PROJET DE RECHERCHE: TECHNIQUES FORMELLES POUR AIDER LA PROGRAMMATION

GABRIEL SCHERER

Une partie importante de la recherche en sciences du logiciel se concentre sur la question de la *vérification* : comment prouver qu'un programme correspond à sa spécification ?

J'aimerais me consacrer à un problème complémentaire : comment utiliser les outils formels développés dans la communauté des langages de programmation et ses voisines (théorie de la démonstration, logique, et démonstration automatique) pour *aider* l'écriture de programmes, en développant des outils et langages qui rendent la programmation plus aisée, plus concise, et réduisent les risques d'erreurs ?

Je propose trois axes principaux :

(1) L'étude de représentations canoniques des programmes dans des systèmes de types avancés. Pour ce faire, je compte continuer à explorer le lien entre théorie de la démonstration, et en particulier le *focusing*, et les langages de programmation. J'ai proposé dans ma thèse la technique de *saturation* qui étend le *focusing* pour obtenir des formes canoniques pour le λ-calcul simplement typé. Comment étendre cette saturation aux types polymorphes et dépendants ? J'espère obtenir comme résultat pratique un outil de synthèse de fragments de programmes dirigé par les types, même des types riches.

(2) L'élaboration de procédures de décision d'équivalence de programmes, et l'exploration des nouveaux outils et constructions de programmation qu'elles rendent possibles. J'espère obtenir comme résultat pratique des outils de vérification des restructurations de code (*refactoring*), légers et faciles d'emploi, qui augmentent donc la confiance et le confort des programmeurs.

(3) Un travail théorique et pratique sur la conception de systèmes de programmation multi-langages avec des garanties formelles de bonne inter-opérabilité, et donc d'utilisabilité et de sûreté. J'aimerais pouvoir prendre un langage monolithique, le transformer en un système langages séparés conçus pour interagir harmonieusement les uns avec les autres, et obtenir ainsi un meilleur système de programmation, plus simple à utiliser, et plus simple à faire évoluer.

La vérification et l'aide à la programmation par méthodes formelles sont fortement complémentaires. Les approches de vérification – par exemple la preuve de programme, les types dépendants, la vérification d'invariants logiques, le *model-checking* ou les tests – demandent à l'utilisateur de fournir des informations supplémentaires qui constituent une *spécification*, totale ou partielle, du programme, et qui restreignent donc l'espace des programmes possibles. Plus on a d'information sur le programme, plus l'espace à explorer est restreint, plus ces outils d'aide à la programmation peuvent être efficaces. (On peut voir les systèmes de types puissants comme une version plus généraliste des contraintes structurelles fortes imposées par l'expression ou le modelage de programme par des automates ou leurs variantes étendues.)

À l'inverse, la programmation certifiée avec spécifications avancées est aujourd'hui difficile d'utilisation et réservée aux experts ; de nouveaux outils d'aide à la programmation permettront d'espérer et de réclamer non seulement un gain en sûreté, mais aussi en confort de programmation. Ils pourraient alors jouer un rôle important dans l'adoption à plus grande échelle des méthodes de programmation vérifiée.

Par exemple, j'ai étudié pendant ma thèse la question de décider si un type a un habitant unique (tous les programmes possibles à ce type sont équivalents), et montré qu'on retrouve de tels types dans des bibliothèques logicielles utilisées en pratique, ce qui promet des applications à la synthèse de programmes dirigée par les types. Cela n'aurait pas été envisageable dans un langage moins typé, par exemple où les données sont représentées par des entiers – un type aux habitants pas du tout uniques. C'est grâce à l'utilisation de types plus informatifs, les types *abstraits* – introduits au départ pour la sûreté et la modularité – que les langages fonctionnels typés rendent réaliste la synthèse de programme dirigée par les types.

1. Focusing et langages de programmation

La correspondance de Curry-Howard entre preuves et programmes est maintenant bien connue et a donné pendant des décennies de beaux résultats théoriques et pratiques, côté logique (meilleure compréhension calculatoire de règles logiques), côté programmation ("*theorems for free*"), et dans des domaines mixtes comme les assistants de preuve à base de type dépendants.

Il existe aussi une communauté qui voit la recherche de preuve (dans une logique bien choisie) comme une forme de calcul correspondant précisément à la *programmation logique*, popularisée par le langage Prolog.

Mais la communauté des langages fonctionnels s'est concentrée sur le lien entre exécution des programmes et réduction/simplification des preuves, passant à côté d'une observation qui a été centrale à mes travaux de thèse : la structure de la *recherche* de preuve nous renseigne sur la structure des programmes (en forme normale). On peut voir tout programme fonctionnel comme une preuve, et donc le résultat d'un processus de recherche ; mieux comprendre la structure de cette recherche, par exemple par le *focusing*, permet de mieux comprendre les programmes, comment les représenter et manipuler.

1.1. Saturer les types riches.

J'ai transposé pendant ma thèse la notion de *focusing* de la théorie de la démonstration aux langages de programmation, et ai obtenu, en ajoutant la notion de *saturation*, une représentation canonique des programmes utilisant des types dits "simples", représentants les types de données usuels en programmation : fonctions, paires, sommes disjointes. J'ai pu montrer ainsi que des types simples dont les habitants sont uniques apparaissent dans des programmes typés existants.

Mais les types *simples* que mon algorithme peut gérer ne font que décrire les structures de données ; il existe d'autres types dans les langages typés modernes, en particulier les types *polymorphes* qui expriment la généricité et les types *dépendants* qui peuvent exprimer des spécifications mathématiques.

Une notion de forme canonique qui prendrait aussi en compte ces types riches serait nécessaire à une application à plus grande échelle de cette technique pour la synthèse de programmes – et permettrait aussi d'appliquer cette technique à des types plus fins, et donc plus souvent à habitants uniques.

C'est un projet ambitieux qui comporte des obstacles scientifiques à surmonter. La décidabilité de l'équivalence de types en présence du type vide, résultat obtenu pendant mon post-doc en utilisant le *focusing* et la *saturation*, était resté un problème ouvert connu depuis au moins 1995. Nous espérons que cette nouvelle approche permettra aussi des avancées importantes pour les types riches.

Première difficulté, l'indécidabilité. Dans un système avec polymorphisme, trouver un habitant correspond à la recherche de preuve dans des logiques d'ordre supérieur : l'existence d'un habitant, et d'autant plus son unicité, deviennent indécidables. D'un algorithme qui termine sur tous les types simples, on passerait à une procédure partielle répondant parfois "je ne sais pas" – ce qui ne l'empêcherait pas d'être très utile aux programmeurs dans les cas où elle sait, tout comme un solveur SAT ou SMT peut exploser sur certaines requêtes mais reste un outil précieux dans les cas courants.

Une autre difficulté provient de la *paramétricité*, une garantie sémantique apportée par les types polymorphes, souvent présentée sous l'angle des "*theorems for free*". Par exemple, une fonction générique en le type de son entrée ne peut pas faire d'hypothèse sur ce type et ne peut donc pas inspecter la valeur de cette entrée. On peut ainsi prouver qu'une fonction de type $\forall\alpha, \alpha \to \alpha$, dans un langage totalement fonctionnel, ne peut être que la fonction identité. Mais aujourd'hui on ne sait pas, dans le cas général, remplacer le raisonnement sémantique sur la paramétricité par une approche syntaxique permettant l'automatisation par un algorithme – pour $\forall\alpha, \alpha \to \alpha$ on peut faire manuellement une recherche de forme normale, mais comment généraliser de façon uniforme à tous les types ?

Une autre formulation du même problème, plus formelle, est la suivante. On sait que pour les systèmes de types simples, l'équivalence observationelle, définie sémantiquement, correspond à la relation de $\beta\eta$-équivalence définie syntactiquement. En présence de polymorphisme, la relation de $\beta\eta$-équivalence est strictement plus faible que l'équivalence observationelle. Est-il possible de proposer une équivalence syntactique plus forte ?

Troisième difficulté à étudier, le *focusing* est exprimé par les logiciens en utilisant le *calcul des séquents*, une présentation des preuves qui ne correspond pas directement aux termes des langages fonctionnels. On comprend bien comment passer de l'un à l'autre en présence de types simples et de polymorphisme, mais la combinaison du calcul des séquents et des types dépendants, en présence de sommes, pose des problèmes théoriques non résolus, comme expliqué dans [Her05] – sans les sommes, voir par exemple [LDM11] pour un

calcul des séquents dépendant focalisé. (Étienne Miquey, doctorant dans l'équipe PPS du laboratoire IRIF, a fait un travail sur le sujet très prometteur mais pas encore publié.)

1.2. **Saturation et ré-écriture.** Un besoin important pour utiliser la recherche de représentations canoniques est de pouvoir faire une recherche modulo certaines hypothèses supplémentaires. Par exemple, en présence de deux variables $x, y : A$ et d'une fonction $r : A \to A \to \texttt{Bool}$ inconnue, l'espace des formes canoniques à un type B est fortement simplifié si on ajoute l'hypothèse que r est une relation symétrique, $r(x, y) = r(y, x)$.

De façon générale on peut vouloir énumérer les fragments de programme canoniques modulo une famille d'équations sur les variables libres utilisées par ces fragments – supposer qu'un paramètre est une loi de groupe, par exemple. Une restriction naturelle est de supposer les équations orientées en un système de ré-écriture. Mais les termes présents dans ces équations, fournies par l'utilisateur, ne sont pas en forme focalisée : comment les reconnaître efficacement pendant le processus de recherche des formes focalisées ?

1.3. **Synthèse de programmes arbitraires.** Quand j'ai commencé mes travaux de thèse, la question de la recherche de programmes dirigée par les types avait été relativement peu étudiée. Elle a fait l'objet d'un regain d'intérêt ces dernières années, avec en particulier les travaux parallèles de Peter-Michael Osera à UPenn et ses collaborateurs à Princeton [OZ15 ; Fra+16], et des travaux indépendants au MIT [PKS16]. Leur approche n'est pas de se restreindre au cas non-ambigu où le type détermine uniquement le fragment de programme, mais de demander des tests à l'utilisateur et de s'arrêter au premier programme qui les passe.

Même dans ce cadre, obtenir des représentations plus canoniques est crucial : la recherche par *force brute* de programmes se heurte vite à une explosion combinatoire, même sur des petites tailles, et toutes les idées sont bonnes pour restreindre la redondance parmi les programmes. Les premiers travaux de Osera [OZ15] présentaient d'ailleurs des heuristiques trouvées en tâtonnant, qui sont en fait des cas particuliers de la structure imposée par le *focusing*. J'ai pris contact avec ce groupe pendant mon post-doc ; leur proposer les termes focalisés saturés comme base de recherche pourrait conduire à une collaboration intéressante.

2. Équivalence de programmes

Les outils de preuve automatique ont fait d'énormes progrès et sont maintenant utilisés dans de nombreux domaines, dont la vérification de programmes et les sciences du logiciel en général – des techniques de recherche de preuve puissantes sont aussi utilisées, par exemple, pour les bases de données ou les ontologies. On a donc des approches fructueuses, et beaucoup d'implémentations utilisées en pratique, pour vérifier automatiquement qu'un programme répond à une spécification. Comparativement, il y a eu peu de travail sur la vérification automatique que deux programmes sont équivalents : les types sommes rendent la théorie délicate et sont rencontrés très vite en pratique. Par exemple, un booléen est un type somme, et déplacer un test plus tôt ou plus tard dans un programme correspond à une opération dite d'η-expansion dont la justification est non-triviale. Les progrès accomplis pendant ma thèse et mon post-doc devraient permettre de passer ces difficultés.

J'aimerais implémenter en pratique des outils de test d'équivalence pour des programmes fonctionnels et évaluer leur applicabilité. C'est un problème qui est infaisable (*intractable*) dans le cas général, et qui pourrait donc ne rien donner. Mais on peut espérer des applications concrètes quand on sait que les deux programmes à tester ne sont pas très différents, ou alors qu'un algorithme très incomplet serait suffisant.

2.1. **Vérification de *refactoring*.** Le *refactoring*, la restructuration de programmes, est une partie importante de l'activité de programmation. Elle est souvent une étape préalable à des changements plus ambitieux, faite séparément pour être plus facile à relire et vérifier par les collègues ou collaborateurs : un bon *refactoring* transforme le programme pour le rendre plus maintenable mais ne modifie pas son comportement.

Puisque ces transformations sont simples, on peut espérer qu'un test d'équivalence de programmes soit capable de vérifier leur validité, apportant un confort important aux utilisateurs. Le sujet a pourtant été peu étudié jusqu'à présent. Les rares outils en programmation fonctionnelle soit se restreignent à des manipulations de données de premier ordre [Cla+12] soit reposent sur un usage interactif [FSG15] où le programmeur combine explicitement des transformations unitaires – une méthode de travail assez éloignée de celle des programmeurs et programmeuses aujourd'hui. Des outils au niveau des modèles [BHE08] reposent sur l'équivalence de processus concurrents, aujourd'hui mieux comprise et implémentée, mais difficilement applicable aux langages fonctionnels.

2.2. **Vérification de compatiblité ou non-ambiguïté.** Dans le système de modules dits "applicatifs" des langages fonctionnels de la famille ML, il est possible de construire un module à partir d'un module paramétré, et d'un choix de paramètre (un autre module). Si le même module paramétré est appliqué à deux modules "identiques" passés en paramètre, les modules résultats sont compatibles – ils peuvent échanger des valeurs. C'est important pour faire communiquer deux bibliothèques ayant fait indépendamment le même choix de paramètre, sans devoir les modifier pour dépendre d'un parent commun.

Aujourd'hui, la notion d'identité des paramètres est définie par une approximation syntaxique qui repose sur les noms. Deux bibliothèques peuvent faire le même choix de paramètre indépendamment et coopérer ensuite, mais elles ne peuvent pas utiliser deux paramètres définis de la même façon, le paramètre doit venir d'une dépendance commune. Un algorithme robuste d'équivalence entre programmes (et modules) permettrait de lever cette restriction, en testant que deux paramètres (non syntaxiquements égaux) sont équivalents, pour rendre plus de modules compatibles et donc la programmation modulaire plus aisée.

Un autre cas d'usage proche correspond à des cas d'héritage en diamant : que faire quand un objet hérite les opérations de deux sous-objets qui définissent une opération de même nom ? On peut interdire ce cas, choisir un ordre arbitraire, produire deux opérations séparées, forcer l'utilisateur à renommer... Souvent les deux opérations étaient en fait définies de façon identique : un test d'équivalence de programme (même incomplet) permettrait de respecter l'intention du programmeur dans ce cas. Ce problème se présente aussi naturellement dans les formalisations de hiérarchies de structures algébriques, par exemple avec les espèces de Focalize [Ayr+08] ou les coercions en Coq [Gar+09] : un monoïde fini a la structure d'un ensemble fini et d'un monoïde, et ces deux sous-structures ont le même ensemble de support.

3. Conception de multi-langages simples et sûrs

L'évolution des langages généralistes tend vers l'accrétion de fonctionnalités pour mieux servir des domaines spécialisés divers, qui peuvent poser des problèmes de cohérence ou de complexité pour le langage dans son ensemble. OCaml, GHC Haskell, Scala ou encore C++ ont acquis au fil des années des fonctionnalités qui réjouissent leurs utilisateurs experts mais intimident leurs débutants, rendent leur enseignement plus difficile, rendent les compilateurs plus fragiles, et augmentent le coût et la difficulté de la construction d'outils gérant tout le langage. La recherche en langages de programmation intensifie ces difficultés en proposant des fonctionnalités puissantes mais souvent plus complexes et difficile à combiner entre elles : types dépendants, systèmes d'effets, types linéaires, sous-ensembles normalisants, opérateurs de contrôle, etc. Il est aujourd'hui nécessaire de faire diminuer cette complexité d'utilisation pour que les résultats de nos recherches puissent continuer à toucher un public le plus large possible.

3.1. **Conception de nouveaux systèmes multi-langages.** Je propose d'étudier la conception de systèmes *multi-langages*, dans lesquels plusieurs langages séparés co-opèrent pour couvrir les mêmes domaines applicatifs, en en gérant chacun seulement une petite partie. Idéalement, à expressivité égale ces systèmes seraient moins complexes à utiliser, en permettant en particulier aux utilisateurs qui le souhaitent d'apprendre et d'utiliser seulement une partie des langages du système.

Pour cela, il faut mieux comprendre les problèmes d'interactions entre langages : comment concevoir ces systèmes de façon à éviter les "fuites d'abstraction", où l'usage interne des fonctionnalités d'un langage peut casser les propriétés attendues par les utilisateurs d'un autre langage ? Une étude précise de cette question, telle que je l'ai commencée pendant mon post-doc à Northeastern, devrait nous donner des outils pour pallier aux intéractions problématiques entre langages existants, et pour concevoir de nouveaux systèmes multi-langages plus harmonieux.

Notre approche pour formaliser l'absence de fuites d'abstractions (*abstraction leaks*) est de prouver la propriété de *pleine abstraction* : une transformation f d'un langage S vers un langage T est pleinement abstraite, relativement à des notions d'équivalence $(=_S), (=_T)$ sur ces deux langages, si

$$\forall t, u \in S, \quad t =_S u \implies f(t) =_T f(u)$$

Quand on considère la relation d'équivalence contextuelle (ou observationelle), une transformation est pleinement abstraite si, quand deux fragments du langage source S sont indistinguables (aucune façon de les utiliser dans S ne peut observer de différence entre les deux), alors les fragments transformés sont indistinguables dans T : même en utilisant les fonctionnalités différentes du langage T, on ne peut pas observer de différence. Si l'on considère maintenant le plongement d'un langage S dans un multi-langage $S + S'$, ce plongement est pleinement abstrait si les raisonnements équationnels qui sont valides dans S le restent dans le multi-langage.

Pendant mon post-doc j'ai formalisé un cas d'étude, un système formé d'un langage fonctionnel simple et d'un langage à types linéaires – permettant des manipulations sûres de protocoles complexes ou de plus bas niveau – et prouvé que le plongement du langage fonctionnel dans le multi-langage est pleinement abstrait. J'aimerais mieux comprendre les avantages et inconvénients de cette approche par rapport aux conceptions de langages monolithiques intégrant les deux aspects, comme Mezzo [PP13] ou Rust [15], et je pense qu'une façon de rendre disponibles des constructions de plus bas niveau sans augmenter la complexité du langage pour les utilisateurs non-experts aurait de nombreuses applications.

À moyen terme, on imagine un langage fonctionnel généraliste, comme OCaml, accompagné d'un "langage avancé" complémentaire, qui permette d'écrire les implémentations internes avec un contrôle fin sur les performances (modifications fortes d'état à possesseur unique, contrôle des allocations et de la représentation mémoire...), avec une garantie d'utilisabilité : tant qu'une personne n'utilise que des interfaces exprimées en OCaml, elle n'a pas besoin de connaître le langage avancé. À long terme, de nombreux autres aspects avancés de la recherche en langage de programmation peuvent être proposés comme des langages séparés : types dépendants, typage fin des opérateurs de contrôle, typage graduel...

Pour concevoir un "langage du futur" qui ne soit pas un monstre, il faut chercher le *multi-langage* du futur.

3.2. **Étude et amélioration de multi-langages existants.** La question de la fuite d'abstraction se pose tout particulièrement à l'interface entre un langage typé et un langage non typé (typage graduel), entre un langage vérifié et un langage non-vérifié (WhyML et OCaml), ou encore entre un langage totalement fonctionnel (Coq) et un langage avec effets (OCaml) – dans ce dernier cas, voir les travaux prometteurs de [DTT16]. On peut aussi voir un problème de l'interaction entre langages au niveau des assistants de preuve : comment relier entre eux des développements menés avec des outils différents ? La question de l'interopérabilité gracieuse, de l'absence de cassures d'abstractions, se transforme en la question cruciale de la validité des preuves combinées. En effet, les assistants de preuve modernes ne contiennent pas que des dérivations statiques, mais aussi des comportements calculatoires, que la combinaison de prouveurs doit préserver : il s'agit bien d'interopérabilité à la fois statique et dynamique, comme en programmation.

Depuis deux langages existants, on peut toujours créer un multi-langage tel que les plongements des langages existants soient pleinement abstraits : il suffit que le système de typage du multi-langage interdise aux deux fragments d'échanger des données (ou de se passer le contrôle) – on programme mixte est alors seulement un programme de l'un des deux langages. Deux langages arbitraires peuvent souvent partager leurs types de données de base (les entiers par exemple) de façon sûre ; mais le partage de comportements d'ordre supérieur, de fonctions ou d'objets par exemple, est beaucoup plus délicat.

Par exemple, il n'est pas possible de représenter les fonctions de Coq en OCaml de façon sûre, puisque OCaml ne peut pas représenter statiquement le type des fonctions qui terminent toujours et ne produisent pas d'effets de bord. (À l'inverse on peut choisir une représentation monadique des fonctions OCaml en Coq.) Cela suggère d'enrichir OCaml avec un type permettant une meilleure interaction, ce qui pose le problème scientifique suivant : si l'on considère un langage ML enrichi d'un type de fonctions totalement pures, d'une part, et le calcul des constructions (inductives) d'autre part, leur plongement dans un multi-langage commun est-il pleinement abstrait ? Je suis convaincu que la réponse est "oui", mais obtenir le résultat peut être difficile. Les cas de l'interaction entre le lambda-calcul pur non typé et simplement typé [DPP16], et entre le lambda-calcul simplement typé (avec récursion) et un lambda-calcul avec polymorphisme [NBA16] n'ont été résolus que cette année.

L'étude des multi-langages et des propriétés de pleine abstraction est un outil de conception pour de nouveaux systèmes de programmation plus simples, mais aussi un outil d'amélioration des systèmes existants.

4. Équipes d'accueil

Trois laboratoires où mon projet de recherche trouverait parfaitement sa place m'ont fait l'honneur d'être prêts à m'accueillir. En ordre alphabétique : l'IRIF et son équipe PPS, le LIX et son équipe Parsifal, et le LRI et son équipe VALS.

J'ai eu la chance de suivre des cours et de côtoyer des membres de ces laboratoires pendant mes études : Roberto Di Cosmo (PPS) m'a enseigné la logique linéaire, Jean-Christophe Filliâtre (VALS) la compilation, Paul-André Melliès (PPS) la théorie des catégories, et Dale Miller (Parsifal) la programmation logique. J'ai visité depuis leurs laboratoires qui sont des endroits conviviaux où j'aurais plaisir à travailler.

Ces trois laboratoires ont chacun des projets de développement logiciel ambitieux et implémentés dans le langage OCaml, auxquels je pourrais donc aussi apporter mon expertise sur le développement OCaml, et plus généralement l'accompagnement des projets de logiciel libre.

4.1. **IRIF : PPS.** L'équipe PPS a des compétences riches en logique et en sciences du logiciel. Elle est au centre du développement de l'assistant de preuve Coq, qui est d'abord un outil de preuve mathématique, mais aussi un langage de programmation à types dépendants. Un langage au typage si riche – et donc aux contraintes de comportement si fortes sur les programmes – est un champ d'étude idéal pour des outils de programmation dirigée par les types. J'ai fait en 2013 une étude des types utilisés dans les programmes à types dépendants publiés dans la littérature, et de nombreux types à habitants uniques y apparaissent [Sch13]. Une fonctionnalité d'inférence non-ambiguë déterminée par les types y serait plus efficace et plus souvent employée que dans un langage fonctionnel ordinaire.

Alexis Saurin est un expert en *focusing* avec qui j'aurais plaisir à collaborer, sur l'extension des formes normales saturées à des types riches par exemple. Yann Régis-Gianas s'intéresse à de nombreux problèmes de sciences du logiciel ; par exemple, ses travaux récents avec Thibaut Girka sur le calcul des différences entre programmes [GMR15] sont bien alignés avec ma proposition d'étudier les *refactoring* de programmes. Yann a aussi un projet de langage purement fonctionnel certifiant, Pangolin, actuellement en pause et qui s'inscrit très bien dans mon projet global de recherche.

Hugo Herbelin et Pierre-Louis Curien sont les initiateurs d'une famille de travaux sur des langages de programmation pour machines abstraites dont les formulations récentes, en particulier les calculs polarisés établis avec Guillaume Munch-Maccagnoni (non PPS), sont liées à la fois au *focusing* et à l'étude théorique des effets de bords dans les langages de programmation.

Le logiciel Coq est une grande réussite à laquelle contribuent de façon centrale les membres de PPS, et présente des problématiques délicates de développement logiciel. Un travail important d'ouverture du développement a été accompli en 2016, grâce en particulier aux efforts de Maxime Dénès (non PPS), mais cela reste un stade préliminaire comparé à la situation du langage OCaml – j'aimerais contribuer à faire progresser les contributions extérieures. Coq est aussi un programme exigeant qui utilise beaucoup d'aspects avancés du langage OCaml, et nécessite parfois la correction de problèmes dans l'implémentation d'OCaml ou motive de nouvelles fonctionnalités. J'ai été pendant ma thèse en contact régulier avec l'équipe Coq en tant que développeur de l'implémentation OCaml (Pierre-Marie Pédrot, Maxime Dénès, Hugo Herbelin ou Matthieu Sozeau pourront le confirmer), et je serais ravi de renforcer ce pont entre leurs développeurs.

Enfin, certaines parties de mon projet de recherche appellent à l'utilisation d'outils de démonstration automatique et une acquisition de compétences dans ce domaine. Cette direction serait un apport intéressant aux compétences de PPS aujourd'hui et pourrait ouvrir la porte à une collaboration fructueuse avec d'autres équipes (au LRI et au LIX, mais aussi au LORIA et au LSV).

4.2. **LIX : Parsifal.** L'équipe Parsifal du laboratoire LIX est aujourd'hui le meilleur endroit au monde pour travailler sur le *focusing*. Elle a par exemple accompli un travail remarquable sur l'étude des liens fins entre le focusing et les stratégies existantes de recherche de preuve et de programmation logique [Cha10 ; FLM13], et son travail sur l'adaptation du focusing à la logique intuitioniste [CM09 ; BS10] et sur les formes canoniques de preuves multi-focalisées [SCM08 ; CHM12] m'a beaucoup inspiré.

Stéphane Lengrand et Kaustuv Chaudhuri ont une expertise riche en preuve automatique dont j'aimerais profiter. Stéphane connaît les approches SMT, grâce en particulier à des collaborations fréquentes avec le SRI à Stanford, et travaille à les intégrer dans un assistant de preuve. Kaustuv est un expert de la "méthode inverse", une approche de la recherche de preuve par saturation qui est performante sur des problèmes difficiles en logique linéaire ou intuitionniste.

J'apporterais à cette équipe une expertise en conception de langages de programmation fonctionnels. L'étude de la programmation logique est un point fort de Parsifal, et mes travaux ont montré que les deux points de vue incarnés par ces deux approches (les programmes comme preuves et comme résultat d'une recherche de preuve) peuvent produire des résultats nouveaux.

Enfin, je contribuerais aux développements logiciels du projet Parsifal, notamment l'assistant de preuve Abella, particulièrement adapté à l'étude des langages de programmation et qui commence à être utilisé en-dehors de l'équipe, et le projet plus expérimental Psyche, qui utilise la recherche sur le *focusing* pour trouver un équilibre harmonieux entre preuve automatique et preuve interactive.

4.3. **LRI : VALS.** L'équipe VALS développe la plateforme de vérification de programme Why3, qui permet à ses utilisateurs d'inclure dans leurs programmes des invariants logiques qui sont vérifiés automatiquement. Elle possède aussi une compétence très forte en démonstration automatique, avec en particulier le solveur SMT Alt-Ergo et le vérifieur de modèle Cubicle, et en l'art délicat de combiner ces outils avec les obligations de preuves générées par la vérification logicielle. Ce travail, qui progresse depuis de nombreuses années, donne à cette équipe une expérience pratique très rare de la programmation aidée par la démonstration automatique, dans laquelle mon projet de recherche s'inscrit parfaitement.

Mon projet d'étude de l'interopérabilité entre langages, que ce soit entre le certifié et le non-certifié ou le haut-niveau et le bas-niveau, correspond aussi à un besoin découlant naturellement des travaux de cette équipe. En plus de Why3, Thibaut Balabonski travaille sur le projet Mezzo, mélangeant programmation fonctionelle et linéarité – sous la forme différente et intéressante d'une logique de séparation – et Jean-Christophe Filliâtre coordonne le projet ANR VOCAL, qui construit une bibliothèque généraliste vérifiée pour OCaml, et pose donc des questions d'interaction certifié / non-certifié et d'interaction entre les assistants de preuve.

Le langage WhyML au centre de la plateforme Why3 est un langage qui combine des traits fonctionnels et impératifs. Cela a demandé des développements intéressants et utiles pour la vérification déductive, mais cela limite aussi la liberté des transformations de programme par rapport à un langage aux effets plus restreints (par exemple avec un système d'effets) ; deux appels de fonctions indépendants ne peuvent pas forcément être réordonnés, par exemple. En 2009, VALS avait commencé l'intégration d'un langage fonctionnel d'ordre supérieur dans leur plateforme de programmation certifiée [KF09], que je souhaiterais reprendre et continuer : c'est un excellent cadre pour appliquer des techniques de démonstration automatique aux langages fonctionnels, et pour étudier les questions d'interopérabilité entre langages.

Références

[Her05] Hugo Herbelin. *C'est maintenant qu'on calcule, au cœur de la dualité*. habiliation thesis. 2005.

[Ayr+08] Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, Lionel Habib, Thérèse Hardin, Mathieu Jaume, Charles Morisset, François Pessaux, Renaud Rioboo et Pierre Weis. "Trusted Software within Focal". C&ESAR, 2008.

[BHE08] Dénes Bisztray, Reiko Heckel et Hartmut Ehrig. "Verification of architectural refactorings by rule extraction". FASE, 2008.

[SCM08] Alexis Saurin, Kaustuv Chaudhuri et Dale Miller. "Canonical Sequent Proofs via Multi-Focusing". IFIP TCS, 2008.

[CM09] Liang Chuck et Dale Miller. "Focusing and Polarization in Linear, Intuitionistic, and Classical Logics". *Theoretical Computer Science* 410.46 (2009).

[Gar+09] François Garillot, Georges Gonthier, Assia Mahboubi et Laurence Rideau. "Packaging mathematical structures". TPHOLs, 2009.

[KF09] Johannes Kanig et Jean-Christophe Filliâtre. "Who : A Verifier for Effectful Higher-order Programs". ML Workshop, 2009.

[BS10] Taus Brock-Nannestad et Carsten Schürmann. "Focused Natural Deduction". LPAR, 2010.

[Cha10] Kaustuv Chaudhuri. "Magically Constraining the Inverse Method Using Dynamic Polarity Assignment". LPAR, 2010.

[LDM11] Stéphane Lengrand, Roy Dyckhoff et James McKinna. "A Focused Sequent Calculus Framework for Proof Search in Pure Type Systems". *Logical Methods in Computer Science* 7.1 (2011).

[CHM12] Kaustuv Chaudhuri, Stefan Hetzl et Dale Miller. "A Systematic Approach to Canonicity in the Classical Sequent Calculus". EACSL, 2012.

[Cla+12] Koen Claessen, Moa Johansson, Dan Rosén et Nicholas Smallbone. "HipSpec : Automating Inductive Proofs of Program Properties." ATX workshop, 2012.

[FLM13] Mahfuza Farooque, Stéphane Lengrand et Assia Mahboubi. "A bisimulation between DPLL(T) and a proof-search strategy for the focused sequent calculus". LFMTP, 2013.

[PP13] François Pottier et Jonathan Protzenko. "Programming with permissions in Mezzo". ICFP, 2013.

[Sch13] Gabriel Scherer. "Mining opportunities for unique inhabitants in dependent programs". 2013.

[15] https://en.wikipedia.org/wiki/Rust_(programming_language). 2015.

[FSG15] Andrew Farmer, Neil Sculthorpe et Andy Gill. "Reasoning with the HERMIT : Tool Support for Equational Reasoning on GHC Core Programs". Haskell symposium, 2015.

[GMR15] Thibaut Girka, David Mentré et Yann Régis-Gianas. "A Mechanically Checked Generation of Correlating Programs Directed by Structured Syntactic Differences". *Automated Technology for Verification and Analysis*. 2015.

[OZ15] Peter-Michael Osera et Steve Zdancewic. "Type-and-Example-Directed Program Synthesis". PLDI, 2015.

[DTT16] Pierre-Évariste Dagand, Nicolas Tabareau et Éric Tanter. "Partial type equivalences for verified dependent interoperability". ICFP, 2016.

[DPP16] Dominique Devriese, Marco Patrignani et Frank Piessens. "Fully abstract compilation by approximate back-translation". POPL, 2016.

[Fra+16] Jonathan Frankle, Peter-Michael Osera, David Walker et Steve Zdancewic. "Example-directed synthesis : a type-theoretic interpretation". POPL, 2016.

[NBA16] Max S. New, William J. Bowman et Amal Ahmed. "Fully Abstract Compilation via Universal Embedding". ICFP, 2016.

[PKS16] Nadia Polikarpova, Ivan Kuraj et Armando Solar-Lezama. "Program synthesis from polymorphic refinement types". PLDI, 2016.