mémoires mutables. Cette logique de séparation permet de décrire très précisément de nombreuses structures de données par des prédicats de représentation : listes simplement ou doublement chaînées, listes circulaires, arbres, etc., et de spécifier et vérifier leurs principales opérations.

Enfin, nous avons esquissé deux démonstrations de la correction sémantique de cette logique de séparation, l'une reposant sur une propriété de non-déterminisme de l'allocation mémoire, l'autre sur une quantification sur tous les encadrements possibles.

Cours 4 – Parallélisme à mémoire partagée : la logique de séparation concurrente

Les processeurs multicœurs sont un exemple d'architecture parallèle à mémoire partagée, où plusieurs unités de calcul travaillent simultanément sur une mémoire commune. La programmation de ces architectures est difficile : il faut maîtriser les interférences possibles entre les actions des processus, et éviter les courses critiques (*race conditions*) entre des écritures et des lectures simultanées.

Quelles logiques de programmes nous permettent de vérifier des programmes parallèles à mémoire partagée ? Pour répondre à cette question, le quatrième cours a introduit la logique de séparation concurrente (CSL, *Concurrent Separation Logic*), une extension de la logique de séparation avec des règles de raisonnement sur le parallélisme et l'exclusion mutuelle.

La logique de séparation décrit très simplement le calcul parallèle sans partage de ressources, où les processus s'exécutent en parallèle sur des portions disjointes de la mémoire. C'est le cas de nombreux algorithmes récursifs sur les tableaux ou sur les arbres, où les appels récursifs s'effectuent sur des sous-arbres ou sous-tableaux disjoints.

La CSL, comme introduite par P. O'Hearn en 2004, ajoute des règles de raisonnement sur les sections critiques permettant à plusieurs processus d'accéder à des ressources partagées à condition que ces accès s'effectuent en exclusion mutuelle. Les ressources partagées sont décrites par des formules de logique de séparation qui doivent être invariantes en dehors des sections critiques. Cela permet de décrire non seulement de nombreux idiomes de synchronisation entre processus, mais aussi les transferts de ressources qui s'effectuent implicitement lors de ces synchronisations.

Nous avons défini une CSL pour le petit langage du cours précédent enrichi de constructions décrivant le parallélisme et les instructions atomiques. Nous avons montré comment construire sur ce langage et cette logique des sémaphores binaires, des sections critiques, et des schémas producteur-consommateur. Enfin, nous avons

montré la correction sémantique de cette CSL en reprenant une démonstration publiée par V. Vafeiadis en 2011.

Cours 5 – Quelques extensions de la logique de séparation

Dans le cinquième cours, nous avons étudié quatre extensions des logiques de séparation des cours précédents qui permettent ou facilitent la spécification et la vérification d'une plus large classe de programmes.

La première extension est l'opérateur d'implication séparante, familièrement appelé « baguette magique » en raison de sa forme, qui est l'adjoint de la conjonction séparante, au même titre que l'implication usuelle est l'adjoint de la conjonction usuelle. Cette « baguette magique » facilite le raisonnement en logique de séparation, notamment *via* la règle de conséquence ramifiée ou *via* un calcul de plus faibles préconditions.

La deuxième extension permet de vérifier des processus qui partagent des données mais y accèdent en lecture seule, sans modifications. Il s'agit d'associer des permissions aux cellules de la mémoire, ces permissions pouvant être partielles (permettant uniquement la lecture) ou complètes (permettant aussi l'écriture et la libération). Deux modèles bien connus de permissions partielles sont les permissions fractionnaires et les permissions comptées. Nous avons illustré l'utilisation de ces dernières pour vérifier un verrou à lecteurs multiples implémenté par deux sémaphores.

Le « code fantôme » est la troisième technique étudiée dans cette séance. Il s'agit de commandes qui ne sont pas exécutées dans le programme final mais contribuent à définir des « variables fantômes » qui simplifient la vérification. Dans le cadre du calcul parallèle, code et variables fantômes permettent de garder trace des calculs faits par chacun des processus et de la contribution de ces calculs individuels à l'exécution globale du programme.

La dernière extension que nous avons décrite permet de stocker en mémoire des verrous et leur invariant de ressources à côté des données protégées par ces verrous. Cela permet de spécifier et de vérifier des algorithmes parallèles à grain fin, comme nous l'avons illustré avec une structure de liste simplement chaînée avec verrouillage couplé.

Aussi disparates qu'elles semblent, ces extensions et bien d'autres sont des cas particuliers d'un petit nombre de notions plus générales, comme le montre l'infrastructure Iris pour les logiques de programmes.

Cours 6 – Logiques pour la mémoire partagée faiblement cohérente

Depuis le quatrième cours, notre vision d'une exécution parallèle d'un programme est celle d'un entrelacement des actions élémentaires des processus constituant le programme. Ce modèle du parallélisme est appelé séquentiellement cohérent (SC). Un avantage de ce modèle est qu'il donne un sens précis aux courses critiques qui

peuvent se produire pendant l'exécution, comme on en trouve dans certains algorithmes parallèles.

Malheureusement, le modèle SC est une fiction : les processeurs et les langages contemporains ne garantissent pas la cohérence séquentielle et fournissent à l'utilisateur des modèles mémoires faiblement cohérents, qui exhibent des comportements impossibles à expliquer par des entrelacements. Ces relaxations de la cohérence séquentielle proviennent autant de dispositifs matériels (tampons d'écriture, exécution spéculative dans le désordre, etc.) que d'optimisations à la compilation (réordonnancements et factorisations de code). Divers mécanismes matériels (barrières) et linguistiques permettent de contrôler ces relaxations.

Comment vérifier les programmes parallèles dans ces conditions ? Pour une large classe de programmes, la logique de séparation concurrente « standard » suffit : les programmes vérifiables dans cette logique n'ayant pas de courses critiques, ils s'exécutent à l'identique dans le modèle SC et dans tout modèle relâché qui présente la garantie DRF (*Data-Race Freedom*), ce qui est le cas de tous les modèles matériels et logiciels connus. Nous avons esquissé une démonstration dans le cas particulier du modèle TSO.

L'étape suivante est de raisonner sur les accès atomiques faiblement cohérents, comme ceux fournis par les standards C/C++ 2011. La logique FSL de V. Vafeiadis et C. Narayan ajoute à la logique de séparation concurrente de nouvelles assertions et des règles de déduction pour les accès atomiques de type *release-acquire* utilisés pour le passage de messages. L'extension aux accès atomiques de type relaxé se heurte au problème des valeurs *ex nihilo* (*values out of thin air*) du modèle C/C++ 2011, problème que résout la sémantique « prometteuse » proposée par J. Kang *et al*. Cela débouche sur la logique SLR de K. Svensden *et al.*, qui ajoute à FSL un traitement complet des accès atomiques relaxés.

Cours 7 – Logiques pour les langages fonctionnels et l'ordre supérieur

Le dernier cours de l'année a étudié les logiques de programmes pour les fonctions, incluant les fonctions d'ordre supérieur et les fonctions comme valeurs de première classe. Les procédures récursives sont l'une des premières extensions de la logique de Hoare. Les règles de raisonnement sont à la fois élégantes dans leur traitement de la récursion et compliquées par de nombreuses restrictions sur l'utilisation et la modification des variables.

Une logique de séparation pour un langage à variables immuables et cases mémoire mutables permet non seulement une formulation plus claire des règles pour les procédures et les fonctions du premier ordre, mais aussi une extension à l'ordre supérieur. Dans cette extension, les triplets de Hoare font partie des assertions de la logique et peuvent donc être utilisés en préconditions de fonctions d'ordre supérieure pour spécifier les arguments de type « fonction », ainsi que dans les prédicats de

représentation pour spécifier les méthodes d'un objet tout en cachant son état interne. La correction sémantique de la règle pour les fonctions récursives pose des problèmes de circularité, qui se résolvent soit par une extension de la technique de comptage de pas, soit par l'utilisation d'une logique modale comprenant la modalité $\triangleright$, « plus tard ».

De nouvelles manières de présenter et d'utiliser les logiques de programmes s'ouvrent à nous dans les logiques d'ordre supérieur et à types dépendants. Le système CFML représente un programme fonctionnel et impératif par sa formule caractéristique : un prédicat d'ordre supérieur qui décrit de manière compositionnelle toutes les préconditions et postconditions valides du programme. Cette représentation est particulièrement adaptée à la vérification interactive ainsi qu'à la spécification des fonctions d'ordre supérieur. Utilisant des types dépendants, les monades de Hoare et de Dijkstra ajoutent aux monades usuelles des préconditions et des postconditions (monades de Hoare) ou un calcul de plus faible précondition (monades de Dijkstra). Le langage F* s'appuie sur une hiérarchie de monades de Dijkstra et sur un typeur qui produit des conditions de vérification pour fournir un excellent environnement de programmation fonctionnelle et impérative vérifiée.

SÉMINAIRES – EN RELATION AVEC LE SUJET DU COURS

Séminaire 1 – Les logiques de programmes à l'épreuve du réel : tours et détours avec Frama-C/WP

Loïc Correnson (CEA), le 11 mars 2021

Dans ce premier séminaire de l'année, il a été question de la mise en œuvre mécanisée des logiques de programmes pour les logiciels écrits dans le langage C au moyen de l'outil Frama-C/WP.

Le conférencier a abordé la difficulté de maîtriser la complexité du calcul de plus faible précondition – *weakest precondition calculus* ou « wp » – et l'importance du modèle mémoire et de la simplification des obligations de preuve. Du point de vue de l'utilisateur, il a présenté les approches méthodologiques permettant de s'attaquer industriellement à la preuve de programmes complexes.

Enfin, il a posé la question de la « qualité » d'une vérification formelle et de sa base de confiance, ce qui lui a permis finalement de revisiter la dualité entre test et vérification formelle.

Séminaire 2 – Preuve auto-active de programmes en SPARK

Yannick Moy (Adacore), le 18 mars 2021

SPARK est une technologie *open source* de preuve de programmes par analyse de flux de données et vérification déductive pour les programmes écrits dans le langage

de programmation Ada. SPARK est codéveloppé par AdaCore, Altran et Inria, et commercialisé par AdaCore dans différents domaines critiques (avionique, militaire, spatial, ferroviaire, automobile, médical).

Dans ce séminaire, le conférencier a présenté les choix qui ont été faits pour le langage de spécification de SPARK et l'outil de preuve associé, les difficultés d'application en pratique, et comment l'approche de preuve auto-active y répond. Il a abordé les améliorations récentes du langage de spécification et de l'outil pour traiter les données partiellement initialisées et les pointeurs sans alias.

Séminaire 3 – *VeriFast: Semi-automated modular verification of concurrent C and Java programs using separation logic*

Bart Jacobs (K. U. Leuven), le 25 mars 2021

Le troisième séminaire de l'année, présenté en anglais, a été consacré à l'outil VeriFast de vérification modulaire et semi-automatique de programmes C et Java à base de logique de séparation.

L'outil prend en entrée des fichiers source C ou Java annotés par des préconditions et postconditions de fonctions et de méthodes, des invariants de boucles, des définitions de structures de données sous forme de prédicats en logique de spécification. Généralement, il retourne en quelques secondes soit « aucune erreur trouvée » (ce qui implique l'absence d'erreurs à l'exécution et la validité des spécifications), soit une trace symbolique d'exécution menant à une erreur.

Le conférencier montré l'outil en action sur de nombreux exemples et présenté ses travaux en cours.

Séminaire 4 – Raisonner à propos du temps en logique de séparation

François Pottier (Inria), le 1er avril 2021

Dans ce séminaire, le conférencier a montré qu'une logique de programmes permet non seulement d'établir la correction d'un programme, mais aussi de contrôler le nombre d'opérations qu'il effectue, et donc, indirectement, son temps de calcul.

Il a expliqué comment ce comptage peut se faire d'abord en logique de Hoare, puis en logique de séparation, où le comptage du temps peut s'effectuer de façon plus décentralisée, et où des artifices comptables très élaborés peuvent être mis en place par l'utilisateur.

Séminaire 5 – Protocoles personnalisés en logique de séparation : ressources fantômes et invariants dans la logique Iris

Jacques-Henri Jourdan (CNRS), le 8 avril 2021

Ce séminaire a été consacré à la logique de séparation Iris. Cette logique, récemment développée avec l'aide de l'assistant de preuve Coq, permet la vérification

de programmes concurrents à grain fin en offrant la possibilité de définir des protocoles personnalisés d'interaction entre fils d'exécution ou entre composants logiciels.

Le conférencier a montré comment définir ces protocoles grâce à deux outils simples et fondamentaux mais néanmoins particulièrement puissants : les invariants et l'état fantôme. Il a décrit ces deux concepts et les règles de raisonnements associées, tout en démontrant à l'aide de plusieurs exemples leur puissance et leur flexibilité.

Séminaire 6 – *Gillian: A multi-language platform for compositional symbolic analysis*

Philippa Gardner (Imperial College London), le 15 avril 2021

Le dernier séminaire de l'année, présenté en anglais, a décrit la plate-forme multi-langages Gillian de vérification par exécution symbolique et ses utilisations pour vérifier et pour trouver des erreurs dans des bibliothèques JavaScript et C. Gillian met en œuvre trois types d'analyses : le test symbolique de programmes complets ; la vérification déductive à base de logique de séparation ; et le test compositionnel automatique à base de bi-abduction. Il repose sur un moteur d'exécution symbolique qui unifie la détection d'erreurs et la vérification formelle.

La conférencière a montré plusieurs applications de Gillian développées dans son équipe : trouver des erreurs dans les bibliothèques de structures de données Buckets.js and Collections-C ; trouver des erreurs et démontrer la correction bornée pour une bibliothèque de type jQuery appelée « cash » ; et vérifier la fonction de désérialisation du système de messages AWS Encryption SDK.

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.

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 2020-2021 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/RA2020/cambium/.

VÉRIFICATION DÉDUCTIVE DE PROGRAMMES

Nous avons développé de nouvelles logiques de programmes, construites au-dessus de la logique Iris, pour deux nouveaux traits du langage OCaml : le parallélisme à mémoire partagée (thèse en cours de Glen Mével) et les effets algébriques avec leurs gestionnaires d'effets (thèse en cours de Paulo de Vilhena). Une première application a été la vérification formelle d'une structure de données concurrente de type « file d'attente » qui tire parti du modèle mémoire faiblement cohérent d'OCaml.

PROGRAMMATION FONCTIONNELLE TYPÉE EN OCAML

Nous avons travaillé avec le OCaml Labs de l'université de Cambridge pour préparer l'intégration définitive dans le système OCaml des deux nouveaux traits du langage mentionnés ci-dessus : parallélisme à mémoire partagée et effets algébriques. En particulier, nous avons revu le traitement des signaux et interruptions asynchrones et ajouté de nouvelles bibliothèques pour la synchronisation. Par ailleurs, nous avons ajouté un mécanisme pour déclarer l'injectivité des constructeurs de types, réduit les temps de compilation par le biais de meilleurs algorithmes d'analyse par flots de données, et accéléré le GC par l'utilisation de *prefetching* de mémoire.

VÉRIFICATION DE COMPILATEURS

Nous avons continué l'étude de la synthèse de code efficace pour les calculs matriciels et tensoriels à partir de spécifications de haut niveau, et de la vérification formelle de cette synthèse. Nous avons publié un article sur la vérification d'un générateur de code séquentiel à partir d'un modèle polyhédrique déja optimisé. Dans le cadre de sa thèse, Basile Clément a inventé un algorithme de validation *a posteriori* capable de vérifier la correction du code synthétisé par le système Halide et par d'autres approches. Enfin, nous avons poursuivi le développement de CompCert, notre compilateur C formellement vérifié, avec la formalisation et la vérification complètes des *bit fields* dans les types structures, et l'amélioration de la passe de *branch tunneling*.

MODÉLISATION ET TEST DE MODÈLES MÉMOIRE FAIBLEMENT COHÉRENTS

Nous continuons l'étude de formalismes axiomatiques et sémantiques pour décrire les modèles mémoire fournis par les processeurs multicœurs et les langages de programmation contemporains. Nous avons étendu les modèles mémoire AArch64 et x86 pour rendre compte des accès mémoires de tailles mixtes ainsi que des interactions avec la mémoire virtuelle.

PUBLICATIONS

Alglave J., Deacon W., Grisenthwaite R., Hacquard A. et Maranget L., « Armed cats: Formal concurrency modelling at Arm », *ACM Transactions on Programming Languages and Systems*, vol. 43, nº 2, 2021, art. 8, https://doi.org/10.1145/3458926.

Bour F., Clément B. et Scherer G., « Tail Modulo Cons », *in* Y. Regis-Gianas et C. Keller (dir.), *JFLA 2021. 32es Journées francophones des langages applicatifs (du 7 au 9 avril 2021 sur l'Internet)*, 2021, p. 48-68, https://hal.science/hal-03190426v2.

Courant N. et Leroy X., « Verified code generation for the polyhedral model », *in* : *Proceedings of the ACM on Programming Languages*, vol. 5, nº POPL, 2021, art. 40, p. 1-24, https://doi.org/10.1145/3434321.

De Vilhena P.E. et Pottier F., « A separation logic for effect handlers », *in : Proceedings of the ACM on Programming Languages*, vol. 5, nº POPL, 2021, art. 33, https://doi.org/10.1145/3434314.

Ladeveze Q., « Mécanisation du modèle RC11 et de la propriété DRF-SC », *in* Y. Regis-Gianas et C. Keller (dir.), *JFLA 2021. 32es Journées francophones des langages applicatifs (du 7 au 9 avril 2021 sur l'Internet)*, 2021, p. 78-94, https://hal.science/hal-03190426v2.

Madiot J.-M., Pous D. et Sangiorgi D., « Modular coinduction up-to for higher-order languages via first-order transition systems », *Logical Methods in Computer Science*, vol. 17, nº 3, 2021, https://doi.org/10.46298/lmcs-17%283%3A25%292021.

Mével G. et Jourdan J.-H., « Formal verification of a concurrent bounded queue in a weak memory model », *Proceedings of the ACM on Programming Languages*, vol. 5, nº ICFP, 2021, art. 66, https://doi.org/10.1145/3473571.

Pottier F., « Strong automated testing of OCaml libraries », *in* Y. Regis-Gianas et C. Keller (dir.), *JFLA 2021. 32es Journées francophones des langages applicatifs (du 7 au 9 avril 2021 sur l'Internet)*, p. 3-20, https://hal.science/hal-03190426v2.

Williams A., *Refactoring Functional Programs with Ornaments*, thèse de doctorat en informatique, sous la dir. de D. Rémy, université de Paris Cité, décembre 2020, https://theses.hal.science/tel-03126602.