Projet de recherche : Preuves de programmes en contexte

ARMAËL GUÉNEAU

Mon objectif est de faciliter l'adoption de code formellement vérifié au sein de systèmes logiciels réalistes, en utilisant les outils formels issus de la recherche en langages de programmation et preuves mécanisées. La recherche en preuves formelles de ces 10 dernières années a vu le succès de projets ambitieux tels CompCert, CakeML ou seL4, qui établissent de bout en bout la correction de logiciels complets, tels que des compilateurs ou des systèmes d'exploitation. Vérifier formellement un logiciel de cette ampleur requiert des efforts considérables, de plusieurs dizaines d'années-hommes. Pourtant, ces efforts ne permettent finalement de vérifier que quelques dizaines de milliers de lignes de code, là où l'implémentation de logiciels modernes se compte en dizaines voire centaines de millions de lignes.

Par exemple, le logiciel embarqué dans une voiture moderne atteint les 100 millions de lignes de code : il est difficile d'envisager tout vérifier. Heureusement, seule une fraction du logiciel en question est effectivement critique à la sécurité des passagers du véhicule. On espère donc pouvoir intégrer dans une même voiture des *composants* logiciels critiques (vérifiés, sécurisés) avec des composants qui n'en sont pas et que l'on ne prend pas la peine de vérifier. Il est alors crucial de disposer de garanties très fortes concernant l'interface entre ces composants et leurs interactions possibles — il serait par exemple peu souhaitable que le système de freins puisse être piraté par l'intermédiaire du Wi-Fi [1].

Dans le contexte de systèmes logiciels modernes, souvent bâtis comme un assemblage de composants variés formant un ensemble trop gros ou difficile à spécifier, il est donc particulièrement pertinent de chercher à vérifier le bon fonctionnement et la sûreté de composants, *lorsque placés dans le contexte du système entier*. Pourtant, il existe relativement peu de travaux de recherche répondant à la question : « **Comment vérifier formellement la correction de composants logiciels utilisés dans un contexte non vérifié ?** ».

La solution se trouve probablement dans la combinaison de différentes méthodologies : spécification des interfaces, méthodes de vérification et d'analyse statique, tests dynamiques, vérifications au niveau du matériel. Je propose d'attaquer la question selon trois axes principaux :

(1) L'étude de principes de raisonnement formels pour la conception de composants systèmes sûrs, exécutés sur un certain type de machines fournissant une sécurité accrue, les machines à capabilités (*capability machines*). Je compte m'intéresser à l'utilisation de la logique de séparation et des relations logiques pour capturer les propriétés de composants utilisant les capabilités, et également les appliquer pour raisonner sur l'interaction avec des langages de plus haut niveau.

(2) Un travail théorique et pratique pour la vérification de composants (bibliothèques) dans le cadre d'un langage haut niveau statiquement typé, en particulier pour des composants multi-langages exposant une interface haut niveau pour une implémentation bas niveau. J'espère obtenir une méthode de preuve permettant d'établir, pour un module d'un langage haut niveau dont l'implémentation contient du code bas niveau, que celui-ci se "comporte bien" : il peut être utilisé dans n'importe quel contexte bien typé, sans compromettre les bonnes propriétés du langage de haut niveau.

(3) L'élaboration d'une méthodologie pour vérifier la complexité asymptotique de bibliothèques en même temps que leur correction fonctionnelle : dans le cas d'un algorithme ou d'une structure de données, sa complexité est partie intégrante de sa spécification. J'aimerais intégrer l'approche que j'ai développée dans

1. https://www.mylegalneeds.com/blog/the-realities-of-car-hacking-through-wifi-and-bluetooth.cfm

ma thèse avec des outils existants d'inférence automatique de complexité, pour la rendre plus accessible. J'espère également la généraliser pour pouvoir l'appliquer à la preuve de bornes de complexité en espace, en plus de bornes de complexité en temps.

1 PREUVES DE SÉCURITÉ ET MACHINES À CAPABILITÉS

Une machine à capabilités est un type de microprocesseur étendant le fonctionnement d'une architecture conventionnelle avec des mécanismes de protection mémoire et de séparation des privilèges, garantis directement au niveau du matériel ("*security by design*"). Ces machines introduisent la notion de capabilités : il s'agit de mots machines munis de méta-données, que l'on ne peut créer ou modifier que selon des règles strictes, et qui permettent de fournir un accès restreint à la mémoire, jouant le rôle d'un pointeur de code ou de données. Leur utilisation permet d'éliminer à la source une large classe de failles de sécurité, dont notamment celles causées par les célèbres *buffer overflows*.

Ces machines sont aujourd'hui un candidat crédible pour devenir l'architecture de demain, envahissant nos téléphones et ordinateurs, mais n'ont reçu pour l'instant qu'une attention limitée de la part de la communauté de recherche en langages de programmation.

L'utilisation d'une machine à capabilités permet d'envisager de nouvelles manières de concevoir des systèmes plus sûrs, permettant un cloisonnement à grain plus fin, ou encore avec une base de confiance plus réduite. En particulier, il est alors envisageable de garantir la sûreté d'un composant sans nécessiter la coopération du système d'exploitation, et ce même lorsque ce composant peut interagir avec du code totalement inconnu.

Toutefois, les capabilités ne fournissent une sûreté accrue qu'à condition de bien les utiliser ! Concevoir des scénarios complexes d'utilisation des capabilités ou proposer l'introduction de nouvelles capabilités n'est pas une tâche aisée : les capabilités opèrent à bas niveau, et la sécurité qu'elles fournissent dépend de leur interaction avec l'ensemble des fonctionnalités de la machine. Il est donc important de pouvoir raisonner formellement sur les garanties fournies par la machine, dans des scénarios concrets ; sans cadre formel, il est aisé de commettre des erreurs potentiellement critiques. De plus, il est très difficile de corriger un bug dans un CPU une fois que celui-ci a été fabriqué : on a donc envie de s'assurer en amont que telle ou telle utilisation des capabilités fournit bien les garanties que l'on espère.

Durant mon post-doc, j'ai pris part à une ligne de recherche s'intéressant à l'utilisation d'outils formels (logique de séparation et relations logiques) pour raisonner sur un mode d'utilisation subtil de capabilités implémentant une convention d'appel sûre ; le tout étant mécanisé dans l'assistant de preuve Coq et le framework Iris.

1.1 Vérification de composants système

J'espère étendre et redéployer la méthodologie utilisée pour raisonner sur la convention d'appel (qui fournit une forme de communication sûre entre composants ne se faisant pas confiance mutuellement), pour établir la sûreté d'autres composants système qui seraient utiles pour bâtir une pile logicielle vérifiée utilisant les capabilités.

C'est un projet qui, pour être mené à bien, nécessite d'une part une bonne compréhension des enjeux d'un point de vue système, afin de garantir la pertinence du code et des propriétés vérifiées ; d'autre part une maîtrise de techniques formelles avancées (relations logiques et la logique de programmes Iris) ; et finalement des compétences pratiques de preuve de programmes pour pouvoir mener à bout la vérification de programmes bas niveau complexes.

Pendant mon post-doc j'ai été en contact avec le groupe de Peter Sewell (Université de Cambridge) qui contribue aux approches formelles au sein du projet CHERI ; les propositions ci-après pourraient conduire à une collaboration intéressante.

Allocateur mémoire. Un rapport du Microsoft Security Response Center sur CHERI [JEA20] pointe du doigt le fait que l'allocateur mémoire est un composant particulièrement critique d'un système à capabilités.

Le groupe CHERI s'intéresse à la conception d'allocateurs mémoire garantissant une forme de "sûreté temporelle" éliminant les attaques du type *use after free* (CHERIvoke [XWA+19] puis Cornucopia [FBJ+20]). Lorsqu'un client appelle `free(c)` (signalant qu'il libère une région mémoire allouée précédemment) on cherche alors à garantir que celui-ci perd accès à toutes les occurences de la capabilité `c`. Ainsi, on empêche un programme bogué ou malicieux de continuer à utiliser `c` après l'appel à `free`, alors que la portion de mémoire correspondante a été réutilisée par l'allocateur et allouée à quelqu'un d'autre.

Les articles mentionnés ci-dessus se concentrent sur une évaluation pratique des idées proposées. D'un point de vue formel, il reste alors premièrement à énoncer la propriété de sécurité que l'on attend d'un tel allocateur mémoire, puis effectivement vérifier cette propriété vis-à-vis d'une implémentation concrète.

J'espère pouvoir m'inspirer des outils formels développés lors du travail au sujet de la convention d'appel sûre. Il y a certains points communs avec le fonctionnement d'un allocateur mémoire sûr. Dans les deux cas, l'objectif est de pouvoir *révoquer* l'accès à certaines capabilités à un certain point de l'exécution en les effaçant de la mémoire. Il reste toutefois un certain nombre de difficultés spécifiques. D'une part, il reste à énoncer la spécification haut niveau de ce que serait un allocateur mémoire sûr. Ceci est d'autant plus compliqué que les implémentations proposées par le groupe CHERI implémentent une notion relachée de sûreté temporelle (pour des raisons d'efficacité) : il est possible d'utiliser une capabilité `c` peu après l'appel à `free(c)`, mais dans ce cas son utilisation est en fait sûre, car la mémoire correspondante n'a pas encore été réutilisée. D'autre part, les techniques de preuve devront être adaptées, car les mécanismes d'effacement de la mémoire sont différents : dans la convention d'appel, chaque participant se charge de nettoyer le fragment de pile dont il est responsable ; dans le cas de l'allocateur, une routine système séparée se charge du nettoyage de la mémoire entière. Finalement, d'un point de vue d'un client vérifié de l'allocateur, on veut pouvoir raisonner comme s'il avait la possession unique d'une région allouée tant qu'il ne l'a pas libérée — ce qui semble en première approche contredire le fait que la routine de nettoyage possède une capabilité lui donnant accès à l'ensemble de la mémoire.

Scheduler et gestion des interruptions. L'ordonnanceur (*scheduler*) est un composant essentiel d'un système, permettant de tirer parti de la nature asynchrone des communications avec le monde extérieur, avoir un système de tâches concurrentes, ou simplement limiter le temps alloué à un programme (indispensable dans un système réaliste). Un ordonnanceur est typiquement implémenté en ayant recours au mécanisme d'*interruptions* matérielles : lorsqu'une interruption est reçue (déclenchée par une horloge ou un périphérique externe), une routine de gestion des interruptions est exécutée, donnant l'opportunité au système de gérer l'interruption ou passer le contrôle à un autre programme.

Les implications en terme de sécurité sont non-triviales : lorsqu'un programme est interrompu et que la routine de gestion des interruptions est exécutée, celle-ci gagne accès au contenu des registres du programme qui s'exécutait précédemment (et donc gagne accès à ses capabilités). On veut alors par exemple s'assurer que les valeurs contenues dans les registres soient sauvegardées et restaurées correctement.

Une piste de recherche prometteuse, identifiée avec mes collègues à l'université d'Aarhus, KU Leuven et Vrije Universiteit Brussel, serait l'application de nos techniques formelles pour comprendre comment structurer un ordonnanceur sur une machine à capabilités, tout en respectant un principe de moindre autorité. On espère pouvoir premièrement isoler et vérifier une routine de gestion des interruptions minimale, dont le seul rôle est la gestion des registres. Cette routine donnerait ensuite le contrôle à l'implémentation de l'ordonnanceur, pour laquelle on peut vérifier les propriétés désirées (par exemple, vivacité du système). Il devrait être possible de ne pas lui faire confiance : tant que la routine minimale sûre est présente, on veut pouvoir raisonner sur du code utilisateur comme s'il s'agissait d'un programme concurrent, quelle que soit l'implémentation concrète de l'ordonnanceur.

Il pourrait également être intéressant de s'inspirer des idées du projet Bossa mené par Julia Lawall et Gilles Muller [LGC+20, BFLM06, MDL05], qui s'intéressent au développement et à la vérification d'ordonnanceurs réalistes, tout en permettant de s'abstraire des détails d'implémentation du système d'exploitation.

1.2 Compilation vers machines à capabilités

Dans mon travail de post-doc, je me suis principalement consacré à raisonner à propos de code machine, autant pour le code inconnu que pour l'implémentation vérifiée d'un composant. Il est légitime de vouloir profiter de la sûreté fournie par la machine tout en écrivant le code dans un langage de plus haut niveau. Par exemple, le groupe CHERI travaille à une extension de C et de la chaîne de compilation correspondante (CHERI-C [WRD+20]) qui utilise des capabilités pour augmenter la sûreté des programmes C en compilant certaines abstractions du langage source (buffers, structures, etc) vers des capabilités.

Le langage CakeML (et son compilateur vérifié) serait un autre candidat potentiel. Il est possible pour un programme CakeML d'appeler des routines externes via un mécanisme de FFI ("*foreign function interface*") fourni par le langage, mais l'utilisateur doit alors faire confiance à l'implémentation de ces routines externes pour ne pas modifier la mémoire du programme CakeML, seulement écrire et lire dans un buffer dédié, et ne pas garder de référence vers ce buffer (qui pourra être déplacé par le *garbage collector*).

Je pense qu'il est possible de tirer parti d'un schéma de compilation utilisant les capabilités pour permettre à des composants CakeML d'être *robustes* vis-à-vis des routines externes invoquées via la FFI : alors, même si celles-ci comportent des failles de sécurité catastrophiques, elles ne pourront pas compromettre le fonctionnement du programme CakeML.

Conceptuellement, il suffirait d'implémenter dans le compilateur CakeML une convention d'appel sûre basée sur les capabilités (comme par exemple celle proposée dans mon article POPL'21). À moyen terme, c'est ce que j'espère pouvoir faire ; et la tâche n'est déjà pas triviale : il faut ajouter un modèle de machine à capabilités à l'écosystème CakeML, et déployer une convention d'appel sûre expérimentale dans le cadre d'un compilateur réaliste. À plus long terme, on pourra chercher à munir CakeML d'un théorème de *secure compilation*, en adaptant l'argument développé dans le cadre de l'article POPL'21 pour raisonner sur les propriétés de sécurités fournies par la convention d'appel après compilation.

2 BONNE INTEROPÉRABILITÉ VIA LE TYPAGE ET COMPOSANTS MULTI-LANGAGES

Les bibliothèques d'un langage de haut niveau sont une autre forme de composants logiciels réutilisables. Ceux-ci sont composés au sein d'un même langage, qui fournit généralement des mécanismes d'encapsulation bénéfiques à la programmation modulaire (et qui peuvent être statiques ou dynamiques). Comment s'appuyer sur ces mécanismes d'encapsulation pour vérifier que les garanties établies pour une bibliothèque donnée seront préservées lors de son utilisation dans un contexte non vérifié ? En particulier, je m'intéresse à relier des spécifications établies en logique de séparation (qui raisonnent donc en terme de ressources et de possession unique) avec des interfaces exprimées dans un système de type à la ML (qui ne permet pas de raisonner sur des ressources ou des question de possession).

Il s'agit alors de concevoir l'interface de la bibliothèque de façon à pouvoir montrer que les préconditions attendues par le composant vérifié sont garanties par les mécanismes d'encapsulation du langage.

2.1 Interaction entre composant vérifié et contexte non vérifié mais bien typé

Dans le cas d'un langage statiquement typé, si l'on vérifie formellement une bibliothèque, puis qu'on l'expose à un contexte bien typé mais non vérifié, le danger est que ce contexte utilise la bibliothèque dans des cas non couverts par la spécification. On a alors "mal" utilisé le composant, et la preuve formelle de sa correction ne s'applique plus ! Par exemple, considérons le cas de la vérification de la correction d'une bibliothèque implémentée

en OCaml en utilisant la logique de séparation (comme le permet par exemple l'outil CFML [Cha19]). La logique de séparation permet d'exprimer la possession unique d'une structure de données, notion qui n'est pas capturée par le système de types d'OCaml. Rien n'empêche alors un client non vérifié de modifier une table de hachage pendant que celui-ci la traverse, alors que ceci est implicitement interdit par la spécification, corrompant silencieusement la structure de données.

J'espère montrer que, pour un système de types fournissant une notion d'encapsulation suffisante (comme le système de modules d'OCaml fournissant des types abstraits), alors il est souvent possible de garantir que des spécifications en logique de séparation sont préservées par un contexte typé, soit statiquement, soit en signalant explicitement via une erreur à l'exécution quand ce n'est pas le cas.

J'anticipe trois composants pour cette approche.

Premièrement, j'espère définir une interprétation sémantique d'un sous-ensemble du système de modules de ML, afin de capturer les propriétés d'abstraction et d'encapsulation correspondantes. L'utilisation d'Iris est particulièrement indiquée ici (définir une notion de "typage sémantique" pour un λ-calcul avec typage polymorphe, références et types existentiels est une application classique d'Iris [BB20, §17]). Considérer un système de modules entier est significativement plus compliqué (voir les travaux de Rossberg et collaborateurs [RRD14] et plus récemment Crary [Cra17]) ; je compte me restreindre au sous-ensemble raisonnable (signatures avec types abstraits, foncteurs de premier ordre) le plus utilisé en pratique.

Deuxièmement, je prévois de montrer sur des exemples de bibliothèques concrètes que la combinaison de types abstraits et tests dynamiques légers permet de garantir des spécifications riches en logique de séparation (par exemple, le transfert de possession effectué lors de l'utilisation d'un itérateur).

Finalement, il restera à concevoir une méthode de preuve pour pouvoir vérifier formellement que l'implémentation vérifiée d'une bibliothèque et ses tests dynamiques peut être exposée à un type donné. On cherchera à montrer que pour un contexte arbitraire bien typé, si l'exécution de ce contexte associé à la bibliothèque ne conduit pas à l'échec d'un test dynamique, alors ce contexte utilise en fait la bibliothèque correctement selon sa spécification formelle.

2.2 Composants multi-langages : interface haut niveau et implémentation bas niveau

En pratique, un nombre non négligeable de bibliothèques de langages de haut niveau sont partiellement implémentées avec du code bas niveau (typiquement, écrit en C), pour interopérer avec des bibliothèques ou des interfaces systèmes, ou pour des raisons d'efficacité. Les détails de la FFI (qui définit comment lier du code de haut niveau avec du code plus bas niveau) varient d'un langage à un autre. Dans le cas d'une FFI haute performance, comme celle offerte par OCaml, le programmeur a un contrôle fin lui permettant d'implémenter un code de FFI le plus efficace possible ; mais en retour, il doit obéir de nombreuses restrictions lors de son interaction avec le langage de haut niveau.

Malheureusement, après 25 ans d'existence, OCaml n'est toujours pas muni d'une spécification ou d'un modèle formel de l'interface FFI exposée au programmeur. La programmation de code d'interfaçage entre OCaml et C (*stubs*) reste un art réservé aux initiés, et le débogage est difficile. Le travail de Furr et Foster [FF08] est un premier pas dans la bonne direction, mais se concentre sur un outil d'analyse pragmatique des *stubs* existants, et ne modèle (en isolation) qu'un sous-ensemble de la FFI exposée par OCaml.

J'aimerais pouvoir décrire formellement l'ensemble des mécanismes nécessaires pour vérifier de bout en bout un composant multi-langage. Comment vérifier formellement qu'un composant, implémenté dans un langage bas niveau et exposé selon une interface haut niveau bien typé : 1) est correct, 2) satisfait son interface, et 3) que cette interface ne permet pas de briser les bonnes propriétés du langage environnant ?

Les motivations sont ici similaires à celles du projet RustBelt [JJKD18], qui s'attache à vérifier la sûreté "en contexte" de bibliothèques Rust utilisant le fragment "unsafe" du langage dans leur implémentation. Cependant,

on a ici un certain nombre de difficultés supplémentaires, provenant du fait qu'on s'intéresse à un composant mélangeant deux langages différents, plutôt qu'utilisant un fragment non sûr d'un même langage.

Les deux langages d'implémentation sont liés au niveau machine, après compilation. À moyen terme, on espère avoir un résultat de compilation vérifiée pour le composant multi-langage, qui nécessite alors de relier les énoncés de compilation vérifiée des deux compilateurs impliqués (le compilateur pour le langage de haut niveau, et celui pour le langage de bas niveau). Différentes notions de compilation correcte compositionnelle ont été étudiées dans la littérature [PA19]. J'espère pouvoir énoncer une notion plus faible et demandant moins d'efforts pour être intégrée à un compilateur vérifié : en effet, il suffit ici que les deux énoncés de compilation correcte puissent être composés pour un composant donné, connu, et dont on vérifie la correction.

La plupart des langages de haut niveau proposent une gestion automatique de la mémoire, via un GC (*garbage collector*). Pour de tels langages, il devient nécessaire en vue de notre objectif de spécifier formellement l'interface offerte par le GC au code bas niveau. Par exemple, si le GC peut déplacer en mémoire les valeurs du langage de haut niveau, partager un pointeur vers une valeur n'est possible que sous certaines conditions. Autre exemple, si un fragment de mémoire géré par le code bas niveau détient une valeur du langage haut niveau, il est nécessaire d'en informer explicitement le GC afin que celle-ci ne soit pas considérée comme "morte" et désallouée. Je compte donc, dans le cadre d'une logique de séparation pour C : 1) donner un cadre pour spécifier l'interface bas niveau offerte par le GC, en s'intéressant aux différentes fonctionnalités offertes par différents langages (OCaml, .NET, Go, Lua, ...) ; 2) utiliser ce cadre pour énoncer une spécification formelle détaillée dans le cas d'OCaml ; 3) utiliser cette spécification pour vérifier la correction d'exemples concrets de code C utilisant la FFI OCaml. À noter qu'il est également possible d'encoder les règles pour interagir avec le GC dans un système de types comme celui de Rust [oca] ; mais dans un cadre formel, il me semble plus simple de s'en tenir à l'utilisation une logique de séparation, en première approche.

Finalement, il est important de pouvoir prouver d'un module contenant du code bas niveau qu'il préserve les bonnes propriétés du langage haut niveau. Il ne suffit pas d'établir que celui-ci est sûr vis-à-vis du GC et que l'on peut lui donner une interface bien typée. Par exemple, il serait possible d'exposer des fonctions permettant la représentation mémoire de valeurs, mais peu désirable car brisant des propriétés d'abstraction ou de paramétricité. Je compte m'inspirer de l'approche consistant à prouver la propriété de *pleine adéquation* (full abstraction), étudiée en particulier par Amal Ahmed dans le contexte des programmes multi-langages [Ahm15, SNRA18]. On chercherait alors à prouver, pour un langage de haut niveau L et un composant donné C, que le langage $L + C$ peut se plonger de manière pleinement abstraite vers L. On obtiendrait alors que n'importe quel utilisateur du module C ne peut le distinguer d'un module implémenté uniquement dans le langage L.

Il s'agit d'un programme ambitieux dont nous avons exposé ci-dessus plusieurs axes. En première approche, il semble pertinent de chercher à établir un résultat de correction de bout en bout dans un scénario simplifié, en considérant des langages jouets (mini-ML avec un GC trivial et mini-C par exemple) ; et seulement après chercher à appliquer certains des éléments de l'approche au cadre de langages plus réalistes.

3 VÉRIFICATION DE CORRECTION ET COMPLEXITÉ ASYMPTOTIQUE

Une certaine catégorie de bibliothèques hautement réutilisables sont celles implémentant des structures de données et algorithmes, et dont l'implémentation est généralement subtile. La spécification mathématique de telles bibliothèques inclut alors non seulement leur correction mais également leur garantie de complexité : la raison d'être d'un algorithme subtil est généralement sa complexité, par opposition à une solution naïve. Un bug de complexité dans l'implémentation d'une bibliothèque peut de plus exposer son utilisateur à des attaques de type "déni de service", où un attaquant épuise les ressources de la machine (temps ou mémoire).

J'ai développé dans ma thèse une méthode de preuve pour la correction et la complexité, utilisant la notion de *crédits temps* en logique de séparation. Ceux-ci permettent de tirer parti des invariants de correction fonctionnelle

pour les arguments de complexité, et permettent également d'établir des bornes de complexité amortie. Toutefois, cette approche (embarquée dans Coq comme une extension de CFML) est pour l'instant relativement difficile d'utilisation et reste réservée aux experts.

3.1 Intégration avec des outils automatiques d'inférence de complexité

Je propose de travailler à l'intégration de l'approche développée pendant ma thèse – difficile d'emploi mais très expressive – avec des outils existants *d'inférence* de complexité, qui sont complètement automatiques mais plus limités.

La majorité des travaux existants se concentrent en effet essentiellement soit sur des approches automatisées (limitées dans l'ensemble des programmes et bornes gérés) [VH04, FMH14, CHRS17, HDW17, MKK17, ADL17], ou des approches embarquées dans un assistant de preuve et requérant une vérification manuelle [vM08, Dan08, MFN+16, ZH18, MJP19, HL19].

J'aimerais pouvoir utiliser un des outils automatisés comme "procédure auxiliaire" (de décision et d'inférence) depuis un assistant de preuve pour les morceaux de programme rentrant dans le cadre de l'outil, et en utilisant une approche plus manuelle pour les bornes ou les programmes plus complexes.

Le travail de Quentin Carbonneaux *et al.* [CHS15, CHRS17, Car18] se rapproche de cette idée : Quentin a développé un outil d'inférence automatique de complexité produisant des certificats Coq. Ces certificats s'appliquent à une représentation intermédiaire pour des programmes issus d'un code objet type "bytecode LLVM", et concernent donc plutôt l'analyse de programmes bas niveau manipulant des entiers (plutôt que des structures de données inductives).

Le travail de Jan Hoffmann sur RAML [HAH12, HDW17] est également prometteur, puisqu'effectuant une inférence de complexité sur un sous-ensemble fonctionnel d'OCaml avec types de données inductifs. L'analyse ne produit pas de certificats Coq, mais construit une dérivation de typage, dans le système de types utilisé par RAML pour son analyse de complexité.

Un premier objectif serait alors de traduire un certificat issu de l'outil de Quentin Carbonneaux ou une dérivation de type issue de RAML en une dérivation de preuve en logique de séparation avec crédits temps, utilisable en Coq. J'ai été en contact avec Quentin pendant ma thèse, ce qui pourrait mener à une collaboration.

Un défi supplémentaire : pourrait-on utiliser un outil d'inférence automatique pour un programme appelant une fonction auxiliaire dont la complexité a été prouvée manuellement (en logique de séparation) ? Ce n'est pas évident : les outils d'inférence comme RAML travaillent en interne avec des spécifications de complexité "modulo résolution de contraintes" (résolution qui est faite en toute dernière étape) ; à l'inverse, pour des spécifications établies à la main, on préfère des énoncés plus abstraits, idéalement en utilisant la notation asymptotique O. Il y aurait donc un travail théorique supplémentaire à effectuer pour étendre les outils d'inférence afin de leur permettre de tirer parti de telles spécifications.

3.2 Vérification d'algorithmes concrets

J'espère continuer à appliquer les approches de preuves de complexité que je développe pour la vérification formelle de bibliothèques concrètes, d'une part pour des algorithmes de l'état de l'art, dans la lignée de l'algorithme de détection incrémentale de cycles développé pendant ma thèse ; et d'autre part pour des structures de données "de tous les jours" dans la lignée du projet Vocal [voc].

À plus long terme, on peut se demander quelle pourrait être la prochaine étape importante pour évaluer des outils de preuve de complexité améliorés. Dans la lignée des algorithmes incrémentaux (ou dynamiques) sur les graphes, on pourrait s'intéresser au problème de "test de planarité". Celui-ci a vu une avancée majeure récemment, avec la publication par Holm et Rotenberg [HR20] d'un nouvel algorithme dont la complexité amortie en $O(\log^3 n)$ améliore la borne précédente de $O(\sqrt{n})$ vieille de plus de 20 ans. Le degré de sophistication de cet

algorithme et sa preuve rendent peu réaliste de s'y attaquer avec les outils de vérification et résultats vérifiés d'aujourd'hui ; mais il me semble constituer néanmoins un "horizon" au moins conceptuellement atteignable avec les techniques (Coq et logique de séparation avec crédits temps) dont on dispose.

3.3 Complexité en espace

En plus de bornes de complexité en temps, il est également utile d'établir des bornes de complexité en espace. On pense en particulier au cas de structures de données avec partage (notamment structures de données fonctionnelles ou (semi-)persistantes), dont il est difficile d'analyser la consommation mémoire.

En collaboration avec François Pottier, Arthur Charguéraud et Jacques-Henri Jourdan, je compte m'intéresser à la généralisation des crédits temps en des *crédits mémoire*. La possession de crédits mémoire garantirait alors qu'un certain nombre de cases mémoires sont disponibles ; si l'on prouve une spécification en logique de séparation avec crédits mémoire, alors le nombre de crédits en précondition donnerait une borne sur la quantité maximale de mémoire requise lors de l'exécution.

L'idée elle-même se retrouve chez Martin Hofmann, qui décrit en 2000 [Hof00] un langage avec gestion manuelle de la mémoire où un "crédit" (un diamant) représente l'adresse d'un bloc mémoire libre en mémoire. Cependant, j'aimerais m'intéresser au cas d'un langage avec gestion automatique de la mémoire (via un GC), qui est plus subtil et reste à explorer. Dans ce cadre, il devient notamment bien plus difficile de raisonner sur le partage et le moment où les données sont libérées.

Techniquement, une difficulté est que l'on aimerait étendre la logique de séparation pour pouvoir raisonner sur la *vivacité* d'une structure en mémoire. Si on libère la mémoire correspondant à une structure (qui était précédemment utilisée), alors on espère obtenir des crédits mémoire en échange. Dans un langage avec gestion automatique de la mémoire, ceci n'est possible que si on est capable de justifier que le reste du programme ne maintient pas la structure vivante. Peut-on définir, au niveau de la logique de séparation, une notion permettant de raisonner de façon modulaire sur des "références uniques" à une structure en mémoire ?

4 ÉQUIPES D'ACCUEIL

4.1 IRIF : PPS

L'équipe PPS a des compétences riches en sciences du logiciel. Elle est le centre historique du développement de l'assistant de preuve Coq, outil qui supporte ma recherche et que je prévois d'utiliser dans chaque axe de mon projet : ce serait un lieu propice à la réalisation de mon projet de recherche.

Mon travail pourra également contribuer au développement de Coq lui même. Pendant ma thèse, j'ai collaboré avec Jacques-Henri Jourdan sur la vérification de la complexité d'un algorithme de graphe de l'état de l'art. Ce travail était initialement motivé par l'implémentation de Coq elle-même : une version plus avancée de ce même algorithme est utilisée dans le noyau de Coq, et sa performance y est critique. Il serait donc intéressant de poursuivre ce travail, qui entre dans l'axe §3.2 de mon projet, en collaboration avec Hugo Herbelin et également Matthieu Sozeau (anciennement PPS, maintenant à l'équipe Inria Gallinette), qui mène le projet MetaCoq.

Mon travail de thèse m'a conduit à m'interroger sur les façons possibles, dans le cadre de Coq, d'exposer un "environnement de preuve" spécialisé pour un domaine d'application particulier (en l'occurence, les preuves de complexité). Par exemple, ma bibliothèque `procrastination` détourne l'utilisation des *evars* de Coq de leur utilisation normale afin fournir une interface utilisateur différente, mais le résultat est loin d'être parfait. L'axe §3.1 de mon projet (preuves de complexité plus agréables) bénéficierait donc particulièrement de l'expertise de l'équipe de développement de Coq.

Ralf Treinen s'intéresse à raisonner formellement sur les dépendances et l'évolution de composants dans le cadre de dépots logiciels *open source*. Ces problématiques se placent directement dans le prolongement de l'axe §2.1 de mon projet. Je propose d'y formaliser l'interaction entre un composant et son contexte, en fonction du

type qu'il expose. Se pose alors la question de l'évolution d'un tel composant : comment utiliser un modèle des types d'OCaml pour exprimer qu'un changement de l'interface du composant est par exemple rétro-compatible, en terme de versions utilisables par un système de gestion de paquets ? Sur cette question, et plus généralement les axes §2.1 et §2.2, je pourrais également collaborer avec Hugo Férée, qui a étudié la modélisation d'un système de modules et signature riche comme celui d'OCaml dans le cadre du projet Rotor [RFTO19].

Je profiterais également grandement d'interactions avec l'équipe « Algorithmes et complexité » de l'IRIF, dans le cadre de l'axe §3 de mon projet. Notamment, ce serait une opportunité judicieuse pour comprendre comment adapter mes techniques de preuve formelle aux modèles de calcul qui y sont étudiés (par exemple, algorithmes probabilistes, *streaming*, ...). Le cas des algorithmes *streaming* est particulièrement intéressant, car il inclut une contrainte de complexité en mémoire (sous-linéaire), rejoignant ainsi l'axe §3.3 de mon projet.

Finalement, j'apporterais à l'équipe une expertise nouvelle dans les problématiques liées aux machines à capabilités ; ainsi que dans l'utilisation et le développement de logiques de programmes embarquées dans un assistant de preuve (grâce à mon expérience avec CFML, Iris, et CakeML).

4.2 IRISA : Celtique

L'équipe Celtique de l'IRISA a une expertise impressionnante à la fois en sécurité et en compilation vérifiée. Un certain nombre de travaux sont par ailleurs menés à bien dans le cadre ambitieux du compilateur vérifié CompCert, incluant par exemple une variante de CompCert préservant les propriétés *constant time* [BBG+19] (important pour la sécurité de code cryptographique), ou encore pour garantir des propriétés de cloisonnement (*software fault isolation*) [BBD+19] — révélant par ailleurs la forte expertise locale en l'assistant de preuve Coq. Il s'agirait donc d'un endroit idéal pour développer les axes de mon projet liés à la sécurité et aux machines à capabilités (§1). Notamment, l'axe §1.2 (compilation vers capabilités) pourrait profiter des compétences fortes de l'équipe sur le compilateur vérifié CompCert, et pourrait mener à des collaborations directement liées à l'utilisation de CompCert pour cibler des machines à capabilités.

Sandrine Blazy s'intéresse à raisonner formellement à propos de code machine [BLP16], et plus récemment à raisonner sur l'interaction entre code bas niveau et langage haut niveau dans le cadre d'un compilateur *just-in-time* vérifié [BBP20]. Ce travail soulève un certain nombre de problématiques partagées avec l'axe §2.2 de mon projet concernant la vérification de composants multi-langages : une collaboration sur le sujet ne pourrait être que bénéfique.

Alan Schmitt et Thomas Jensen sont parmi les concepteurs des "sémantiques squelettiques" [BGJS19], qui permettent de raisonner à propos de langages dont la sémantique, si écrite naïvement, serait beaucoup trop grosse pour être humainement utilisable. On retrouve exactement la même problématique à propos des modèles formels de machines à capabilités CHERI réalistes. Celles-ci comportent tellement de détails et de cas d'erreurs différents, qu'on ne sait actuellement pas les intégrer de façon raisonnable avec une logique de séparation comme Iris, d'une manière qui permettrait de prouver la correction de programmes même modestes. Il serait donc très intéressant d'explorer l'utilisation de sémantiques squelettiques, afin de passer à l'échelle et pouvoir raisonner formellement sur des machines à capabilités réalistes.

En plus de son expertise en sécurité et compilation vérifiée, Frédéric Besson est un expert de l'implémentation de procédures vérifiées dans Coq, grâce à son travail sur la bibliothèque Micromega. Son expertise serait très utile pour le développement de procédures vérifiées d'aide à la preuve de complexité, dans le cadre de l'axe §3.1 de mon projet de recherche.

J'apporterais à l'équipe mon expérience dans l'utilisation de logiques de programmes (notamment logique de séparation, grâce à mon expérience de thèse et de post-doc) pour la vérification de programmes, ainsi que la connaissance des problématiques spécifiques liées aux machines à capabilités — thèmes qui me semblent bien complémenter les compétences de l'équipe en sécurité et vérification.

4.3 LMF

Le LMF résulte de la fusion récente entre l'équipe VALS (équipe de l'ancien LRI) et du LSV (UMR de l'ENS Paris-Saclay). Les thématiques de recherche abordées au LMF sont proches de nombreux aspects abordés dans mon projet de recherche. Cela en ferait, je pense, un excellent cadre pour de futures collaborations fructueuses.

L'équipe possède une compétence très forte en vérification, et développe notamment la plateforme de vérification de programmes Why3, qui représente un compromis très intéressant entre expressivité et automatisation. Why3 serait un cadre d'expérimentation idéal pour développer la méthodologie de preuve de complexité plus automatisée et plus agréable à utiliser, qui fait l'objet de l'axe en §3.1 de mon projet. Ce travail bénéficierait donc grandement d'une collaboration avec les architectes de Why3, notamment Jean-Christophe Filliâtre, Andrei Paskevich, Guillaume Melquiond et Claude Marché.

Comme mentionné précédemment, j'ai collaboré pendant ma thèse avec Jacques-Henri Jourdan sur la vérification d'un algorithme de graphe, et un volet entier de ce travail reste encore à faire : l'équipe de développement Coq est intéressée par la vérification d'une version plus avancée de ce même algorithme, qui est actuellement utilisée dans l'implémentation du noyau de Coq. Continuer cette collaboration très prometteuse entrerait de plus parfaitement dans l'axe §3.2 de mon projet. Jacques-Henri Jourdan est également un expert du *runtime* du langage OCaml, grâce à son travail sur le *profiler* mémoire Statmemprof, ainsi qu'en logique de séparation, grâce à son travail au sein du projet RustBelt. Je pourrais grandement profiter de ces deux expertises combinées dans le cadre de mon projet de vérification de composants multi-langages (§2.2).

Par ailleurs, Jean-Christophe Filliâtre coordonne le projet ANR Vocal (auquel j'ai participé pendant ma thèse), qui construit une bibliothèque généraliste vérifiée pour OCaml, et soulève donc les questions d'interactions entre code vérifié / non-vérifié qui font l'objet de l'axe §2.1 de mon projet, en plus d'être une cible idéale pour la vérification formelle de la complexité de structures de données et algorithmes (§3.2).

Dans le cadre de l'axe §3.1 de mon projet, concernant l'intégration d'outils automatisés d'inférence de complexité avec des outils de vérification, je bénéficierais également de l'expertise du LMF. En particullier, Chantal Keller a travaillé à l'intégration avec Coq de certificats issus d'outils de preuve automatique, et le projet Dedukti mené par Gilles Dowek se consacre à développer un cadre général permettant de transporter des certificats de preuve entre outils formels.

Enfin, j'apporterais à l'équipe une nouvelle expertise en vérification de propriétés de sécurité et sur les questions liées aux machines à capabilités, acquise lors de mon post-doc.

RÉFÉRENCES

[ADL17] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP) :43 :1–43 :29, August 2017.

[Ahm15] Amal Ahmed. Verified compilers for a multi-language world. SNAPL, 2015.

[BB20] Lars Birkedal and Aleš Bizjak. Lectures notes on Iris : Higher-order concurrent separation logic, 2020. https://iris-project.org/tutorial-material.html.

[BBD+19] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. Compiling Sandboxes : Formally Verified Software Fault Isolation. In *ESOP 2019 - 28th European Symposium on Programming*, volume 11423 of *LNCS*, pages 499–524, Prague, Czech Republic, April 2019. Springer.

[BBG+19] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[BBP20] Aurèle Barrière, Sandrine Blazy, and David Pichardie. Towards formally verified just-in-time compilation. In *CoqPL*, 2020.

[BFLM06] Jean-Paul Bodeveix, Mamoun Filali, Julia L. Lawall, and Gilles Muller. Automatic Verification of Bossa Scheduler Properties. In Stephan Merz and Tobias Nipkow, editors, *Automatic Verification of Critical Systems*, pages 19–34, Nancy, France, September 2006.

[BGJS19] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal semantics and their interpretations. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[BLP16] Sandrine Blazy, Vincent Laporte, and David Pichardie. Verified abstract interpretation techniques for disassembling low-level self-modifying code. *J. Autom. Reason.*, 56(3) :283–308, March 2016.

[Car18] Quentin Carbonneaux. *Modular and Certified Resource-Bound Analyses*. PhD thesis, 2018.

[Cha19] Arthur Charguéraud. The CFML tool and library. http://www.chargueraud.org/softs/cfml/, 2019.

[CHRS17] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated resource analysis with Coq proof objects. volume 10427, pages 64–85, 2017.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 467–478, New York, NY, USA, 2015. ACM.

[Cra17] Karl Crary. Modules, abstraction, and parametric polymorphism. POPL, 2017.

[Dan08] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. 2008.

[FBJ+20] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia : Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[FF08] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.*, 30(4), August 2008.

[FMH14] Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 275–295, Cham, 2014. Springer International Publishing.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. volume 7358, pages 781–786, 2012.

[HDW17] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. pages 359–373, 2017.

[HL19] Maximilian P. L. Haslbeck and Peter Lammich. Refinement with Time - Refining the Run-Time of Algorithms in Isabelle/HOL. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20 :1–20 :18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Hof00] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4) :258–289, December 2000.

[HR20] Jacob Holm and Eva Rotenberg. Fully-dynamic planarity testing in polylogarithmic time. STOC, 2020.

[JEA20] Nicolas Joly, Saif ElSherei, and Saar Amar. Security analysis of CHERI ISA. Technical report, Microsoft Security Response Center, 2020.

[JJKD18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt : securing the foundations of the Rust programming language. POPL, 2018.

[LGC+20] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema : Application to work conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[MDL05] G. Muller, H. Duchesne, and J. L. Lawall. A framework for simplifying the development of kernel schedulers : design and performance evaluation. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 219–227, 2005.

[MFN+16] Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. A Coq library for internal verification of running-times. volume 9613, pages 144–162, 2016.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. volume 11423, pages 1–27, 2019.

[MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. pages 330–343, 2017.

[oca] The `ocaml-interop` Rust library. https://github.com/simplestaking/ocaml-interop.

[PA19] Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). ICFP, 2019.

[RFTO19] Reuben N. S. Rowe, Hugo Férée, Simon J. Thompson, and Scott Owens. Characterising renaming within ocaml's module system : Theory and implementation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 950–965, New York, NY, USA, 2019. Association for Computing Machinery.

[RRD14] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. JFP, 2014.

[SNRA18] Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. FabULous interoperability for ML and a linear language. ICFP, 2018.

[VH04] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, pages 86–101, Berlin, Heidelberg, 2004. Springer-Verlag.

[vM08] Eelis van der Weegen and James McKinna. A machine-checked proof of the average-case complexity of Quicksort in Coq. volume 5497, pages 256–271, 2008.

[voc] Projet Vocal : The verified OCaml library. https://vocal.lri.fr/.

[WRD+20] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical report, 2020.

[XWA+19] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. CHERIvoke : Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM International Symposium on Microarchitecture*, 2019.

[ZH18] Bohua Zhan and Maximilian P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. 2018.