RAPPORT SUR LES TRAVAUX EFFECTUÉS

GABRIEL SCHERER

La démarche d'ensemble de mon travail est de découvrir et d'attaquer les problèmes théoriques dont la résolution est nécessaire pour pouvoir concevoir de meilleurs langages de programmation, permettant d'écrire des programmes plus concis, plus sûrs et plus faciles à faire évoluer, ou écrire des outils d'aide à la programmation pour améliorer l'usage de langages existants.

1. Types à habitants uniques

Certaines constructions des langages de programmation permettent à un ordinateur de "deviner" des parties du programme que le programmeur humain aurait écrit, en s'aidant du contexte. Cela peut apporter beaucoup de concision et d'aisance, mais aussi poser des problèmes de compréhension et provoquer des erreurs : comment savoir si ce qui est "deviné" correspond à ce qu'on souhaitait exprimer ? C'est une source de difficultés de conception pour les langages ayant une forme d'implicite (Haskell, Scala, Rust, Coq), qui doivent se restreindre à des cas simples et ajouter des restrictions souvent *ad-hoc* ou non modulaires.

Pour mieux comprendre les fondements théoriques de cette opération, j'ai étudié pendant ma thèse, sous la direction de Didier Rémy, le cas où le contexte détermine uniquement le programme à deviner. Dans un langage typé, un jugement de typage représente ce qu'on peut déduire, statiquement, des attentes réciproques entre un fragment de programme et le contexte qui l'entoure. On peut donc reformuler la question de la non-ambiguïté ainsi : quels sont les types qui ont un habitant unique (modulo équivalence de programmes) ?

Plus le système de typage est riche, plus il permet de modéliser finement les attentes du contexte ; mais chercher des habitants à ces types riches et vérifier leur équivalence devient aussi plus difficile. Pendant ma thèse, j'ai démontré que ce problème était décidable pour un système de types dits "simples" qui exprime tous les types de données usuels des langages de programmation : fonctions, paires (produit cartésien), et sommes disjointes ("un entier ou un booléen").

Les sommes posent des difficultés pour décider l'équivalence de programmes – premier pas vers la compréhension de l'unicité. Alors qu'on sait décider l'équivalence de programmes avec des fonctions et paires depuis les années 1970, l'équivalence de programmes en présence de sommes disjointes n'a été décidée qu'en 1995 [1], et la question toujours étudiée dans les années 2000 [2, 3, 4]. Il s'agit de concevoir un processus de "normalisation" des programmes, tel qu'il soit facile de déterminer l'équivalence de deux formes normales.

En plus de ces travaux de la communauté des langages de programmation, j'ai choisi d'utiliser le *focusing*, un outil de la théorie de la démonstration. Les logiciens cherchent des représentations "canoniques" des preuves formelles, qui ne contiennent pas de redondances – deux représentations distinctes qui exprimeraient intuitivement la "même" preuve. Ce mélange des genres, programmation fonctionnelle et théorie de la démonstration, n'avait jamais été proposé dans ce contexte. J'ai pu prouver qu'il avait un sens dans l'article [13], présenté à TLCA 2015, où je montre une correspondance formelle entre le concept logique de *multi-focusing maximal*, développé dans les années 2000 [5, 6], et les processus de normalisation de programmes avec des sommes [4] . Cette forme de normalisation, cependant, demande de partir d'un programme connu, et ne permet pas d'énumérer les formes normales à un type donné.

En ajoutant au *multi-focusing* une notion de "saturation", j'ai pu obtenir une forme normale "canonique" énumérable des programmes fonctionnels simplement typés, où deux représentations distinctes correspondent à des programmes qui se comportent différemment. Pour décider la question de l'habitation unique, il suffit alors d'énumérer ces représentations distinctes, en s'arrêtant à deux. L'étude de l'*espace de recherche* des preuves correspond donc bien à celle de représentations plus *canoniques* d'un langage de programmation. Ce travail [15] a été présenté à ICFP 2015, et une version journal a été soumise à l'été 2016.

Machines abstraites. Les "machines abstraites" sont une famille de représentations des programmes de plus bas niveau, qui sont plus faciles à exécuter par une machine. J'ai collaboré avec Guillaume Munch-Maccagnoni, alors post-doc à Cambridge (UK), pour mieux comprendre les liens entre mes travaux et sa

recherche sur les machines abstraites polarisées. Nous avons montré que les machines polarisées sont une bonne syntaxe pour étudier le focusing et les sommes dans un article [12] présenté à LICS 2015.

J'ai aussi collaboré avec Pierre-Évariste Dagand, alors post-doc dans mon laboratoire de thèse (Gallium), pour exposer un lien entre ces machines abstraites et les techniques de "réalisabilité classique" de Jean-Louis Krivine (PPS/IRIF) : une preuve par réalisabilité qu'une classe de programmes termine correspond à un évaluateur par machine abstraite. Ce travail fut présenté aux JFLA 2015 [11], avec une version journal en préparation.

La notion de polarisation joue un rôle central dans les études actuelles [16] sur la sémantique des effets de bords dans les langages de programmation, c'est donc un premier pas vers une meilleure intégration des effets de bords dans les constructions et outils que j'étudie.

Équivalence de programmes. À la fin de mon travail de thèse, j'ai réalisé des expérimentations sur l'implémentation de mon algorithme de décision qui m'ont permis de découvrir un fait surprenant : il semblait possible d'ajouter le *type vide* à mon implémentation sans casser ses bonnes propriétés. C'est étonnant puisque la question de décider l'équivalence en présence de ce type vide était toujours ouverte [1]. La difficulté vient de l'effet explosif du type vide sur l'équivalence de programme : si les hypothèses d'un jugement de type permettent de construire un habitant du type vide (de montrer une contradiction), alors tous les programmes de ce type sont équivalents entre eux. Le type vide n'est pas couramment utilisé dans des programmes, mais joue parfois un rôle important en représentant le fait qu'un cas ne peut pas se produire.

Cette expérience pratique suggérait la possibilité d'étendre ma notion de formes normales (programmes focusés saturés) à un système avec un type vide, et en particulier un algorithme pour décider l'équivalence de programme dans ce cadre, présenté aux journées LAC-GeoCal du GdR IM. Prouver sa correction demande des outils de preuve supplémentaires, une étude plus fine de la saturation, et que je n'ai pas eu le temps de développer pendant ma thèse ; j'ai achevé ce travail pendant mon post-doc, et le résultat a été accepté [17] à POPL 2017.

2. Types dépendants, réduction forte, ressources

Dans le cadre d'un stage avec Andreas Abel à Munich en 2011, j'ai travaillé sur les systèmes à types dépendants, des types très riches permettant de représenter les énoncés mathématiques, et donc de construire des assistants de preuve. Nous avons travaillé sur une construction de modèle (par relations logiques) pour montrer que le système de type, vu comme une logique, est cohérent, qu'il ne permet pas de prouver d'énoncé faux. Le travail a été publié dans le journal "Logical Methods in Computer Science" (LMCS) [7].

Dans la plupart des langages de programmation typés, on peut évaluer les différentes parties du programme dans n'importe quel ordre, cela change le comportement du programme mais ne peut pas casser la sûreté du typage. Dans son travail de thèse [10], Julien Crétin, aussi encadré à Gallium par Didier Rémy, a proposé une construction permettant aux programmes de supposer des invariants logiques, et les utiliser implicitement pour vérifier le typage. Cette construction empêche d'évaluer dans n'importe quel ordre, car évaluer sous une hypothèse fausse peut casser le typage. J'ai étendu son travail par une étude plus fine des stratégies de réduction valides dans ce cadre, permettant d'expliquer précisément à quels endroits l'évaluation doit être bloquée. Ce travail a été présenté aux journées LTP du GdR GPL puis publié à ESOP 2015 [14]. Il a permis de mieux comprendre que les types algébriques dits "gardés", qui sont utilisés en OCaml et Haskell et font des hypothèses implicites d'égalités entre types, ne sont sûrs qu'avec certains ordres d'évaluation.

Enfin, à la suite d'une courte visite à Yale en 2013, j'ai collaboré avec Jan Hoffmann sur un problème rencontré par les systèmes de types avec analyse de ressources (temps, mémoire), qui ne gèrent pas bien les fonctions qui renvoient elles-mêmes des fonctions. Notre travail [8], publié à LPAR 2013, propose une modification des types de fonctions pour garder trace des ressources attribuées non seulement aux paramètres de la fonction, mais aussi aux variables du contexte ambiant qu'elles utilisent.

3. Développement du langage OCaml

L'équipe-projet INRIA où j'ai fait ma thèse, Gallium, est le berceau historique du langage de programmation OCaml, aujourd'hui utilisé et développé plus largement dans la recherche et l'industrie, en France et à l'étranger. J'ai participé depuis le début de ma thèse au développement du langage – sa spécification et son

1. TLCA list of Open Problems, Problem 15 (Alex Simpson, 1995)

implémentation. Selon les mesures d'activités par nombres de *commits*/modifications [2], je suis aujourd'hui le troisième contributeur le plus actif, et le deuxième sur la période de 2012 à aujourd'hui. Je maintiens aussi l'outil de compilation de projet ocamlbuild, utilisé par plus de 4000 des 6000 paquets OCaml du répertoire centralisé OPAM, et l'extension communautaire de la bibliothèque standard, batteries.

Un de mes axes de contribution a été de faire évoluer le processus de développement pour le rendre plus ouvert aux contributions extérieures. OCaml avait il y a quelques années la réputation d'être un projet au développement assez fermé, accueillant tièdement les contributions externes pour améliorer le langage ou son implémentation. J'ai proposé l'utilisation de la plateforme de développement Github pour faciliter les contributions extérieures, et fourni une grande partie du travail de revue et de retours sur ces contributions, pour les encourager. Entre 2006 et 2012, 11 développeurs différents, dont un seul ne faisait partie ni de l'équipe de développement ni de Gallium, ont apporté des changements et améliorations à l'implémentation principale. Sur la période 2012-2016, 93 personnes ont contribué.

Ma participation au développement du langage OCaml a soulevé des problématiques de recherche variées. J'ai travaillé avec Nicolas Pouillard à une proposition d'espaces de noms (*namespaces*) pour le langage, qui n'a pas été retenue, mais a permis de mieux comprendre les choix de conception. Mon étude avec Didier Rémy de la variance des types algébriques gardés a été présentée au ML Workshop en 2012 et publiée [9] à ESOP 2013. Mon travail avec Luc Maranget sur les variables ambiguës dans un motif, présenté au ML workshop 2016, a une version longue en cours de rédaction.

4. Travaux en cours

Je suis depuis Janvier 2016 en post-doc à Northeastern University, Boston, sous la direction d'Amal Ahmed.

Programmation multi-langages. Les langages de programmation actuels accumulent des fonctionnalités pour couvrir le plus de domaines applicatifs possibles, au prix d'une complexité et difficulté d'apprentissage croissantes. Une alternative au langage-mastodonte, déjà utilisée en pratique, est d'écrire des programmes hybrides en utilisant plusieurs langages simultanément, chacun sur la partie où il peut briller. Malheureusement, les propriétés théoriques de ces systèmes multi-langages sont beaucoup moins étudiées et mal comprises.

Quand on mélange plusieurs langages, on voudrait que la coopération entre eux soit la plus harmonieuse possible. S'il y a des différences sémantiques qui font qu'un des langages viole des propriétés supposées par les autres, cela peut introduire des comportements inattendus par les programmeurs, qui ne connaissent pas forcément tous les langages du système, et des erreurs. Une meilleure compréhension théorique de ces systèmes est essentielle pour mieux définir la notion de "coopération harmonieuse" et pouvoir la vérifier concrètement.

Je suis venu travailler avec Amal Ahmed car elle est spécialiste d'outils formels, les relations logiques et la pleine abstraction, qu'elle utilise dans le cadre de la spécification des compilateurs – un compilateur manipule aussi plusieurs langages, un langage source, un langage assembleur, et des représentations intermédiaires. Je lui ai proposé d'utiliser ces outils sur ce problème d'étude formelle de systèmes multi-langages.

Nous avons soumis à l'automne un article à ce sujet, avec Amal et un de ses étudiants en thèse, Max New. Dans cet article, je propose et j'étudie un système multi-langage, où l'un des langages est une version idéalisée d'un langage fonctionnel comme OCaml, et l'autre un langage de plus bas niveau avec types linéaires, permettant de raisonner finement sur la possession de la mémoire, et donc de réécrire des pointeurs de façon sûre. J'ai établi le théorème qu'ajouter ce langage bas-niveau ne "casse pas l'abstraction" du langage fonctionnel : deux programmes indistinguables dans le langage fonctionnel le restent dans le système hybride. Autrement dit, un programmeur qui ne connaît que le langage fonctionnel peut utiliser des bibliothèques qui utilisent en sous-main le langage bas niveau, sans risquer de mauvaises surprises.

Vérification d'optimisations spéculatives. Indépendamment, j'ai commencé à l'automne, avec Amal Ahmed, Jan Vitek (aussi à Northeastern) et quelques étudiants de Jan, un projet de vérification formelle des optimisations "spéculatives" faites par les implémentations avancées (just-in-time) de langages dynamiques – Jan Vitek étudie le langage R, mais ces techniques s'appliquent aussi à Javascript, Matlab ou Smalltalk.

Un compilateur statique ne fait des optimisations que quand il peut montrer qu'elles sont toujours valides. Pendant une itération sur un tableau, une hypothèse sur les éléments peut être faite si elle s'applique à tous les éléments. Un langage dynamique encourage un style de programmation moins structuré où tous les types de données se mélangent. L'optimisation spéculative consiste à faire des paris à partir d'observation passées (les

2. https://github.com/ocaml/ocaml/graphs/contributors, pseudonyme `gasche`

1000 premiers éléments de cette boucle étaient des flottants), compiler le code comme si le "pari" était toujours vrai (on aura toujours des flottants pour la suite), en laissant seulement une "garde", un test qui, s'il devient faux, provoque la *dé-optimisation* dynamique du programme et sa recompilation. Cette approche permet de faire beaucoup d'hypothèses supplémentaires, sans prendre en compte les cas où elles ne s'appliquent pas, et donc d'optimiser plus agressivement.

Elle présente un challenge du point de vue de la vérification : comment prouver que ces optimisations préservent le comportement du programme ? Aujourd'hui les ingénieurs qui maintiennent ces compilateurs avancent un peu à tâtons, en restant très proches des détails d'implémentation, et ont souvent du mal à répondre à la question de quelles transformations sont valides ou non dans ce cadre.

Nous avons développé un langage simple [3] pour raisonner sur ces transformations spéculatives, et commencé à travailler sur la justification de leur validité.

5. Enseignement et service communautaire

J'ai enseigné la programmation en tant que chargé de travaux pratiques. 3 ans de Caml Light en MPSI, 2 ans de Java en L1, et 1 an de C en L3 et de OCaml en M1. Dans chaque cas, j'ai rédigé des sujets.

J'apprécie beaucoup de faire des revues d'articles, et les personnes qui m'en envoient semblent satisfaites de mon travail – demander par exemple à Jade Alglave, Michele Pagani, Jan Vitek ou Boris Yakobowski. J'ai eu le plaisir de faire jusqu'à présent 17 revues pour des articles de conférence et 2 revues pour un journal (MSCS). J'en ferai quelques autres en 2017 en tant que membre du comité de programme de ICFP.

J'ai aussi aidé à organiser des conférences. J'ai organisé NEPLS (*New England Programming Language Symposium*) à Northeastern en Octobre dernier [4], et j'ai été *Student Volunteer Captain*, en charge de l'organisation des volontaires étudiants, à ICFP 2015 (23 volontaires) et ICFP 2016 (33 volontaires). En 2017, je serai aussi *program chair* du workshop OCaml, hébergé par ICFP.

Références

[1] Neil Ghani. "Beta-Eta Equality for Coproducts". TLCA, 1995.

[2] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann et Philip J. Scott. "Normalization by Evaluation for Typed Lambda Calculus with Coproducts". LICS, 2001.

[3] Vincent Balat, Roberto Di Cosmo et Marcelo P. Fiore. "Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums". POPL, 2004.

[4] Sam Lindley. "Extensional Rewriting with Sums". TLCA, 2007.

[5] Dale Miller et Alexis Saurin. "From Proofs to Focused Proofs : A Modular Proof of Focalization in Linear Logic". CSL, 2007.

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

[7] Andreas Abel et Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". *Logical Methods in Computer Science* (2012). (*apport* : auteur secondaire).

[8] Gabriel Scherer et Jan Hoffmann. "Tracking Data-Flow with Open Closure Types". LPAR, 2013.

[9] Gabriel Scherer et Didier Rémy. "GADTs meet subtyping". ESOP, 2013.

[10] Julien Crétin. "Erasable coercions : a unified approach to type systems". Thèse. Paris-Diderot, 2014.

[11] Pierre-Évariste Dagand et Gabriel Scherer. "Normalization by realizability also evaluates". JFLA, 2015. (*apport* : auteur principal).

[12] Guillaume Munch-Maccagnoni et Gabriel Scherer. "Polarised Intermediate Representation of Lambda Calculus with Sums". LICS, 2015. (*apport* : partage égal).

[13] Gabriel Scherer. "Multi-focusing on extensional rewriting with sums". TLCA, 2015.

[14] Gabriel Scherer et Didier Rémy. "Full reduction in the face of absurdity". ESOP, 2015.

[15] Gabriel Scherer et Didier Rémy. "Which simple types have a unique inhabitant?" ICFP, 2015.

[16] Pierre-Louis Curien, Marcelo Fiore et Guillaume Munch-Maccagnoni. "A Theory of Effects and Resources : Adjunction Models and Polarised Calculi". POPL, 2016.

[17] Gabriel Scherer. "Deciding equivalence with sums and the empty type". POPL, 2017.

3. https://github.com/reactorlabs/sourir/

4. Un workshop d'une journée à portée régionale. Programme de mon édition : http://nepls.org/Events/30/