Candidature au poste n°4240/0209 de maître·sse de conférences à l'École Normale Supérieure de Lyon

Raphaël Monat

Table des matières

Résumé du dossier

Mots-clés : méthodes formelles, sûreté du logiciel, analyse de programmes, compilation.

Postes occupés : doctorant contractuel (2018–2021) puis ATER temps plein (2021–2022), au LIP6 (Sorbonne Université).

Enseignement : 384h effectuées de 2018 à 2022, dont 162h de la L3 au M2. Thèmes : programmation (Python, C, OCaml, Java), compilation, analyse de programmes, mathématiques discrètes, algorithmique.

Axes de recherche actuels :

- Analyse statique de programmes Python utilisant des bibliothèques C (publications à SAS'21, ECOOP'20, collaborateurs : Antoine Miné et Abdelraouf Ouadjaout),
- Compilateur moderne pour le code des impôts (publication à CC'21 et transfert en cours cf. page 33, collaborateurs : Denis Merigoux et Jonathan Protzenko).

Mes travaux de recherche sont disponibles en libre accès sur mon site `rmonat.fr`.

Mobilité internationale : stages de recherche de M2 à Sarrebruck (Février – Juin 2017) et de M1 à Oxford (Mai – Juillet 2016).

Vie de la communauté :

- membre du collège code source et logiciel libre du comité national pour la science ouverte,
- membre élu du conseil de laboratoire du LIP6, membre du conseil des doctorants du LIP6,
- comité de programme SAS'22,
- comité d'évaluation des artefacts logiciels SPLASH'22, PLDI'22, CAV'22, ECOOP'21, PLDI'21, POPL'21, SAS'20.

Recommandations : vous pouvez contacter :

- Antoine Miné (directeur de thèse, professeur à Sorbonne Université),
- Jonathan Protzenko (co-auteur article CC'21 sur les impôts, chercheur chez Microsoft),
- Nathalie Sznajder (maîtresse de conférence à Sorbonne Université, responsable du cours "mathématiques discrètes" dans lequel je suis intervenu cette année),
- Pierre-Evariste Dagand (chargé de recherche CNRS à l'IRIF, Université de Paris ; membre de mes comités de suivi de thèse).

Raphaël Monat

ATER au LIP6, Sorbonne Université
Mots-clés : Sciences du logiciel, Méthodes formelles, Sûreté du logiciel
23 bd de Brandebourg, 94200 Ivry-sur-Seine
raphael.monat@lip6.fr | rmonat.fr | +33 6 42 33 06 18 | 28/07/1995 | gitlab.com/rmonat

Formations

Thèse sous la direction d'Antoine Miné — LIP6, Sorbonne Université
Analyse statique de type et de valeur, par interprétation abstraite, de programmes Python avec des bibliothèques C. — Septembre 2018 – Novembre 2021

	Antoine Miné	Sorbonne Université, France	Directeur
	Isabella Mastroeni	Università di Verona, Italie	Rapportrice
	Anders Møller	Aarhus Universitet, Danemark	Rapporteur
Jury :	Emmanuel Chailloux	Sorbonne Université, France	Président du jury
	Francesco Logozzo	Facebook Seattle, USA	Examinateur
	Peter Müller	ETH Zürich, Suisse	Examinateur
	Alan Schmitt	Inria Rennes, France	Examinateur

Master 2 Master Parisien de Recherche en Informatique — Université Paris Diderot
Moyenne 16.81/20 — 2017-2018

Master 2 Informatique Fondamentale — École Normale Supérieure de Lyon
Moyenne 16.49/20 — 2016–2017

Master 1 Informatique Fondamentale — École Normale Supérieure de Lyon
Moyenne 16.36/20 — 2015–2016

Licence 3 Informatique Fondamentale — École Normale Supérieure de Lyon
Moyenne 16.43/20 — 2014–2015

Entrée à l'École Normale Supérieure de Lyon sur le concours informatique — École Normale Supérieure de Lyon
2014

Classes préparatoires MPSI/MP* option informatique — Lycée Louis-le-Grand, Paris
2012–2014

Baccalauréat scientifique, mention très bien — Lycée Aristide Bergès, Isère
2012

Expériences professionnelles

Attaché temporaire d'enseignement et de recherche — LIP6, Sorbonne Université
Recherche dans l'équipe APR (dirigée par Antoine Miné). Enseignements dans l'UFR d'ingénierie (192h). — Septembre 2021 - Août 2022

Doctorant contractuel avec mission complémentaire d'enseignement — LIP6, Sorbonne Université
Recherche sous la direction d'Antoine Miné. Enseignements dans l'UFR d'ingénierie (64h/an). — Septembre 2018 - Août 2021

Stage de recherche de M2 sous la direction d'Antoine Miné — LIP6, Sorbonne Université
Création d'une analyse relationnelle de typage pour Python. — Mars – Juin 2018

J'ai défini de manière théorique une analyse de programmes Python proche d'un système de type avec du polymorphisme paramétrique. Cette analyse utilisait un unique domaine abstrait monolithique, qui pouvait exprimer que certaines variables avaient un même type, celui-ci variant dans un ensemble. L'implémentation permettait d'obtenir des résultats sur des petits programmes de moins de 50 lignes de code. Ma première année de thèse a consisté à reprendre ces travaux pour séparer le domaine abstrait en plusieurs domaines indépendants, et améliorer significativement le passage à l'échelle de l'analyse.

Stage de recherche de M2 sous la direction d'Eva Darulova — MPI-SWS, Sarrebruck, Allemagne
Extension d'une formalisation en Coq et HOL4 pour un synthétiseur de programmes sur les flottants. — Février – Juin 2017

Le groupe de vérification automatique et d'approximation (AVA) du Max Planck Institute for Software Systems, dirigé par Eva Darulova, développe un outil appelé Daisy, permettant de compiler des programmes numériques idéalisés, définis sur les réels, en des programmes calculant sur les flottants, avec une marge d'erreur donnée en sortie par le compilateur. Ce compilateur était capable d'optimiser l'équilibre entre précision et performance des programmes grâce à l'utilisation de plusieurs précisions pour les flottants. Un doctorant du groupe, Heiko Becker, avait établit une formalisation (en Coq et HOL) permettant de vérifier via des certificats que la compilation utilisant une analyse d'intervalles et une précision fixée était correcte. J'ai généralisé cette formalisation pour prouver la compilation correcte dans le

cas d'une précision mixte. J'ai aussi travaillé à formaliser une analyse basée sur l'arithmétique affine. Une publication de groupe sur ce travail a été acceptée à FMCAD 2018.

Stage de recherche de M1 sous la direction de Hongseok Yang Oxford, Royaume-Uni

Inférence variationnelle pour les langages de programmation probabilistes. Mai – Juillet 2016

Les modèles probabilistes sont utilisés dans de nombreux domaines pour résoudre différents problèmes, allant de la reconnaissance d'images au diagnostic de maladies. L'utilisation de modèles permet de séparer l'encodage du problème (dans un modèle probabiliste) de sa résolution, et d'avoir des algorithmes d'inférence spécifiques à chaque classe de modèle. Une approche à l'intersection des langages de programmation et de l'apprentissage automatique est la programmation probabiliste. Cette approche permet de généraliser la notion de modèle probabiliste, et a pour but de pouvoir réaliser automatiquement l'inférence à l'exécution des programmes probabilistes. J'ai travaillé avec Hongseok Yang à formaliser les programmes probabilistes sous une forme de système de transition probabiliste. Le problème d'inférence peut être vu comme un problème d'optimisation, que nous avons simplifié pour dériver un nouvel algorithme d'inférence variationnel général pour les programmes probabilistes.

Stage de recherche de L3 sous la direction d'Antoine Miné Antique, ENS Ulm

Développement d'une analyse de programmes concurrents avec une mémoire à cohérence séquentielle. Juin – Juillet 2015

J'ai utilisé le cadre de l'interprétation abstraite pour développer, implémenter et tester de nouvelles méthodes pour analyser plus précisément des programmes concurrents. Le modèle mémoire considéré est celui de la cohérence séquentielle. J'ai étendu une approche d'analyse modulaire sur les threads, développée précédemment par Antoine et basée sur la logique "Rely-Guarantee", pour améliorer la précision de l'analyse dans la prise en compte des interférences entre les threads. J'ai développé à partir de zéro un prototype, appelé Batman, permettant d'analyser des programmes concurrents jouets. Ce prototype est écrit en OCaml et utilise la bibliothèque de domaines abstraits numériques Apron. J'ai comparé les résultats obtenus à ceux d'un autre outil appelé ConcurInterproc. J'ai étendu ces travaux en dehors de mon stage et publié ces résultats avec Antoine Miné à VMCAI 2017.

Enseignements

Sauf mention contraire, les enseignements détaillés ci-dessous ont été effectués à Sorbonne Université, dans l'UFR d'ingénierie, auprès d'étudiants en informatique, et les enseignements en master ont été effectués dans le cadre du master STL "Sciences et Techniques du Logiciel".

Acronymes :

- TME : travaux sur machine encadrés
- CM : cours magistral

Type d'enseignement effectué par niveau

	L1	L2	L3	M1	M2
CM					4h
TD	60h	52h	30h	60.5h	14h
TME	58h	71h	29.5h		24h
Total	118h	123h	59.5h	60.5h	42h

Type d'enseignement effectué par thème

	L1	L2	L3	M1	M2
Info. théorique	20h	83h		52h	
Programmation	98h	40h	31.5h		
Spécialité			28h	8.5h	42h
Total	118h	123h	59.5h	60.5h	42h

Info. théorique : mathématiques discrètes, logique, algorithmique
Spécialité : compilation, analyse et preuve de programmes

L1 **Éléments de Programmation 2 (C)**, Chargé de TD/TME (20 étudiant·e·s, 38.5h) 2021–2022
Correction des examens.

L1 **Éléments de Programmation 1 (Python)**, chargé de TD/TME (26 étudiant·e·s, 38.5h) 2019–2020
Création et correction de 4 interrogations de TD, et correction des examens.

L1 **Éléments de Programmation 1 (Python)**, chargé de TME (29 étudiant·e·s, 19.25h) 2018–2019
Correction d'une épreuve machine. Amélioration des supports.

L1 **Éléments de Programmation 1 (Python)**, remplacement en TD (30 étudiant·e·s, 1.75h) 2018–2019

L1 **Ateliers de Recherche Encadrée**, chargé de TD/TME (4 binômes, 20h) 2018–2019
Création des sujets proposés (arbres bicolores et enveloppes convexes).

L2 **Logique**, Chargé de TD/TME (33 étudiant·e·s, 38.5h) 2021–2022
Correction des examens.

L2 **Mathématiques Discrètes**, chargé de TD/TME (33 étudiant·e·s, 44.5h) 2021–2022
Deux interrogations de TD à préparer et corriger. Correction des examens et des projets. Amélioration des supports.

L2 **Programmation Fonctionnelle (OCaml)**, chargé de TD/TME (26 étudiant·e·s, 19.25h) 2021–2022

Création de 2 sujets de TME et intégration dans la plateforme LearnOcaml. Correction des examens. Amélioration des supports.

L2 **Programmation Fonctionnelle (Ocaml)**, remplacement ponctuel en TME (27 étudiant·e·s, 1.75h) 2019–2020

L2 **Fonctions et Procédures de Calcul (Ocaml)**, chargé de TME (10 étudiant·e·s, 19.25h) 2018–2019
Co-création d'un sujet de projet, et correction du projet.

L3 **Programmation Objet Avancée (Java)**, chargé de TD/TME (27 étudiant·e·s, 31.5h) 2020–2021
Correction des examens. Amélioration des supports.

L3 **Compilation**, chargé de TD/TME (12 étudiant·e·s, 28h) 2018–2019
Correction des examens. Amélioration des supports.

M1 **Algorithmique Avancée**, chargé de TD (31 étudiant·e·s, 32h) 2021–2022
Correction des examens. Amélioration des supports. Correction d'un projet de programmation.

M1 **Algorithmique Avancée**, chargé de TD (31 étudiant·e·s, 20h) 2019–2020
Correction des examens. Amélioration des supports.

M1 **Projet STL**, co-encadrant de binômes sur des projets (8.5h) 2019–2020

M2 **MPRI**[1] **: Interprétation abstraite**, intervenant CM (15 étudiant·e·s, 2h) 2021–2022
Création du support pour mon intervention (résultats obtenus durant ma thèse). **Cours donné en anglais.**

M2 **Typage et Analyse Statique**, chargé de TD/TME (31 étudiant·e·s, 28h) 2021–2022
Refonte des sujets de TD. Correction d'un projet, et oral d'évaluation des étudiants sur des articles de recherche.

M2 **Spécification et Validation de Programmes (Coq)**, chargé de TD/TME (10 étudiant·e·s, 10h) 2021–2022
Création des quatre sujets de TME.

M2 **Groupe de Recherche en Algorithmique et en Programmation**, intervenant CM (12 étudiant·e·s, 2h) 2019–2020
Création du support pour mon intervention (présentation de mon sujet de thèse et de ce qu'est une thèse).

Publications

Dans le domaine des méthodes formelles dont je fais partie, *les articles sont majoritairement publiés dans des conférences internationales avec comité de sélection*. Les articles dans les journaux avec comités de sélection sont plus rares. *Les auteur·rice·s sont usuellement ordonné·e·s par contribution décroissante.* Pour les publications parlant de travaux de groupe sur Mopsa (VSTTE 2019 et JFLA 2021), nous avons choisi d'utiliser un ordre alphabétique. Mes publications sont ordonnées par ordre chronologique décroissant. SAS est la conférence spécialisée dans le domaine de l'analyse statique par interprétation abstraite.

Politique de publication. Je préfère soumettre mes travaux à des conférences qui ont des politiques de publication ouverte avec des frais raisonnables pour les auteurs (proches du prix coûtant, $\leq 100€$), ou qui sont atteignables sans prendre l'avion. ECOOP[2] est la conférence du domaine respectant le plus ces critères, puisqu'elle a lieu en Europe et qu'elle utilise les actes ouverts LIPIcs[3] (les autres conférences du domaine utilisent l'ACM ou Springer comme éditeurs).

Légende :

- Titre en noir : auteur principal (gris sinon).
- Artefacts logiciels : (ACM), (autres).
- article double colonne (sinon).
- article démonstration d'un outil.

Les titres sont cliquables et renvoient aux informations de publication sur mon site (avec des liens vers une version publique et la version de l'éditeur, au minimum) ; les acronymes des conférences pointent vers le programme de l'événement correspondant ; les badges d'artefacts logiciels vers l'artefact validé par les pairs.

Conférences Internationales avec Comité de Lecture

A Multilanguage Static Analysis of Python Programs with Native C Extensions SAS 2021
Raphaël Monat, *Abdelraouf Ouadjaout, Antoine Miné* 23 pages

A Modern Compiler for the French Tax Code CC 2021
Denis Merigoux, Raphaël Monat, *Jonathan Protzenko* 11 pages

Static Type Analysis by Abstract Interpretation of Python Programs ECOOP 2020
Raphaël Monat, *Abdelraouf Ouadjaout, Antoine Miné* 27 pages

1. MPRI : master parisien de recherche en informatique, spécialisé en informatique fondamentale et à destination des étudiants poursuivant dans la recherche. Formation supervisée par l'université de Paris, ENS Ulm, ENS Paris-Saclay, École Polytechnique, Telecom Paris.
2. `https://ecoop.org/`
3. `https://www.dagstuhl.de/en/publications/lipics`

Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer (Invité) — VSTTE 2019
Matthieu Journault, Antoine Miné, Raphaël Monat*, Abdelraouf Ouadjaout* — 16 pages

A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4 — FMCAD 2018
Heiko Becker, Nikita Zyuzin, Raphaël Monat*, Eva Darulova, Magnus O. Myreen, Anthony Fox* — 8 pages

Precise Thread-Modular Abstract Interpretation of Concurrent Programs using Relational Interference Abstractions — VMCAI 2017
Raphaël Monat*, Antoine Miné* — 17 pages

Workshop Internationaux avec Comité de Lecture

Value and Allocation Sensitivity in Static Python Analyses — SOAP 2020
Raphaël Monat*, Abdelraouf Ouadjaout, Antoine Miné* — 6 pages

Conférences Nationales avec Comité de Lecture

Mlang : an Open-Source Toolchain for the Income Tax Computation — JFLA 2021
Denis Merigoux, Raphaël Monat — 2 pages

Démonstration de la plateforme Mopsa d'analyse statique de programmes par interprétation abstraite — JFLA 2021
Matthieu Journault, Antoine Miné, Raphaël Monat*, Antoine Miné* — 2 pages

Étude formelle de l'implémentation du code des impôts — JFLA 2020
Denis Merigoux, Raphaël Monat*, Christophe Gaie* — 16 pages

Manuscrits et rapports techniques

Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries — Manuscrit de thèse
Raphaël Monat — 275 pages

Static Analysis by Abstract Interpretation Collecting Types of Python Programs — Rapport de stage (M2 2018)
Raphaël Monat — 20 pages

Certificate Checking in Coq and HOL4 for Static Analyses of Mixed-Precision Floating-Point Arithmetic — Rapport de stage (M2 2017)
Raphaël Monat — 24 pages

Variational Inference in Probabilistic Programs : Formal Derivation of a Black-box Approach — Rapport de stage (M1)
Raphaël Monat — 26 pages

Thread-Modular Analysis Designing Relational Abstractions of Interferences — Rapport de stage (L3)
Raphaël Monat — 21 pages

Exposés

Le temps indiqué est celui de l'exposé, sans les questions. Titres cliquables vers mon site.

Exposé invité

01/12/21 **A Multilanguage Static Analysis of Python/C Programs with Mopsa**, 25 minutes — **Facebook TAV**

"Testing and Verification (TAV) Symposium brings together academia and industry in an open environment to exchange ideas and *showcase the top experts from testing and verification scientific research and practice*." (source)

Exposés dans des conférences

18/10/21 **A Multilanguage Static Analysis of Python Programs with Native C Extensions**, 15 minutes — SAS
07/04/21 **Mlang : an Open-Source Toolchain for the Income Tax Computation**, 15 minutes — JFLA
03/03/21 **A Modern Compiler for the French Tax Code**, 12 minutes — CC
15/11/20 **Static Type Analysis by Abstract Interpretation of Python Programs**, 15 minutes — ECOOP
15/06/20 **Value and Allocation Sensitivity in Static Python Analyses**, Best Presentation Award. 20 minutes — SOAP
30/01/20 **Étude formelle de l'implémentation du code des impôts**, 20 minutes — JFLA
07/10/19 **Static Type Analysis of Python Programs : A Type Abstract Domain for Python**, 20 minutes — DS@FM

Séminaires

17/03/22	**A Multilanguage Static Analysis of Python/C Programs with Mopsa**, 30 minutes	SRG, Imperial, London
31/01/22	**Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries**, 40 minutes	MTV, LaBRI, Bordeaux
31/01/22	**A Multilanguage Static Analysis of Python/C Programs with Mopsa**, 30 minutes	SyCoMoRES, CRIStAL, Lille
02/12/21	**A Modern Compiler for the French Tax Code**, 40 minutes	IRILL, Paris
29/11/21	**Static Analysis of Python Programs with Native C Libraries**, 30 minutes	CASH, LIP, Lyon
26/11/21	**A Multilanguage Static Analysis of Python/C Programs with Mopsa**, 30 minutes	Binsec, CEA, Saclay
19/11/21	**A Multilanguage Static Analysis of Python/C Programs with Mopsa**, 30 minutes	Celtique, IRISA, Rennes
05/07/21	**A Multilanguage Static Analysis of Python Programs with Native C Extensions**, 20 minutes	LIP6, Paris
12/03/21	**Mopsa, a Multi-lingual Static Analysis Platform**, 20 minutes	GT LVP, GDR GPL
04/11/19	**Formal study of the French tax code's implementation**, 60 minutes	INRIA Cambium, Paris
23/10/18	**Static Analysis by Abstract Interpretation of Dynamic Programming Languages**, 20 minutes	LIP6, Paris

Poster

02/07/19	**Semantics & Static Type Analysis of Python Programs**	Journée doctorants SIF

Participation à la vie de la communauté

Membre du collège code source et logiciel libre du comité national pour la science ouverte

Mars 2022-...

Conseil du laboratoire du LIP6 (membre élu, 1h/mois)

Avril 2021-...

Conseil des doctorants du LIP6 (1h/mois)

Avril 2021-...

Comité de programme

SAS'22

Comité d'évaluation d'artefacts logiciels (30 à 40h par conférence)

SPLASH'22, PLDI'22, CAV'22, ECOOP'21, PLDI'21, POPL'21, SAS'20

Comité de revue externe

SPLASH'22

Relecteur externe

Science of Computer Programming, SAS'21, ACM TECS, SOAP'21, LOPSTR'19

Étudiant bénévole

POPL'17

Logiciels développés

Noms des logiciels cliquables.

Mopsa, plateforme open-source d'analyse statique de programmes — Depuis Septembre 2018

Contributeur principal. 60kLoc Ocaml. Développement et maintenance de l'analyse Python (13kLoc) et multilangage Python/C (2.7kLoc). Licence LGPL v3. Développement suite à l'octroi d'un financement "Consolidator Grant" de l'ERC à Antoine Miné.

Autoévaluation INS2I : `A3 SO4 SM3 EM2 SDL4 DA4 CD4 MS4 TPM4`.

Mlang, compilateur open-source pour le code des impôts — Depuis Mai 2019

Contributeur principal. 10kLoc OCaml. Mlang permet de répliquer le calcul de l'impôt sur le revenu. Licence GPL v3.

Autoévaluation INS2I : `A4 SO2 SM3 EM4 SDL4 DA4 CD4 MS4 TPM4`. Accompagnement de la DGFiP pour la migration d'un ancien compilateur vers Mlang (lien vers communiqué).

Daisy, compilateur de programmes numériques sur les réels vers des flottants. — Février – Juin 2017

J'ai participé au développement de Daisy. Daisy est écrit en 14kLoc Scala.

Autoévaluation INS2I : `A3 SO3 SM3 EM3 SDL4 DA1 CD2 MS1 TPM1`.

FloVer, vérification mécanisée de certificats de correction générés par Daisy. Février – Juin 2017

J'ai généralisé la formalisation de FloVer, qui contient 10kLoc HOL et 25kLoc Coq. Je faisais partie des deux développeurs principaux (avec Heiko Becker) durant ma période de stage. La généralisation effectuée avait augmenté la taille des formalisations de 30% pour Coq et 18% pour HOL. Daisy et FloVer ont été l'objet d'une publication de groupe à FMCAD'18.

Autoévaluation INS2I : `A3 SO3 SM3 EM3 SDL4 DA2 CD2 MS1 TPM1`.

Batman, preuve de concept d'un analyseur statique capable d'analyser les différents threads de manière modulaire. Juin 2015 – Janvier 2017

Seul contributeur. 5,5kLoc OCaml. Licence GPL v3. Ce prototype a été utilisé pour l'évaluation expérimentale de notre papier à VMCAI'17.

Autoévaluation INS2I : `A1 SO2 SM1 EM1 SDL4 DA4 CD4 MS4 TPM4`.

Langues

- Français, langue maternelle
- Anglais, niveau B2 certifié par le CLES
- Allemand, niveau B1

Programmation & outils

- Pratique courante : OCaml, Python, C, LaTeX, Beamer
- Pratique régulière : Java, Bash, GoHugo, OBS (cours à distance et enregistrement de présentations virtuelles), assistants de preuve Coq et HOL4

Engagements associatifs

Animation bénévole de cours d'escrime pour étudiants, 2h à 3h30 par semaine, 40 semaines par an 2018–2020

Suivi d'une formation fédérale à l'animation de cours d'escrime, 70 heures 2018–2019

Médaille de bronze lors de la finale régionale (SWERC) du concours de programmation ACM ICPC, 2017

Rang 12/75 (médailles jusqu'à la place 14), équipe ENSL2 avec Simon Mauras et Jean-Yves Franceschi.

Délégué lors du M1 d'Informatique Fondamentale à l'ENS de Lyon 2015–2016

Projet d'enseignement

Mon projet d'enseignement est centré sur mes connaissances du domaine des langages de programmation que j'ai acquise durant mon doctorat et mon ATER. Cela concerne en premier lieu l'enseignement de divers paradigmes de programmation (impératif haut et bas niveau, objet, fonctionnel), la compilation et l'analyse de programmes. L'ATER que j'effectue cette année m'a permis d'appréhender une charge d'enseignement similaire à celles des maître·sse·s de conférences. J'ai pu enseigner ces dernières années à tous les niveaux, de la L1 de Sorbonne Université aux M2 du Master Parisien de Recherche en Informatique. Je connais bien le type de formation dispensée par le département d'informatique de l'ENS de Lyon, puisque j'y ai été étudiant de 2014 à 2017.

Compilation. En cas d'intégration dans le DI, je pourrais m'investir dans le cours de "compilation et analyse de programmes" dispensé en M1, et géré par Gabriel Radanne. Cela permettrait de pérenniser progressivement l'enseignement de ce cours, qui est actuellement assuré par des chercheurs CNRS & INRIA de l'équipe CASH du LIP. J'ai déjà des expériences (à la fois en enseignement et en recherche) dans la compilation.

Programmation et développement logiciel. Je pourrais naturellement intervenir dans les UEs de "Programmation", "Projet fonctionnel", "Enseignement en programmation sportive" et "Projet intégré". Les trois langages de programmation utilisés au DI (C, Python, OCaml) coïncident avec ceux que je connais le mieux, de par mon enseignement et ma recherche.

L'UE de programmation propose une progression très libre via un système de "fiches", qui me semble adapté à l'hétérogénéité de niveau des étudiants entrant en L3. Je pense néanmoins que certaines parties techniques fondamentales (en particulier la gestion mémoire et les pointeurs en C), gagneraient à être évalués sur papier, afin de s'assurer d'une compréhension mentale et individuelle des concepts.

Je pourrais aussi participer au cours d'"Enseignement en programmation sportive", dont le but est d'implémenter efficacement des algorithmes, notamment pour préparer au concours ACM ICPC. J'ai participé deux fois en tant qu'étudiant du DI au concours SWERC (finale régionale du concours ACM ICPC), et remporté une médaille de bronze en équipe en 2017.

Je pourrais également intervenir dans le "Projet intégré" de M1 pour renforcer l'enseignement des bonnes pratiques autour du logiciel intégré dans la nouvelle monture de l'UE. Ce cours est le seul du cursus où les étudiant·e·s sont amenés à travailler collectivement – le travail étant très individuel en classe préparatoire, et les projets de L3 étant effectués seul ou en binôme. Je pourrais enseigner les bonnes pratiques de développement, avec l'utilisation d'un système de version collaboratif tel que git, la mise en place de procédures d'évaluation de code par les pairs via des "merge requests" et l'utilisation de l'intégration continue pour éviter toute régression dans le logiciel développé. Additionnellement, je suis très investi dans ma communauté de recherche à évaluer la reproductibilité des résultats obtenus par des logiciels, via la participation à des comités d'évaluations d'artefacts logiciels dans des conférences internationales. J'ai par ailleurs soumis trois artefacts logiciels dans des conférences internationales [4, 8, 10], qui ont tous été évalués par les pairs au plus haut niveau suivant la catégorisation ACM [78].

Préparation à l'agrégation. Dans le cadre de la préparation au concours de l'agrégation en informatique, je pourrais prendre en charge la formation à l'épreuve de travaux pratiques, ainsi que les leçons sur le thème des langages de programmation – ou codiriger cette partie avec l'agrégé·e préparateur·rice recruté·e dès septembre 2022 [35]. Les langages de programmation au programme correspondent à ceux choisis par le DI (C, Python, OCaml). Je préparerai les candidat·e·s aux aspects pédagogiques liés au code : relecture, commentaire, comparaison et amélioration du code déjà écrit (cf. section 2 du sujet 0 [45]).

Enseignements en systèmes, réseaux et architecture. Je suis volontaire pour participer aux enseignements en systèmes, réseaux et architecture à terme, même si je n'ai pas d'expertise issue de mon domaine de recherche. Dans tous les cas, je pense qu'il serait très intéressant d'étudier des rapprochements possibles entre le cours de système et celui de compilation. Cela pourrait commencer par l'utilisation d'un même jeu d'instruction dans les cours de système et de compilation, mais d'autres parallèles sur la représentation mémoire et la gestion des registres seront intéressants à établir.

Nouvelle UE "sécurité du logiciel". À terme, je pourrais proposer un cours de sécurité du logiciel, au deuxième semestre du M1 ou en M2. Celui-ci serait orienté dans la détection de bogues, avec une forte composante sur l'utilisation et la création d'outils basés sur différentes techniques de fuzzing et d'analyse de programme, et pourrait être créé en collaboration avec des membres de l'équipe CASH.

Futur des enseignements au DI

Les étudiant·e·s de L3 issus de classes préparatoires ont une très grande hétérogénéité dans la pratique de l'informatique. Certain·e·s sont des autodidactes très à l'aise avec la programmation (qui peuvent avoir participé au concours Prologin ou aux Olympiades d'Informatique) et il y a à l'opposé des personnes qui n'ont programmé que pendant les rares TPs d'informatique de prépa MPSI/MP. L'ouverture des classes prépas MP2I/MPI (respectivement en septembre 2021 et 2022) – qui consacre un nombre significatif d'heures à l'informatique – va augmenter encore l'hétérogénéité des promotions (l'ENS de Lyon s'est engagée à consacrer la moitié des effectifs du concours informatique via la filière MPI [34]). Je suis volontaire pour participer à une réflexion sur les adaptations du programme de L3 à prévoir, puisque le programme de MP2I/MPI [52] aborde un certain nombre de concepts actuellement couverts en L3 (algorithmes sur les graphes, algorithmes d'approximation, induction structurelle, bases de logique et bases de la gestion en C de la mémoire, des fichiers et de la concurrence). Une harmonisation légère pourrait se faire via un stage de rentrée.

Approche de l'enseignement

Je fais particulièrement attention à créer une atmosphère bienveillante et inclusive lors de mes cours, dans l'optique de faire progresser tou·te·s les étudiant·e·s. En particulier, j'évite de préjuger de la facilité des exercices donnés, je prête attention à utiliser des noms épicènes, tels que "parent" plutôt que "père" dans le contexte d'algorithmes sur les arbres, et je n'utilise pas de connivences de la culture geek qui pourraient exclure certains groupes ou renforcer des stéréotypes nuisible à la diversité. Une collègue m'a récemment fait découvrir les travaux d'Isabelle Collet, et je compte améliorer mes enseignements suivant la notion de "toile de l'égalité" décrite dans une de ses publications récentes [22].

Dans les UEs contenant des travaux pratiques longs (ou des projets), j'incite les étudiant·e·s à utiliser les outils d'intégration continue proposés par Gitlab et GitHub. Cela permet d'introduire une bonne pratique logicielle aux étudiant·e·s, tout en automatisant le rendu et une partie de la correction lorsqu'une base de tests publique est disponible. J'ai déjà utilisé ce mécanisme dans les UEs "Programmation Objet Avancée" et "Typage et Analyse Statique" que j'ai enseigné ces deux dernières années. Le rendu des TPs de l'UE de compilation et analyse de programmes est déjà fait via la plateforme Moodle de l'ENS de Lyon, mais aucun test automatique n'est actuellement déployé.

Responsabilités administratives

Je serais heureux de participer au fonctionnement du DI (ainsi que celui du LIP) en assurant des responsabilités administratives. Un des souhaits exprimés par le DI est de pouvoir suivre le devenir de ses anciens étudiants. Cela me paraît très important, notamment pour affiner la formation dispensée par le DI selon les débouchés actuels. Il serait intéressant de quantifier les débouchés hors enseignement supérieur et recherche, la durée des thèses, ainsi que les formations doctorales qui ont été les plus utiles. Je propose pour cela de mettre en place un suivi individualisé des étudiants, respectivement 1 an, 4 ans et 7 ans après la sortie du DI. Cela pourrait être fait en prenant contact directement avec les anciens étudiants, ou en désignant des responsables par domaine, qui pourraient être en contact avec ces étudiants lors de journées nationales. Ces durées sont choisies pour correspondre en moyenne à un début de thèse, une fin de thèse, et potentiellement un début de poste permanent dans le cas d'un cursus universitaire.

J'ai déjà pris des responsabilités collectives précédemment : je suis membre élu (pour le collège des non-permanents) du conseil de laboratoire du LIP6 et membre du conseil des doctorants du LIP6 depuis avril 2021. J'ai aussi été délégué lors du M1 d'informatique fondamentale au DI. Au niveau national, je suis membre du collège code source et logiciel libre du comité national pour la science ouverte, créé au printemps 2022.

Activité de recherche

Vue d'ensemble

Le but de mes recherches est d'améliorer la qualité des logiciels. J'utilise pour cela le domaine des **méthodes formelles**, afin de définir précisément le comportement des programmes. Cela permet de raisonner rigoureusement sur ces comportements (pour déceler par exemple des états dangereux). Dans certains cas, il est possible d'automatiser le raisonnement pour qu'un ordinateur diagnostique les comportements qui nous intéressent, via des **analyses de programme**.

J'aspire à développer et appliquer des méthodes aux systèmes les plus réalistes possibles. D'un point de vue méthodologique, je commence par sélectionner des programmes (le plus souvent, sur GitHub) et des bogues qui seront visés par mon analyse. Cette première étape est partiellement automatisée via des scripts. Je développe ensuite mes approches **à la fois de manière théorique**, pour obtenir une formalisation rigoureuse, **et via une implémentation**, afin de valider l'approche sur les programmes retenus lors de la première étape. Au cours de ces quatre dernières années, j'ai étudié **deux systèmes réalistes, avec des collaborateurs distincts.**

Le premier système est le **langage de programmation Python**, qui est très populaire. C'est en particulier le **deuxième langage le plus utilisé** (après JavaScript) sur GitHub [38]. Ce système est l'objet des recherches effectuées durant ma thèse avec Antoine Miné et Abdelraouf Ouadjaout. J'ai en grande partie **formalisé la sémantique du langage Python**, sur papier. Cette sémantique a été établie en parcourant le code de l'interprète de référence, CPython, dont le noyau est écrit en 160 000 lignes de code C. J'ai développé des **analyses statiques de programmes sûres et précises** pour le langage Python. En particulier, j'ai développé une **analyse multilangage**, permettant de détecter des bogues dans les programmes utilisant à la fois Python et C. Cette analyse répond à un besoin réel : les programmes sont souvents écrits avec plusieurs langages. Ce problème d'analyse multilangage a été très peu abordé précédemment. Ces analyses ont été implémentées dans la **plateforme open-source Mopsa**, écrite en **60 000 lignes de code OCaml**, et dont je suis un des développeurs principaux.

Le second système est le **code de calcul de l'impôt sur le revenu des particuliers**. Le code source de ce système est public depuis avril 2016, mais j'ai fait partie des premiers scientifiques du domaine à s'y être interessé en 2019. Ce travail de recherche a été effectué en parallèle de ma thèse, en collaboration avec Denis Merigoux (alors doctorant dans Prosecco, Inria Paris), et illustre ma **capacité à conduire des travaux de recherches indépendamment de ma direction et de mon laboratoire de thèse**. Ce système est critique : il concerne 38 millions de foyers fiscaux français, et rapporte 30% des recettes de l'État chaque année. Nos recherches ont permis de **rendre le calcul** de l'impôt sur le revenu publiquement **reproductible** : le code était disponible en ligne dans un langage (appelé "M") propre à l'administration et non documenté. Nous avons pour cela mené un travail de rétro-ingénierie de la sémantique de M, avant de la **formaliser en Coq**. Nous avons aussi développé un **nouveau compilateur open-source (10 000 lignes de code OCaml)** permettant de répliquer le calcul de l'impôt sur le revenu. Je supervise actuellement un **transfert du compilateur** vers la direction générale des finances publiques, en **encadrant le travail de trois développeurs**.

Mon projet de recherche aborde deux thématiques distinctes. J'aimerais développer des analyses statiques de programmes qui sont à la fois **sûres et accessibles** – en terme de précision et d'efficacité – pour être **massivement adoptées** par les développeur·euse·s. Je souhaite aussi appliquer le domaine des méthodes formelles à une classe de programmes assez peu étudiés alors qu'ils ont des impacts sociétaux majeurs : les **implémentations de codes juridiques**.

Une partie de mes recherches utilise le cadre de l'**interprétation abstraite** pour créer des **analyses statiques sûres et automatiques** : toutes les propriétés dérivées par l'analyse sont effectivement valables sur le programme, et l'analyse termine sa tâche de manière autonome, en un temps fini. Ces analyses sont dites statiques car elles n'exécutent pas le programme, afin de pouvoir **raisonner sur toutes les exécutions possibles** (potentiellement en nombre infini). Ces analyses sont cependant incomplètes et peuvent donner lieu à de **fausses alarmes**, lorsque le système ne peut pas prouver qu'un comportement donné est correct. Un objectif est donc d'avoir des systèmes suffisamment précis pour réduire le plus possible le nombre de fausses alarmes. Le cadre de l'interprétation abstraite fournit une méthodologie pour obtenir des analyses statiques. Le point de départ est la **sémantique concrète** du langage étudiée, décrivant un interprète calculant les états accessibles d'un programme donné. Cet interprète ne termine pas en temps fini (par exemple, à cause des boucles). Cette sémantique est donc modifiée pour obtenir une sémantique abstraite, décrivant un interprète calculant une **sur-approximation** des états accessibles en temps fini. Cet interprète va utiliser différents **"domaines abstraits"** pour gérer les traits du langage analysé. Ces approches ont connu un succès particulier pour prouver l'absence d'erreurs à l'exécution dans le contexte des logiciels embarqués critiques. Par exemple, Astrée [25], a été utilisé pour prouver l'absence d'erreurs à l'exécution dans les logiciels de contrôle et de commande des avions Airbus [30].

Analyse statique de programmes Python, utilisant des bibliothèques C

Cette section présente deux de mes contributions effectuées pendant ma thèse : l'analyse de programmes Python, et l'extension de cette analyse pour pouvoir prendre en compte le code C utilisé par certaines bibliothèques Python. J'ai implémenté ces analyses dans la plateforme Mopsa, en cours de développement au LIP6, suite à l'octroi d'un financement "Consolidator Grant" de l'ERC à Antoine Miné. Ce développement est décrit en détail dans une section spécifique.

Le cas des langages dynamiques. Les langages dynamiques sont de plus en plus populaires : JavaScript et Python sont les deux langages les plus utilisés sur GitHub actuellement [38]. Ces langages comportent plusieurs traits les rendant différents des langages tels que C et Java souvent ciblés par les analyseurs statiques. Ils sont typés dynamiquement : les variables n'ont pas besoin d'être déclarées ou typées statiquement. Au cours de l'exécution d'un programme, une variable peut pointer vers différents objets ayant des types différents. Ils ont une structure dynamique d'objet, où il est possible d'ajouter ou de supprimer des champs aux objets durant l'exécution. Ces deux traits dynamiques sont complétés par des opérateurs d'introspection, qui permettent d'inspecter le type ou la structure d'un objet durant l'exécution, et ainsi affecter le flot de contrôle du programme. Ainsi, une analyse de langage dynamique doit être capable d'inférer le type des variables à n'importe quel point de l'exécution, et supporter la structure dynamique d'objet ainsi que les opérateurs d'introspection afin d'être précise. Les langages dynamiques ont par ailleurs des sémantiques très permissives, qu'il faut capturer précisément et exprimer fidèlement dans les analyses pour que celles-ci soient sûres. Peu de domaines abstraits ont été développés pour analyser les structures de données offertes par ces langages, telles que les tableaux dynamiques et les dictionnaires. Le flot de contrôle du programme peut aussi être complexe à analyser lorsque des fonctions asynchrones (ou les générateurs de Python) sont utilisés.

Contribution : analyse de programmes Python

Ma première contribution est la conception d'une analyse sûre et précise pour détecter les erreurs à l'exécution des programmes Python réalistes. Dans le cas de Python, ces erreurs sont toutes les exceptions (générées par l'interprète Python ou le programme exécuté) qui ne sont pas rattrapées et qui vont donc interrompre l'exécution du programme. L'analyse supporte les comportements de Python utilisés en pratique par les développeur·euse·s (structure dynamique d'objet, opérateurs d'introspection, mutabilité des objets, générateurs), afin de pouvoir analyser des programmes Python réalistes.

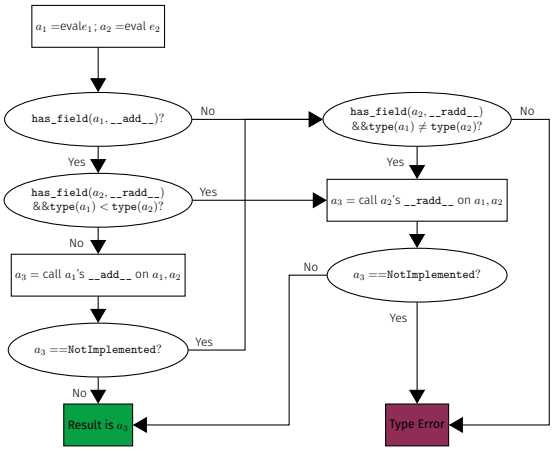
Les différentes analyses de cette contribution seront présentées sur l'exemple en figure 1. Cet exemple définit une classe `Task`, ayant un constructeur `__init__` qui prend en argument un nombre `p` désignant la priorité de la tâche. Ce constructeur lève une exception `ValueError` si la priorité est négative, et la sauvegarde dans le champ `p` de l'instance sinon. Une méthode de comparaison, notée `__lt__` est aussi définie. La fonction `pos_max` parcourt l'objet donné en argument via une itération explicite afin de trouver le maximum et son indice de première apparition. Cette fonction est ensuite appelée sur une liste contenant de 10 à 20 instances de `Task`, chacune ayant une priorité entre 0 et 10 (le non-déterminisme est utilisé pour illustrer plusieurs exécutions). Ce programme ne comporte pas de bogue et nos analyses vont essayer de le prouver.

Le langage Python étant dynamique, j'ai d'abord défini une analyse de type. Cette analyse est capable de détecter précisément les erreurs de type et d'accès à des champs. Pour cela, elle infère précisément vers quel type d'objet chaque variable peut pointer, ainsi que les champs définis pour chaque objet. Dans notre exemple, l'analyse est ainsi capable d'inférer que la comparaison à la ligne 13 est correcte lors de l'appel de la fonction `pos_max` à la ligne 17. En effet, `maxi` et `l[i]` sont des instances de la classe `Task`, donc la fonction `Task.__lt__` sera appelée, et les deux instances ont bien un attribut de priorité `p`, résolus à chaque fois en

FIGURE 1 : Position du maximum

```
1  from random import randint
2
3  class Task:
4    def __init__(self, p):
5      if p < 0: raise ValueError
6      self.p = p
7    def __lt__(self, other):
8      return self.p < other.p
9
10 def pos_max(l):
11   maxi, pos = l[0], 0
12   for i in range(1, len(l)):
13     if maxi < l[i]:
14       maxi, pos = l[i], i
15   return maxi, pos
16
17 m, p = pos_max([Task(randint(0, 10))
18   for x in range(randint(10, 20))])
```

FIGURE 2 – Diagramme d'évaluation de l'addition de e_1 et e_2 en Python



des entiers, qui sont des objets comparables entre eux. Par contre, cette analyse n'est pas assez précise pour prouver que : (i) l'exception `ValueError` ne sera pas levée ligne 5, car aucune information numérique sur la valeur de `p` passée à `__init__` n'est connue de l'analyse, (ii) les accès aux listes lignes 11, 13 et 14 sont corrects (l'analyse n'infère aucune information sur les tailles des listes). Il y aura donc des fausses alarmes sur l'exception `IndexError`.

J'ai ensuite raffiné la précision de l'analyse de type en ajoutant de l'inférence d'informations numériques, par exemple grâce à des intervalles, pour obtenir ainsi une analyse de valeur. Cela permet d'enlever la fausse alarme de la ligne 5 : dans les appels au constructeur de `Task`, on sait que `p` est dans l'intervalle $[0, 10]$. Pour gagner en précision, il faut ensuite ajouter un domaine qui abstrait la taille des listes en introduisant des variables auxiliaires dans le domaine numérique. Ainsi, `len(l)` est une variable du domaine des intervalles désignant la taille de la liste `l`. L'ajout de ce domaine permet de supprimer la fausse alarme de la ligne 11 : lors de l'appel à `pos_max` de la ligne 17, `l` est une liste non-vide et donc l'accès est correct. Cela permet aussi d'inférer que dans la boucle for à la ligne 12, `i` est dans l'intervalle $[1, 20]$ et `len(l)` est dans l'intervalle $[10, 20]$. Ces informations ne permettent néanmoins pas de conclure que l'accès à la liste ligne 13 est correct. Il faut pour cela ajouter des *domaines relationnels* – tels que les polyèdres [23] – pour pouvoir inférer que $1 \leq$`i`$<$`len(l)` et donc que l'accès à la liste `l` est correct. Le support des domaines relationnels exige une attention particulière dans la conception d'un analyseur statique : intuitivement, la communication via valeurs abstraites (tels que les intervalles) ne fonctionne pas dans un cadre relationnel. Ces domaines sont au centre de l'architecture de Mopsa et sont supportés par cette analyse.

J'ai aussi ajouté un mécanisme permettant d'interpréter les annotations de type de Python [66] dans nos analyses [3, section 6.2] [13, section 7.4]. Cela permet de réutiliser les annotations de type de la bibliothèque standard fournies par le projet Typeshed [64], et ainsi supporter rapidement certaines dépendances.

Toutes ces analyses ont été évaluées sur des programmes réels, tels que les benchmarks de performance [61] utilisés par les développeur·euse·s de l'interprète de Python, ou un utilitaire développé par Facebook appelé `PathPicker` [74]. Sur ces benchmarks, l'analyse de type analyse au total plus de 9 000 lignes de code [1] en moins de 4 minutes, celle utilisant des intervalles en 13 minutes [5].

Originalité et difficulté. Les analyses statiques sûres de langages dynamiques étaient concentrées sur le langage JavaScript [43]. Celles-ci ne font pas d'inférence d'informations numériques (intervalles, polyèdres), qui sont nécessaires pour des analyses précises de Python. Un travail précédent sur l'analyse de Python avait été réalisé par Fromherz et al. [36]. Ce travail utilisait un prototype non public qui ne pouvait pas analyser des programmes faisant plus de 300 lignes de code. Mon but était d'être plus modulaire (en proposant le choix entre une analyse de type et une analyse numérique), d'obtenir un meilleur passage à l'échelle, via un support étendu du langage, tout en fournissant une implémentation robuste et maintenable à long terme.

La première difficulté rencontrée dans ce travail était la compréhension de la sémantique du langage Python. Le langage n'est pas standardisé, et l'interprète principal, CPython, sert de référence. Par ailleurs, Python est un langage extrêmement flexible, avec beaucoup de cas exceptionnels à gérer. J'ai donc beaucoup parcouru le code de CPython (dont le noyau est écrit en 160 000 lignes de C) afin de m'assurer de comprendre la sémantique dans les moindres détails. Pour illustrer la complexité de cette sémantique, la figure 2 est une représentation simplifiée d'une opération basique de Python : l'addition. Suivant les champs des objets en lesquels e_1 et e_2 sont évalués, ainsi que la relation entre leurs types, la méthode `__add__` du premier objet peut être appelée, ou la méthode `__radd__` du second objet. Les autres opérations du langage (accès aux attributs des objets, appels de méthode, ...) sont similairement complexes et comportent de nombreux cas qu'il faut modéliser fidèlement pour obtenir une analyse sûre du langage. Python possède aussi un double système de type : un système nominal basé sur l'héritage entre classes, et un système structurel, surnommé "duck typing", basé sur les attributs. Ce double système est géré par deux domaines abstraits complémentaires.

Cette sémantique extrêmement permissive rend nécessaire la connaissance précise du contexte d'appel pour arriver à analyser des fonctions. Par exemple, la fonction `pos_max` de la figure 1 accepte en entrée n'importe quel objet supportant une itération explicite, dont les accès renvoient des objets comparables entre eux. Pour être précis, l'analyse des fonctions est faite par inlining, en analysant le corps de la fonction appelée dans son contexte d'appel. Cela est néanmoins très coûteux et entrave le passage à l'échelle de l'analyse. La création d'approches compositionnelles permettant de réutiliser les analyses de fonctions dans différents contextes d'appels est un des objectifs de mon projet de recherche.

Pour pouvoir analyser des programmes, il a fallu supporter une partie significative du langage et de la bibliothèque standard. Ce support a requis la combinaison de nombreux domaines abstraits, afin de gérer par exemple l'allocation dynamique, les listes, les attributs des objets. Certains domaines existaient déjà dans la littérature ou étaient connus de manière folklorique, mais j'ai dû les adapter, les combiner et les implémenter dans le cadre nouveau de notre analyseur.

Diffusion. La sémantique concrète du langage Python n'est pour le moment que décrite sur papier, sans être exécutable, dans mon manuscrit de thèse [13, pages 101 à 146]. Cette description est accompagnée de liens systématiques vers le code source de CPython afin de la rendre auditable. Un des travaux futurs décrit dans mon projet de recherche est de rendre cette sémantique exécutable. L'analyse de type a été présentée à ECOOP [3], cet article est joint au dossier et accompagné d'un artefact logiciel évalué par les pairs [4]. Une comparaison de l'analyse de type et l'analyse non-relationnelle de valeurs a été présentée à SOAP [5], où j'ai reçu le prix de la meilleure présentation [77]. Le passage à des analyses relationnelles, nécessitant un groupement heuristique des variables, est précisé dans mon manuscrit [13, section 8.3].

Répartition du travail. J'ai effectué ces recherches en collaboration avec Abdelraouf Ouadjaout et Antoine Miné. Je suis en charge de l'analyse de Python dans la plateforme d'analyse statique Mopsa. J'ai rédigé l'article ECOOP (en dehors de l'introduction) et intégré les remarques de mes coauteurs. J'ai entièrement rédigé l'article SOAP et intégré les remarques de mes coauteurs. Je me suis occupé des présentations (en particulier, enregistrements vidéo et réponses aux questions) des deux événements, qui étaient virtuels.

Contribution : analyse de programmes combinant Python et C

Les programmes modernes mélangent de plus en plus souvent différents langages. Cela permet aux développeur·euse·s de combiner les atouts de différents langages et de réutiliser des bibliothèques écrites dans d'autres langages. Nous avons pu mesurer que 20% des 200 bibliothèques Python les plus téléchargées utilisent du code C. Cela permet souvent d'écrire du code Python haut niveau, appelant lui-même du code C plus efficace. Bien qu'utile, l'interopérabilité est une source additionnelle de bogues : les développeur·euse·s doivent prendre en compte des mécanismes de sécurité et des représentations de la mémoire différents suivant les langages. Ces mécanismes sont par ailleurs rarement supportés de manière sûre par les analyseurs statiques, qui se contentent fréquemment d'ignorer les comportements induits par l'autre langage.

J'ai développé une analyse pour des programmes mélangeant l'utilisation de C et de Python. Cette analyse détecte les erreurs d'exécution dans le code C natif (opérations de pointeurs invalides, dépassement d'entiers, ...), dans le code Python (exceptions levées), et à la frontière entre les langages (conversion incorrecte entre les langages, non-concordance des signaux d'erreur, ...). Il est ainsi possible d'analyser directement et de manière entièrement automatique les codes sources Python et C. J'ai implémenté cette analyse de manière modulaire : elle réutilise des analyses C et Python prêtes à l'emploi écrites dans le même analyseur, Mopsa. Cette approche permet le partage entre les domaines abstraits de différents langages, tels que celui gérant l'allocation dynamique, ou les abstractions numériques (intervalles, polyèdres), ce qui permet d'inférer des relations entre les valeurs des différents langages.

FIGURE 3 : Code client Python

```
1  import counter
2  import random
3
4  c = counter.Counter()
5  p = random.randrange(128)
6  c.incr(2**p-1)
7  c.incr()
8  r = c.counter
```

FIGURE 4 : Extrait d'une bibliothèque C

```
9  typedef struct {
10   PyObject ob_base;
11   int count;
12 } CounterO;
13
14 static PyObject*
15 CounterIncr(CounterO *self, PyObject *args)
16 {
17   int i = 1;
18   if(!PyArg_ParseTuple(args, "|i", &i))
19     return NULL;
20   self->count += i;
21   Py_RETURN_NONE;
22 }
23
24 static PyMemberDef CounterMembers[] =
25  {{"counter", T_INT, offsetof(CounterO,
26    count), READONLY, ""}, {NULL}};
```

Un exemple de code multilangage est présenté en figures 3 et 4 (code complet dans [7, figure 1]). Comme les entiers Python ont une précision arbitraire, mais les entiers C sont bornés, ce code contient plusieurs bogues. La bibliothèque C définit un module `counter` dans lequel une classe `Counter` est définie. Du côté C, les instances de cette classe seront des structures `CounterO` (lignes 9-12). Le code client Python peut créer des instances de ce compteur (ligne 4); incrémenter le compteur (par défaut de 1, ou une autre valeur entière peut être passée) : les lignes 6-7 appelleront la fonction C `CounterIncr`; accéder à la valeur du compteur, la ligne 8 sera résolue grâce à un attribut intelligent déclaré lignes 24-26. Suivant la valeur choisie par `p` à l'exécution, trois comportements peuvent se produire : (i) si $0 \leq \texttt{p} \leq 30$, $\texttt{r} = 2^{\texttt{p}}$, (ii) lorsque $\texttt{p} = 31$, la deuxième incrémentation (ligne 7) va créer un dépassement d'entier silencieux C lors de l'addition (ligne 20), et le résultat sera $\texttt{r} = -2^{31}$ – ce qui sera très surprenant pour des développeur·euse·s Python qui ne connaissent pas C, (iii) si $\texttt{p} \geq 64$, l'entier Python `2**p-1` sera trop grand pour être converti en `int i` ligne 18 : une exception `OverflowError` sera levée. L'analyse que j'ai développée est capable de diagnostiquer tous ces cas.

1. Les mesures de taille de code présentées ont été faites avec l'outil `cloc`, qui ne prend pas en compte les commentaires et les lignes vides.

Pour cette approche, nous avons utilisé comme benchmarks 6 bibliothèques disponibles sur GitHub, ayant en moyenne 412 étoiles. La plus grosse bibliothèque (5 700 lignes de code Python et C) est analysée en moins de 5 minutes.

Originalité et difficulté. La plupart des travaux du domaine se cantonnent à l'analyse d'un seul langage. Les seuls travaux précédents se sont concentrés sur le cas des programmes mélangeant Java et C [37, 46, 62, 67]. De manière générale, ces approches essayaient de réduire le problème de l'analyse multilangage Java/C à une analyse de Java, en traduisant *certains* effets du code C sous forme d'annotations Java. Ces approches ne sont pas sûres : elles ne prennent pas en compte tous les comportements du langage C, ni toutes les interactions possibles entre les deux langages. Par opposition, notre travail est le premier publié à effectivement analyser complètement les deux langages sources utilisés dans les programmes (sauf le ramasse-miettes qui n'est pas supporté actuellement). J'ai dû inspecter en détail le code source du mécanisme d'interopérabilité de CPython afin de le modéliser.

Diffusion. Ce travail a été présenté cette année à SAS, la principale conférence d'analyse statique [7]. Cette publication est jointe au dossier et accompagnée d'un artefact qui aussi a été évalué par les pairs [8]. J'ai été invité au "Facebook TAV" [2]cette année [73] pour présenter mes travaux.

Répartition du travail. J'ai entièrement développé et implémenté l'analyse multilangage (2 700 lignes) lors de ma dernière année de thèse. J'ai ensuite rédigé totalement l'article SAS sur ce sujet, intégré les remarques de mes coauteurs, et je me suis occupé de la présentation lors de la conférence virtuelle. Celle-ci réutilise l'analyse de Python (définie précédemment), et l'analyse de C implémentée par Abdelraouf Ouadjaout et Antoine Miné [54].

Production logicielle : plateforme d'analyse statique Mopsa

Mopsa est une plateforme d'analyse statique développée au LIP6 depuis 2016, suite à l'octroi d'un financement "Consolidator Grant" de l'ERC à Antoine Miné. Le développement est open-source et une version publique [15] est disponible depuis Octobre 2020. J'ai participé au développement de Mopsa, et intégré mes analyses dans cette plateforme. J'ai accordé une importance particulière à maintenir mes implémentations, ainsi qu'à vérifier que nos benchmarks passaient toujours au fil des modifications.

Originalité et difficulté. L'implémentation de Mopsa vise à explorer de nouvelles perspectives pour la conception des analyseurs statiques. Mopsa a un triple objectif :
— supporter l'analyse de plusieurs langages (C et Python à l'heure actuelle). Les autres analyseurs sûrs, tels qu'Astrée [18], Frama-C [28] et TAJS [43] sont spécialisés dans l'analyse d'un seul langage.
— permettre aux développeur·euse·s de définir des domaines abstraits de manière modulaire – c'est-à-dire aussi indépendamment les uns des autres que possible. Frama-C et TAJS sont implémentés de manière monolithique, où il est difficile de remplacer une abstraction par une autre gérant un même effet. Une fois les différents effets du langage séparés, cette approche modulaire permet de définir les analyses en couches successives, facilitant ainsi la maintenance. Certaines couches sont partagées entre les langages, ce qui permet de factoriser l'implémentation et de simplifier le support de nouveaux langages.
— permettre aux différents domaines abstraits de coopérer et communiquer de manière relationnelle. Astrée est le seul autre analyseur avec un support étendu des analyses relationnelles, en particulier pour gérer le masquage bit à bit ou les abstractions de tableaux avec des variables auxiliaires [20]. Mopsa permet de composer plus profondément différents domaines, en autorisant des domaines à partager l'utilisation de domaines sous-jacents [2, pages 9-10], [13, sections 2.5 et 3.3.1]. Cette composition permet schématiquement de passer à un arbre de domaines utilisé par les analyseurs classiques tels qu'Astrée [25, figure 1] à un graphe acyclique dirigé [13, figure 3.2], afin d'améliorer notamment la précision des analyses.
Ces objectifs ont nécessité de développer et mettre en œuvre une nouvelle approche lors de la définition et l'implémentation des domaines abstraits.

Diffusion. Nous avons présenté Mopsa dans un article invité à une conférence internationale, VSTTE [2]. Nous avons aussi soumis un article court décrivant l'utilisation de Mopsa dans une conférence nationale, les JFLA [12]. Une présentation détaillée de Mopsa est aussi disponible dans le chapitre 3 de ma thèse [13]. Mopsa est disponible publiquement sur Gitlab [15], tout comme les benchmarks (mentionnés précédemment, ainsi que d'autres benchmarks créés par Abdelraouf et Antoine pour l'analyse de C) que nous avons utilisés

2. "Testing and Verification (TAV) Symposium brings together academia and industry in an open environment to exchange ideas and *showcase the top experts from testing and verification scientific research and practice*.", source : `https://research.fb.com/blog/2021/10/registration-now-open-for-the-2021-testing-and-verification-symposium/`

[16]. J'ai aussi créé des artefacts logiciels comprenant les benchmarks, Mopsa, et des scripts pour reproduire automatiquement l'évaluation expérimentale pour mes publications à ECOOP sur l'analyse de type [4] et SAS sur l'analyse multilangage [8]. Autoévaluation INS2I : `A3 SO4 SM3 EM2 SDL4 DA4 CD4 MS4 TPM4`.

Répartition du travail. Abdelraouf Ouadjaout est le développeur principal du cœur de Mopsa, dont les grands principes ont commencé à être établis en collaboration avec Antoine Miné et Matthieu Journault, avant le début de ma thèse et principalement dans une optique d'analyse de programmes C. Ces principes ont été raffinés et éclaircis durant mes travaux sur Mopsa, et l'ajout des analyses Python dans la plateforme. Ils sont décrits dans une publication de groupe [2]. Mopsa est écrit en 60 000 lignes d'OCaml. J'ai développé l'analyse Python dans le cadre fixé par Mopsa. Je suis le principal développeur de la partie permettant l'analyse de programmes Python (13 000 lignes), et le seul développeur de la partie multilangage Python/C (2 700 lignes). J'ai aussi participé au développement et à la maintenance du reste de Mopsa. En particulier, j'ai mis en place une intégration continue de l'analyseur, et participé au développement d'un mécanisme de "hooks" [13, section 3.4] facilitant le débogage.

Un compilateur moderne pour le calcul de l'impôt sur le revenu

J'ai effectué ce travail en parallèle de ma thèse, avec des collaborateurs différents : Denis Merigoux (alors en thèse dans l'équipe Prosecco, INRIA Paris) et Jonathan Protzenko (Microsoft Research).

Contexte

Suite à la loi "République Numérique" de 2016 [76], la direction générale des finances publiques (DGFiP) avait rendu public le code permettant de calculer l'impôt sur le revenu des particuliers [75, 32]. La publication consistait seulement en la base de code, écrite dans le langage M propre à la DGFiP. Elle n'était ni accompagnée de documentation expliquant la sémantique du langage M, ni d'un compilateur permettant d'exécuter la base de code. Il n'était donc pas possible de reproduire le calcul de l'impôt sur le revenu.

Contribution

Nos recherches ont permis de rendre le calcul de l'impôt sur le revenu publiquement reproductible.

Nous avons pour cela d'abord mené un travail de rétro-ingénierie afin de découvrir la sémantique du langage M, commencé en avril 2019. Nous avons d'abord utilisé des simulateurs disponibles publiquement, puis nous avons pu utiliser des jeux de tests confidentiels transmis par la DGFiP en août 2019. En septembre 2019, nous avons remarqué qu'une partie du code source était manquante en inspectant certains tests. Suite à des négociations et la signature d'un accord de confidentialité, nous avons pu accéder au code manquant en juin 2020. La DGFiP avait écrit une partie de son code en C. Cette partie peut lire et écrire les variables utilisées dans la base de code M en utilisant un état global partagé. Ces fichiers C étaient entremêlés avec les fichiers M compilés en C. Nous avons donc développé un langage spécifique, appelé M++, afin de pouvoir remplacer cette base de code C. Cela nous a permis d'en tirer plusieurs avantages. Tout d'abord, M++ est un langage adapté permettant une écriture beaucoup plus compacte : 100 lignes de code M++ permettent d'implémenter 6 000 lignes de code C. Ensuite, l'utilisation de M++ permet d'être indépendant d'un langage de programmation, et de pouvoir compiler le code vers d'autres sources que le C (qui est un besoin de certains services de la DGFiP). Enfin, la DGFiP ne souhaitait pas publier le code C, en invoquant des raisons de sécurité. Par opposition, le code M++ équivalent, que nous avions écrit, est maintenant public.

La sémantique du langage M a été formalisée dans l'assitant de preuve Coq [63]. Cela nous a permis de nous assurer que tous les cas étaient formalisés. Nous avons aussi défini en Coq un système de typage sur M afin de séparer les scalaires des tableaux, et nous avons prouvé que le système de type était sûr : si une expression M est bien typée dans un certain contexte de typage Γ, alors son exécution donnera une valeur dans un contexte d'exécution Ω cohérent avec Γ.

Nous avons écrit un compilateur, appelé Mlang, permettant de compiler cette base de code M/M++ vers différents langages tels que Python, C et Java. Ce compilateur implémente des optimisations usuelles, et des optimisations spécifiques à la sémantique de M/M++, afin de réduire la taille de la base de code de 80%. Grâce à des outils de fuzzing, nous avons aussi pu générer nos propres cas de tests, ayant une meilleure couverture que les cas confidentiels de la DGFiP. Ces cas de tests générés ont aussi été rendus publics.

Ce projet est un succès car la DGFiP a décidé de remplacer leur compilateur actuel par Mlang, et d'utiliser Mlang en production d'ici 2023 (cf. [72] et le contrat d'engagement joint dans les pièces supplémentaires).

Originalité et difficulté. Notre travail se concentre sur une base de code réelle, qui calcule l'impôt sur le revenu de 38 millions de foyers fiscaux chaque année, et rapporte 75 milliards d'euros, soit 30% des recettes annuelles de l'État. L'étude et l'implémentation de codes à portée juridique ouvre un nouveau domaine prometteur, que j'aimerais explorer dans mes travaux futurs.

Nous avons rencontré des difficultés pour effectuer la rétro-ingénierie de M avec comme seul oracle des cas de tests, sans trace d'exécution. Les interactions avec la DGFiP (par exemple, pour avoir accès à l'entièreté de la base de code) ont nécessité de travailler sur le temps long tout en étant pédagogue avec la hiérarchie, qui a une formation juridique. La création du langage M++ n'a été possible que suite à l'étude approfondie de la base de code C de plusieurs milliers de lignes que nous voulions remplacer. Avoir une sémantique bien définie de M et M++ permettra de créer des outils de vérification formelle sur ce code dans un futur proche.

Diffusion. Ce travail a fait l'objet d'une publication dans une conférence internationale de compilation, Compiler Construction [9] (jointe au dossier), accompagnée d'un artefact logiciel validé par les pairs [10], ainsi que de deux publications [6, 11] dans une conférence nationale de la communauté (les JFLA). Mlang [14] consiste en 10 000 lignes de code OCaml, et 600 lignes de formalisation Coq sur la sémantique de M. Autoévaluation INS2I : `A4 SO2 SM3 EM4 SDL4 DA4 CD4 MS4 TPM4`. Un *transfert technologique* est en cours auprès de la DGFiP [72] depuis juin 2021, afin que la DGFiP puisse remplacer son compilateur actuel par Mlang. Je suis engagé par la DGFiP depuis janvier 2022 pour faciliter la transition, les tâches précisées sont définies dans l'article 2 du contrat d'engagement, jointe aux pièces supplémentaires du dossier. J'encadre ainsi le développement de trois personnes (deux ingénieurs de recherche prestataires, issus d'OCamlPro, et un inspecteur programmeur système d'exploitation, fonctionnaire de la DGFiP), en dirigeant les choix techniques à mettre en œuvre et en passant en revue le code qu'ils écrivent. Le temps consacré à cette mission est estimé à 30 jours entre janvier et août 2022. La recherche sur les codes à portée juridique ouvre un nouveau domaine prometteur, illustré par la création d'un workshop spécifique "Programming Languages and the Law" à la conférence internationale POPL cette année.

Répartition du travail. Denis Merigoux (Prosecco, Inria Paris) et moi avons collaboré à parts égales, comme l'illustre nos positions de co-premiers auteurs dans l'article publié à Compiler Construction [9]. Nous avons effectué ces recherches en parallèle de nos thèses respectives. Jonathan Protzenko, un des directeurs de thèse de Denis, a contribué à la rédaction et partagé son regard critique. Denis a découvert la base de code de la DGFiP, et a initié son travail de rétro-ingénierie fin février 2019 (parsing, inférence de type) ; je l'ai rejoint début mai 2019. J'ai développé la formalisation Coq du langage M, j'ai conçu le langage M++, ainsi que les optimisations du compilateur spécifiques à M/M++. Je me suis occupé de présenter notre travail à la conférence virtuelle. Cette recherche a été menée avec l'approbation mais sans la direction d'Antoine Miné.

Projet de recherche avec intégration dans l'équipe CASH : analyses statiques précises, sûres et efficaces pour les logiciels généralistes

Au début des années 2000, des chercheur·euse·s ont développé Astrée, un analyseur statique par interprétation abstraite qui a prouvé que les commandes de vol des Airbus A340 & A380 n'avaient pas d'erreurs à l'exécution [18, 24]. Depuis, des analyseurs de code moins spécialisés, tels que Facebook Infer [31, 33] et Google Tricorder [56], ont été développés, en particulier par les GAFAM. Ces outils fonctionnent sur des codes plus généralistes, passent à l'échelle sur des bases de code gigantesques, et les résultats d'analyse sont compréhensibles et utilisés par des développeur·euse·s. Néanmoins, ces analyseurs se comportent plutôt comme des détecteurs de bogues et ne sont pas sûrs : ils peuvent passer outre certains bogues. Par ailleurs, les propriétés qu'ils essayent d'établir ne sont pas clairement énoncées.

Mon but est de réunifier ces deux approches, afin d'obtenir des analyses statiques de programmes qui sont à la fois sûres et accessibles (en terme de précision et d'efficacité) pour être massivement adoptées par les développeur·euse·s. Mon premier axe de recherche se concentre sur l'amélioration de la sûreté des analyses statiques. Je souhaite pour cela définir des méta-langages adaptés à la formalisation sémantique des langages de programmation, desquels des outils sémantiques (tels que des analyses automatiques et sûres) peuvent être dérivés automatiquement. Dans un second axe, je souhaite rendre les analyses de programmes plus utilisables, en améliorant leur passage à l'échelle et en les combinant avec d'autres méthodes pour discriminer la présence de bogues potentiels. Ces méthodes seront développées de manière générale, afin de pouvoir les appliquer à différents langages de programmation analysés.

Un troisième axe de recherche, orthogonal aux précédents, se concentre sur l'application des méthodes formelles à une classe de programmes assez peu étudiés jusqu'ici, alors qu'ils ont des impacts sociétaux critiques : les implémentations de codes juridiques. Par exemple, l'impôt sur le revenu des particuliers rapporte autour de 75 milliards d'euros par an, soit 30% des recettes de l'État. S'assurer que ces implémentations sont maintenables et correctes est un enjeu majeur.

Mon intégration se ferait naturellement dans l'équipe CASH du LIP, spécialisée dans la compilation optimisée, la sémantique et l'analyse de programmes. Je partage certains intérêts de recherche avec les équipes PLUME et AriC, tels que la vérification de logiciels, et le développement d'outils automatisés – potentiellement certifiés – pour les flottants.

Science ouverte. En rapport avec ma politique de publication développée dans mon CV, j'attache une importance particulière à rendre les logiciels développés disponibles publiquement (sous licence libre), et l'évaluation expérimentale de mes articles reproductible (via par exemple Zenodo). Les artefacts liés à des articles publiés ont aussi fait l'objet d'une évaluation par les pairs [4, 10, 8]. Cette position rejoint celle de l'équipe CASH, qui est attachée à développer et contribuer à des logiciels de recherche sous licence libre.

Formalisation et analyse sûre de sémantiques complexes

L'interprétation abstraite fournit une méthodologie pour obtenir des analyses statiques, en les dérivant depuis la sémantique du langage cible, appelée sémantique concrète. Les analyseurs de programmes peuvent être vus comme des interprètes sur des "domaines abstraits" spécifiques, calculant de manière efficace une abstraction des valeurs accessibles pour chaque variable. Un analyseur est sûr lorsque la sémantique concrète du langage analysé est sur-approximée par l'interprète abstrait. Hormis Verasco [44], qui est un analyseur statique prouvé correct de bout en bout en Coq, les preuves de sûreté sont effectuées sur papier, ce qui réduit leur garantie de sûreté.

Sémantique formelle de Python. Le langage Python, qui est le deuxième langage le plus utilisé sur GitHub, n'est pas standardisé et ne possède actuellement pas de formalisation sémantique réaliste, qui se conforme aux tests de l'interprète de référence, CPython. En tirant profit de mon expertise sur la sémantique de Python acquise durant ma thèse, je souhaiterais développer à court terme une sémantique formelle de Python, en utilisant un formalisme adapté. Je prévois de travailler avec Yannick Zakowski pour définir la sémantique de Python via les arbres d'interaction [70]. Ce formalisme a l'avantage de pouvoir extraire automatiquement un interprète de la sémantique formalisée. Ma connaissance de la sémantique d'un langage dynamique tel que Python consoliderait les expertises actuelles de l'équipe en sémantique de langages fonctionnels (Gabriel Radanne) et de modèles de concurrence (Ludovic Henrio).

Une fois le comportement de Python modélisé par la sémantique, il faut encore vérifier que cette modélisation est correcte en la comparant à l'interprète de référence sur des tests. Je souhaite passer outre la limitation des travaux précédents sur Python (le seul accepté par des pairs étant [55]), qui n'avaient pas une sémantique suffisamment complète pour l'évaluer sur les tests de CPython. De manière duale, je souhaite

pouvoir générer automatiquement des jeux de tests pour chacun des cas définis dans le formalisme sémantique. Cela permettrait d'obtenir des tests spécialisés, et une couverture systématique de la modélisation. Pour cela, il faudrait arriver à générer des cas de tests satisfaisant des contraintes sur l'exécution de la sémantique, ce qui est un problème difficile. Je compte d'abord tester une approche basée sur le fuzzing (plus efficace dans certains cas [47]) de la sémantique interprétée, ou utiliser de l'exécution symbolique [48] sinon.

Synthèse d'analyses à partir de sémantique concrète. Il serait intéressant d'arriver à synthétiser partiellement des analyses à partir de la sémantique concrète, pour avoir une méthode générale, indépendante du langage, et une garantie de sûreté par construction. Cette approche aurait l'avantage d'être moins coûteuse que la preuve de sûreté de bout en bout faite pour Verasco [44]. Si les différents effets du langage (exceptions, allocation mémoire) ont bien été séparés dans la formalisation de la sémantique concrète, il sera possible de synthétiser seulement certains domaines abstraits et de fournir manuellement des abstractions pour gérer ces effets de manière plus efficace. Cette séparation des effets est un des éléments clés de Mopsa que j'ai l'habitude de pratiquer suite à mes travaux. Dans son travail sur LLVM IR [71], Yannick Zakowski sépare la sémantique de LLVM IR de ses effets, qui sont encodés comme des événements observables et définis séparément. Ce cadre théorique semble ainsi être une base solide pour la synthèse d'analyses proposée ici. La preuve de sûreté d'une analyse composée de domaines générés automatiquement et de ceux ajoutés manuellement pourra bénéficier des recherches présentées dans le prochain paragraphe. À plus court terme, la synthèse d'outils de slicing [68] – basés sur des analyses de dépendances plus simples que les analyses numériques effectuées durant mes travaux antérieurs – pourrait être un premier projet dans cette direction. Mon but est d'arriver à générer automatiquement des parties d'analyses sûres en se basant sur un modèle sémantique du langage, afin d'obtenir une technique générale.

Définitions modulaires de domaines abstraits. Les analyses statiques se composent de différents domaines abstraits complémentaires qui gèrent les différents effets du langage. La sûreté des analyses est exprimée via l'utilisation d'une fonction de concrétisation, qui permet de définir un état de l'analyse comme l'ensemble concret des états de programme correspondants. Les fonctions de concrétisation ainsi que les preuves de sûreté sont souvent faites sur papier, de manière monolithique, sur une combinaison spécifique de domaines abstraits. Il serait intéressant de rendre ces preuves plus modulaires, en considérant un domaine abstrait à la fois, afin de réduire les efforts de preuve lorsqu'un domaine change. La nouvelle forme de composition permise par Mopsa, où des domaines peuvent partager l'utilisation de domaines sous-jacents, nécessite de définir les concrétisations de manière modulaire afin que celles-ci soient correctes [13, section 2.4.6]. Cette composition est présente dans l'implémentation de Mopsa depuis 2018, mais l'écriture des concrétisations a longtemps été une question récurrente dans notre équipe de développement de Mopsa. J'ai enfin pu obtenir une forme satisfaisante pour la rédaction de mon manuscrit de thèse en 2021 [13]. Les preuves modulaires de sûreté ne sont pas encore établies et sont un objectif de recherche à moyen terme. Le cadre des arbres d'interaction permet de définir différentes interprétation d'une même sémantique via des raffinements [71, figure 5]. Cette approche est très attractive et pourrait être généralisée afin d'abstraire progressivement une sémantique concrète.

Rendre les analyses statiques plus utilisables

Je compte développer des techniques permettant d'améliorer le passage à l'échelle des analyses statiques basées sur l'interprétation abstraite, en réutilisant des invariants inférés précédemment, ou en inférant des contrats pour les fonctions. J'aimerais aussi développer des techniques de génération de contre-exemples lorsqu'un bug potentiel est détecté. Ces techniques permettront d'améliorer l'attractivité des analyses statiques auprès des développeur·euse·s. Ces techniques ne seront pas spécifiques à un langage de programmation, afin de pouvoir les intégrer dans différentes plateformes d'analyse statique, et de les évaluer simultanément sur plusieurs langages si Mopsa est utilisé. Je partage avec cette équipe un fort attachement à faire passer à l'échelle les analyses statiques. Je pourrais renforcer l'expertise en interprétation abstraite de l'équipe, qui était une des spécialités de Laure Gonnord, maintenant membre externe de CASH et Professeure des Universités à l'ESISAR (Valence) depuis septembre 2021.

Analyses incrémentales. Les analyseurs statiques actuels combinent une partie faisant de l'inférence (par exemple d'invariants de boucles), avec la propagation de ces invariants dans la suite de l'analyse. Un même programme peut être analysé de manière répétée (dans le cas de l'intégration continue) : il serait intéressant de découpler inférence et propagation. Cette propagation reviendrait à faire de la vérification de programmes, qui est un problème strictement plus simple en termes de calculabilité que de l'analyse [26]. La sauvegarde des invariants, par exemple grâce à des formules logiques, permettrait de les appliquer aux analyses suivantes et

de réduire significativement le temps d'analyse des programmes. Additionnellement, l'utilisation de formules logiques simplifierait la communication avec des solveurs SMT. Cette approche s'inspire de la vérification par certificats, sur lequel j'ai travaillé en stage de M2 à Sarrebruck [1].

Les analyses statiques précises ont recours à des domaines relationnels, tels que les polyèdres [23] ou les octogones [51], qui permettent d'exprimer des relations linéaires entre des variables. Ces domaines sont néanmoins coûteux, avec une complexité au moins cubique en le nombre de variables utilisées. Une technique standard pour réduire leur coût est de remplacer un domaine relationnel par une union de domaines relationnels travaillant sur des groupements restreints de variables [18, section 3.9.6]. Ces groupements sont définis de manière heuristique et propres à des classes de programmes analysés. Je propose d'évaluer à court terme des techniques de raffinement itératif du groupement de ces variables sur des analyses successives, afin d'obtenir une approche générale. La première itération consisterait en une analyse non relationnelle. À la fin de chaque itération, une phase de diagnostic permettrait de détecter des relations nécessaires à l'amélioration de la précision, et de créer de nouveaux groupements. Il serait idéal de faire l'évaluation expérimentale de cette technique dans la plateforme Mopsa, afin de vérifier que cette technique est bien indépendante du langage analysé. Christophe Alias est un expert de la compilation optimisée grâce au modèle polyédrique. Il serait intéressant d'explorer avec lui le transfert de travaux récents sur les polyèdres dans le domaine de l'analyse statique [59], ainsi que dans des modèles sous-polyédriques [58, 65]. La création d'analyses incrémentales pourra aussi être étudié dans le cadre de la compilation optimisante de programmes.

Inférence de contrats pour les analyses de fonctions. La plupart des analyseurs précis actuels (Astrée [18], Frama-C [28]) agissent par inlining et analysent les fonctions à chacun des sites d'appel, ce qui est coûteux. Différentes approches infèrent des contrats pour les fonctions, qui peuvent donc être réutilisés. Plusieurs travaux se concentrent sur les fonctions numériques [19, 39, 40], et un autre est dédié à l'analyse de fonctions avec des pointeurs partagés dans la pile d'appel [60]. Ces analyses sont néanmoins trop spécialisées pour analyser des programmes généraux. La seule exception est une approche récente mêlant logique de séparation et analyse statique [42], capable d'analyser 3 000 lignes du code C de l'éditeur de texte Emacs. Cette approche nécessite néanmoins une intervention manuelle d'un utilisateur expert. Les analyses précédentes visent toutes des langages impératifs simplifiés proches de C. Dans le cas des langages de programmation dynamiques, le fort polymorphisme des fonctions rend leur analyse compositionnelle, sans aucun contexte d'appel, extrêmement difficile. J'envisage de développer des approches partiellement compositionnelles (comme Illous et al. [42]), basées sur de l'inlining, mais capables de généraliser les résultats d'analyses obtenus sous la forme de contrats pouvant être réutilisés a posteriori. Je commencerai par développer de nouvelles abstractions génériques et symboliques de l'état mémoire (les abstractions actuelles sont basées sur les sites d'allocation [17]). Un point de départ pourrait être le travail de Cox et al. [27], actuellement spécialisé pour le langage JavaScript. Ma connaissance précise d'analyse de langages différents (C et Python) sera un atout pour le développement d'une solution générique. Les travaux passés [40, chapitre 5] de Matthieu Moy dans le cadre des analyses compositionnelles de fonctions seraient une source d'inspiration pour mon projet d'inférence de contrats pour les fonctions. La résolution de ce problème est un but ambitieux, nécessitant un travail à long terme.

Génération de contre-exemples. Certaines analyses statiques – telles que celles développées durant mes travaux précédents – font des sur-approximations des états de programme accessibles afin de pouvoir terminer. Cela peut se traduire par la présence de fausses alarmes : l'analyse imprécise détecte une erreur potentielle à un certain point, alors que le programme est correct à cet endroit. Trop de fausses alarmes peuvent mener les développeur·euse·s à abandonner l'utilisation d'outils d'analyse statique [21]. Je souhaite explorer l'utilisation de techniques sous-approximantes (telles que l'exécution symbolique ou le fuzzing), guidées par les résultats des analyses statiques, afin de chercher des contre-exemples réels. La combinaison de fuzzing et d'analyse statique a déjà été utilisée dans des cas restreints, pour guider le fuzzing de parseurs [57] ou de smart contrats [69]. La combinaison d'analyses statiques par interprétation abstraite et de solveurs SMT est une piste intéressante à explorer. Le cas du guidage d'analyse statique par interprétation abstraite via des solveurs SMT a été étudié par Laure Gonnord et Matthieu Moy [53, 41].

Méthodes formelles pour les codes juridiques

Les implémentations liées aux codes juridiques – tels que le code calculant l'impôt sur le revenu en France, existant depuis 1990 – ont longtemps été privées, et donc non étudiées. Les administrations françaises sont dans l'obligation de publier les codes sources qu'elles utilisent depuis 2016 [76]. Le code de calcul de l'impôt sur le revenu des particuliers a été le premier publié, et l'objet de mes travaux précédents dans ce domaine [9, 6, 12]. L'accès à des bases de code réelles rend ce champ assez nouveau, mais l'impact potentiel de ces recherches est très large et propice à être compris par un grand nombre. Mon expérience actuelle de travaux en

commun avec la Direction Générale des Finances Publiques (DGFiP) me permettra de dialoguer efficacement avec d'autres administrations, dans le but de créer de nouveaux partenariats.

De la loi vers le code, avec Catala. La structure spécifique des textes de loi rend difficile leur implémentation [50, section 2]. Additionnellement, la surcharge fréquente de certains textes par des nouveaux au fil des années complexifie la maintenance des implémentations, d'autant plus lorsqu'il n'y a pas de correspondance structurelle entre le code et les textes. Dans le cas de l'impôt sur le revenu, la structure n'est pas préservée dans le code M/M++, et 30% des 90 000 lignes de code ont été modifiées pour mettre à jour l'implémentation de 2019 à 2020. Des langages de programmation adaptés à l'encodage de la loi permettraient de faciliter ces implémentations, tout en augmentant la confiance qu'elles implémentent bien la spécification (la loi). Mes collaborateurs des travaux précédents – Denis Merigoux et Jonathan Protzenko – ont ainsi développé un nouveau langage prometteur, appelé Catala [50]. Je fais partie depuis le début de l'année 2022 de l'équipe de développement de Catala [49]. Cette équipe est internationale et multidisciplinaire; elle est dirigée par Denis Merigoux (équipe Prosecco, Inria Paris). Nous souhaitons vérifier que Catala peut passer à l'échelle sur des vraies bases de code tout en simplifiant la maintenance des codes juridiques. En particulier, la DGFiP souhaiterait porter le calcul de l'impôt sur le revenu, actuellement écrit en M et M++, en utilisant Catala. Ce projet est ambitieux et nécessitera un travail au long cours. Le portage sera fait de manière progressive, afin de pouvoir vérifier que l'implémentation est correcte au fil de l'eau, au coût du développement d'une solution interopérable avec la base de code actuelle. Les langages à domaines spécifiques sont une des spécialités de Gabriel Radanne, et pourront donc faire l'objet de collaboration dans ce domaine.

Vérification de la loi encodée avec Catala. De manière générale, les textes de loi consistent en une clause générale, qui peut être surchargée par des cas exceptionnels. J'aimerais implémenter des techniques permettant de vérifier que les différents cas traités par un texte de loi ne créent pas d'ambigüités entre ceux-ci. Ces techniques pourraient s'intégrer dans la plateforme de preuve pour Catala proposée par Delaët et al. [29] lors d'un workshop. Posséder des techniques de décision automatiques sur ces cas permettra aussi de simplifier la compilation des programmes Catala. La structure des programmes Catala est différente des langages usuels, car chaque article de loi peut réutiliser d'autres articles en modifiant partiellement leur contexte. Catala est actuellement défini formellement par des traductions successives dans différents langages intermédiaires, jusqu'à un langage fonctionnel très simple. Une première étape de ce travail est de définir une formalisation sémantique de Catala afin de disposer de fondations solides pour développer des analyses de programmes. La sémantique de Catala ayant des aspects fonctionnels, une collaboration avec Gabriel Radanne serait naturelle. De manière similaire à ce qui a été proposé pour Python en début de projet, nous pourrions effectuer une formalisation sémantique de Catala via les arbres d'interaction développés par Yannick Zakowski.

Bibliographie

Mes contributions sont décrites en début de bibliographie.

[1] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. C. J. Fox. A verified certificate checker for finite-precision error bounds in coq and HOL4. In *Formal Methods in Computer-Aided Design*, pp. 1–10, 2018.

[2] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Verified Software : Theories, Tools, Experiments (VSTTE)*, pp. 1–18, 2019.

[3] R. Monat, A. Ouadjaout, and A. Miné. Static type analysis by abstract interpretation of python programs. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 1–29, 2020.

[4] R. Monat, A. Ouadjaout, and A. Miné. Static type analysis by abstract interpretation of python programs (artefact). *Dagstuhl Artifacts Ser.*, pp. 11 :1–11 :6, 2020.

[5] R. Monat, A. Ouadjaout, and A. Miné. Value and allocation sensitivity in static Python analyses. In *ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP)*, pp. 8–13, 2020.

[6] D. Merigoux, R. Monat, and C. Gaie. Étude formelle de l'implémentation du code des impôts. In *31ème Journées Francophones des Langages Applicatifs*, pp. 31–46, 2020.

[7] R. Monat, A. Ouadjaout, and A. Miné. A multilanguage static analysis of python programs with native C extensions. In *Static Analysis Symposium (SAS)*, pp. 323–345, 2021.

[8] R. Monat, A. Ouadjaout, and A. Miné. A Multi-Language Static Analysis of Python Programs with Native C Extensions - Artefact, July 2021. URL `https://doi.org/10.5281/zenodo.5141314`.

[9] D. Merigoux, R. Monat, and J. Protzenko. A Modern Compiler for the French Tax Code. In *Compiler Construction (CC)*, pp. 71–82, 2021.

[10] D. Merigoux, R. Monat, and J. Protzenko. A Modern Compiler for the French Tax Code - Artefact, January 2021. URL `https://doi.org/10.5281/zenodo.4456774`.

[11] D. Merigoux and R. Monat. Mlang : an open-source toolchain for the income tax computation. In *32ème Journées Francophones des Langages Applicatifs*, pp. 155–156, 2021.

[12] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Démonstration de la plateforme mopsa d'analyse statique de programmes par interprétation abstraite. In *32ème Journées Francophones des Langages Applicatifs*, pp. 45–47, 2021.

[13] R. Monat. *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries*. PhD thesis, Sorbonne Université, France, 2021. 275 pages.

[14] D. Merigoux and R. Monat. Mlang compiler. `https://github.com/MLanguage/mlang`, 2021. Accessed : 2021-12.

[15] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. MOPSA : modular open platform for static analysis. `https://gitlab.com/mopsa/mopsa-analyzer`, 2021. Accessed : 2021-04.

[16] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Benchmarks used by MOPSA. `https://gitlab.com/mopsa/benchmarks`, 2021. Accessed : 2021-09.

[17] G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium (SAS)*, pp. 221–239, 2006.

[18] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, pp. 71–190, 2015.

[19] R. Boutonnet and N. Halbwachs. Disjunctive relational abstract interpretation for interprocedural program analysis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 136–159, 2019.

[20] M. Chevalier and J. Feret. Sharing ghost variables in a collection of abstract domains. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 158–179, 2020.

[21] M. Christakis and C. Bird. What developers want and need from program analysis : an empirical study. In *International Conference on Automated Software Engineering*, 2016.

[22] I. Collet. Appliquer une pédagogie de l'égalité dans les enseignements d'informatique. `https://interstices.info/appliquer-une-pedagogie-de-legalite-dans-les-enseignements-dinformatique/`, 2022. Accessed : 2022-03.

[23] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pp. 84–96, 1978.

[24] P. Cousot, R. Cousot, J. Feret, A. Miné, X. Rival, B. Blanchet, D. Monniaux, and L. Mauborgne. The astrée static analyzer, 2004. URL `https://www.astree.ens.fr/`. Accessed : 2021-12.

[25] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Annual Asian Computing Science Conference (ASIAN)*, pp. 272–300, 2006.

[26] P. Cousot, R. Giacobazzi, and F. Ranzato. Program analysis is harder than verification : A computability perspective. In *International Conference in Computer-Aided Verification*, pp. 75–95, 2018.

[27] A. Cox, B. E. Chang, and X. Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis Symposium (SAS)*, pp. 134–150, 2014.

[28] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. In *International Conference on Software Engineering and Formal Methods*, pp. 233–247, 2012.

[29] A. Delaët, D. Merigoux, and A. Fromherz. Turning catala into a proof platform for the law. In *Programming Languages and the Law (POPL workshop)*, 2022.

[30] D. Delmas and J. Souyris. Astrée : From research to industry. In *Static Analysis Symposium (SAS)*, pp. 437–451, 2007.

[31] T. I. development team. Infer, a static analysis tool for Java, C++, Objective-C, and C., 2021. URL `https://github.com/facebook/infer`. Accessed : 2021-07.

[32] Direction Générale des Finances Publiques (DGFiP). Les règles du moteur de calcul de l'impôt sur le revenu, 2019. URL `https://gitlab.adullact.net/dgfip/ir-calcul`.

[33] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, pp. 62–70, 2019.

[34] ENS. Position des Écoles normales supérieures sur la nouvelle filière mpi. `http://www.ens-lyon.fr/actualite/formation/position-des-ecoles-normales-superieures-sur-la-nouvelle-filiere-mpi`. Accessed : 2022-03.

[35] ENS de Lyon. Agrégé préparateur en informatique. `http://www.ens-lyon.fr/sites/default/files/2021-10/0302%20AGPR%20INFORMATIQUE%20Profil%20%202022%20Session%201.pdf`. Accessed : 2022-03.

[36] A. Fromherz, A. Ouadjaout, and A. Miné. Static value analysis of python programs by abstract interpretation. In *Nasa Formal Methods (NFM)*, pp. 185–202, 2018.

[37] M. Furr and J. S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, pp. 18 :1–18 :63, 2008.

[38] GitHub. State of the github octoverse. `https://octoverse.github.com/#top-languages-over-the-years`, 2021. Accessed : 2021-09.

[39] E. Goubault, S. Putot, and F. Védrine. Modular static analysis with zonotopes. In *Static Analysis Symposium (SAS)*, pp. 24–40, 2012.

[40] J. Henry. *Static Analysis by Abstract Interpretation and Decision Procedures. (Analyse statique de programme par interprétation abstraite et procédures de décision)*. PhD thesis, University of Grenoble, France, 2014.

[41] J. Henry, D. Monniaux, and M. Moy. PAGAI : A path sensitive static analyser. *Electron. Notes Theor. Comput. Sci.*, pp. 15–25, 2012.

[42] H. Illous, M. Lemerre, and X. Rival. Interprocedural shape analysis using separation logic-based transformer summaries. In *Static Analysis Symposium (SAS)*, 2020.

[43] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Static Analysis Symposium (SAS)*, pp. 238–255, 2009.

[44] J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Principles of Programming Language*s, 2015.

[45] Jury du concours de l'agrégation en informatique. Sujet zéro de l'épreuve de tp pour l'agrégation en informatique. `https://agreg-info.org/files/2022/03/TP_0.zip`. Accessed : 2022-03.

[46] S. Lee, H. Lee, and S. Ryu. Broadening horizons of multilingual static analysis : Semantic summary extraction from C code for JNI program analysis. In *Automated Software Engineering (ASE)*, pp. 127–137, 2020.

[47] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett. Just fuzz it : solving floating-point constraints using coverage-guided fuzzing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, 2019.

[48] B. Loring and J. Kinder. Systematic generation of conformance tests for javascript. *CoRR*, abs/2108.07075, 2021.

[49] D. Merigoux. Catala contributors. `https://catala-lang.org/en/about`, 2022. Accessed : 2022-02.

[50] D. Merigoux, N. Chataing, and J. Protzenko. Catala : a programming language for the law. *Proc. ACM Program. Lang.*, (International Conference on Functional Programming), 2021.

[51] A. Miné. The octagon abstract domain. *High. Order Symb. Comput.*, pp. 31–100, 2006.

[52] Ministère de l'enseignement supérieur, de la recherche et de l'innovation. Programme des classes préparatoires mp2i/mpi. `https://prepas.org/index.php?document=73`. Accessed : 2022-03.

[53] D. Monniaux and L. Gonnord. Using bounded model checking to focus fixpoint iterations. In *Static Analysis Symposium (SAS)*, pp. 369–385, 2011.

[54] A. Ouadjaout and A. Miné. A library modeling language for the static analysis of C programs. In *Static Analysis Symposium (SAS)*, pp. 223–247, 2020.

[55] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python : The full monty. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pp. 217–232, 2013.

[56] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder : Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, pp. 598–608, 2015.

[57] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J. Seifert, and A. Feldmann. Static program analysis as a fuzzing aid. In *Research in Attacks, Intrusions and Defenses (RAID)*, pp. 26–47, 2017.

[58] G. Singh, M. Püschel, and M. T. Vechev. Making numerical program analysis fast. In *Programming Language Design and Implementation (PLDI)*, pp. 303–313, 2015.

[59] G. Singh, M. Püschel, and M. T. Vechev. Fast polyhedra abstract domain. In *Principles of Programming Languages*, pp. 46–59, 2017.

[60] P. Sotin and B. Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *European Symposium on Programming (ESOP)*, pp. 459–479, 2011.

[61] V. Stinner and the Python Benchmark Suite team. Performance benchmarks from Python's reference interpreter. `https://github.com/python/pyperformance/`. Accessed : 2021-08.

[62] G. Tan and G. Morrisett. Ilea : inter-language analysis across Java and C. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pp. 39–56, 2007.

[63] The Coq Development Team. The coq proof assistant, October 2017. URL `http://coq.inria.fr`.

[64] Typeshed contributors. Typeshed. `https://github.com/python/typeshed/`, 2021. Accessed : 2021-04.

[65] R. Upadrasta and A. Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Principles of Programming Languages*, pp. 483–496, 2013.

[66] G. van Rossum, J. Lehtosalo, and Łukasz Langa. Python Enhancement Proposal 484. https://www.python.org/dev/peps/pep-0484/, 2021. Accessed : 2021-08.

[67] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang. JN-SAF : precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code. In *Computer and Communications Security (CCS)*, pp. 1137–1150, 2018.

[68] M. Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pp. 439–449, 1981.

[69] V. Wüstholz and M. Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering (ICSE)*, pp. 789–800, 2020.

[70] L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees : representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, (Principles of Programming Languages), 2020.

[71] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, (International Conference on Functional Programming), 2021.

[72] Avec Mlang, Inria participe à la modernisation du calcul de l'impôt sur le revenu. https://www.inria.fr/fr/mlang-modernisation-calcul-impot-revenu.

[73] Facebook's 2021 testing and verification symposium. https://fbresearchevents.bevylabs.com/events/details/facebook-research-facebook-research-events-presents-2021-testing-and-verification-s

[74] Facebook PathPicker. https://github.com/facebook/PathPicker. Accessed : 2021-12.

[75] Un hackathon pour l'ouverture du code source du calculateur de l'impôt sur le revenu. https://www.economie.gouv.fr/hackathon-calculateur-impots.

[76] Loi république numérique du 7 octobre 2016. https://www.vie-publique.fr/eclairage/20301-loi-republique-numerique-7-octobre-2016-loi-lemaire-quels-changements. Accessed : 2021-11.

[77] SOAP 2020 Best Presentation Award. https://twitter.com/soap_workshop/status/1272625905572163585. Accessed : 2022-01.

[78] ACM. Artifact review and badging. https://www.acm.org/publications/policies/artifact-review-and-badging-current. Accessed : 2022-03.