Activités de recherche

Gabriel Radanne

Résumé

La programmation est une tâche difficile. Le programmeur est confronté a de nombreux obstacles : comment exprimer son intention dans le langage de son choix ? Comment s'assurer qu'un tel programme est correct ? Comment le composer avec d'autres programmes ? Comment le rendre plus efficace ?

Le langage de programmation est l'acteur central de toutes ces questions, et l'outil principal du programmeur. Mon but est d'améliorer ou de concevoir des langages de programmation, afin de les rendre plus facile d'utilisation, plus sûrs, plus efficaces ou encore plus modulaires. Je m'intéresse tout particulièrement à l'adaptation du langage au domaine métier du programmeur. Ceci m'a mené à m'intéresser à de nombreux domaines, du Web à la programmation système. Enfin, je dote les langages que je développe d'une sémantique formelle, ce qui est essentiel pour prouver l'ensemble des propriétés souhaitées sur un langage.

1 Contexte scientifique

La programmation est une tâche difficile. D'autant plus difficile est la tâche d'écrire des programmes *corrects*. De nombreuses techniques ont été utilisées pour cette tâche telle que les tests et la vérification statique. Ces techniques peuvent encore être raffinées via l'amélioration du principal outil de programmation : le langage de programmation lui-même.

Malheureusement, la plupart des langages de programmation sont trop limités et régulièrement utilisés dans des contextes où ils sont peu adaptés. On peut citer les nombreux programmes concurrents écrit en C (dont le modèle mémoire qui, bien que permettant l'écriture de code performant, est particulièrement difficile à prendre en main) ; ou encore les larges applications Web écrites en Javascript (qui ne fournit ni encapsulation ni abstraction ce qui limite fortement la modularité). La programmation est une lutte constante pour transformer le code que le programmeur veut écrire en le code que le langage accepte. De plus, la plupart des langages n'aident aucunement dans la tâche délicate de vérifier les programmes.

Améliorer les langages de programmation mène à de nombreux avantages. Par exemple, les systèmes de types riches permettent de vérifier des propriétés de plus en plus complexes directement pendant l'écriture du programme. Des langages bien conçus sont également plus propices à l'analyse statique ou aux optimisations poussées. Outre la correction, les langages de programmation modernes fournissent également de nombreux avantages ergonomiques tels que la vérification de la documentation, l'auto-complétion ou encore la capacité à refactorer du code simplement en suivant les erreurs de compilation. Dans tout ces cas, la sûreté et l'ergonomie du langage sont améliorées via une conception en tandem avec les analyses associées, que ce soit

des systèmes de types ou des vérifications statiques. Les méthodes formelles sont bien entendu essentielles pour s'assurer de la correction de ces analyses.

Cette capacité à aider les programmeurs à écrire des programmes corrects a conduit à un récent regain de popularité des langages fortement typés. Facebook a introduit Hack et Flow [2, 1] pour typer statiquement leur gigantesque base de code PHP et Javascript et Infer [4] pour vérifier statiquement leurs applications mobiles (écrites en Java, C# ou Objective-C). Mozilla a créé le langage Rust pour aider à la conception d'une nouvelle génération de navigateur Web (dont la complexité s'approche maintenant des systèmes d'exploitations). Microsoft promeut maintenant Typescript et F# pour écrire des applications complexes à la fois pour le Web et en natif. JetBrains a conçu Kotlin, un langage pour la JVM dont le but principal est un excellent support IDE via des analyses statiques variées. Dans tous ces cas, les fonctionnalités du langage ont été conçues dans le but explicite de supporter certains usages spécifiques, de l'ownership de Rust à la logique de séparation dans Infer.

Ma recherche est centrée sur les langages de programmation : comment les rendre plus expressifs, sûrs, faciles d'utilisation et extensibles. Je dote également les langages que je develope d'une sémantique formelle, ce qui est essentiel pour prouver l'ensemble des propriétés souhaitées sur un langage. Le but de ma recherche est d'améliorer le support des langages de programmation pour des *domaines spécifiques*, et ce via trois types d'approches :

— La définition de nouvelles fonctionnalités et de nouveaux langages adaptés à un domaine donné tel que la programmation Web (Section 2) ou les systèmes d'exploitation (Section 3).
— L'amélioration de langages existants avec des fonctionnalités génériques qui complément les aspects dédiés aux domaines. En particulier, j'ai travaillé sur les types linéaires, qui permettent une gestion sûre des ressources (Section 4) et sur l'analyse de terminaison pour le C (Section 5).
— Un langage n'est rien sans un écosystème. Dans cet optique, je conçois également des bibliothèques qui complémentent le langage (Sections 6 et 7).

Ma recherche est souvent accompagnée de développements logiciels que je décris également, annotés avec une auto-évaluation suivant le module INS2I [1].

2 Modularité et expressivité pour la programmation Web sans-étages

Les applications Web sont généralement décomposées en étages : un programme client généralement écrit en Javascript exécuté dans un navigateur Web, et un programme écrit dans un langage arbitraire exécuté sur un serveur. Les langages sans-étages (« tierless » en anglais) permettent aux programmeurs d'exprimer les aspects clients, serveurs, mais également base de donnée, dans un même programme. Les langages sans-étages regroupent ainsi tous les aspects du programme et permettent une vérification du programme Web dans son ensemble, pour vérifier la sûreté des communications, par exemple. Les langages sans-étages sont généralement de nouveaux langages indépendants, ce qui rend les bibliothèques disponibles limitées. De plus, ces langages utilisent généralement des analyses dynamiques, et sont donc difficile à marier avec un typage statique. La modularité et la compilation séparée sont également cruciale dans la programmation Web car elles raccourcissent le cycle de compilation et permettent une meilleure isolation des bibliothèques.

1. https://csins2i.irisa.fr/files/2016/09/FicheLogicielsCliquableCSI-INS2I.pdf

Le but du projet Ocsigen est de fournir des outils pour la programmation Web en OCaml tels un serveur Web et un compilateur d'OCaml vers Javascript. Eliom est le composant langage du projet Ocsigen et consiste en une extension d'OCaml pour la programmation Web sans-étages. Eliom était initialement implémenté par une extension de syntaxe pour OCaml. Malheureusement, cette extension n'avait pas de typage sûr ou de spécification formelle et avait une mauvaise intégration avec le système de type d'OCaml. Dans le cadre de ma thèse, j'ai fourni une spécification formelle du langage Eliom, j'ai ajouté de nouvelles fonctionnalités telles un système de types et de modules et j'ai développé une nouvelle implémentation du langage. J'ai ainsi résolu la tension qui existait entre programmation sans-étages et modularité. Jusqu'à présent, Eliom est le seul langage sans-étages qui supporte à la fois compilation séparée et typage statique.

— Une première étape pour résoudre cette tension a été de réaliser que la programmation Web est très proche de la méta-programmation. Le code serveur émet des fragments de code client qui peuvent être recomposés à la manière des « quotation » de Lisp. Les méta-programmes sont généralement évalués partiellement : la première étape (ici, le code serveur) est exécutée d'abord et renvoie un second programme (le client) qui est ensuite exécuté. Dans le cadre de la programmation Web, nous voulons plutôt séparer statiquement le code client et le code serveur et les compiler séparément. J'ai fourni une sémantique évaluée et une sémantique compilée pour Eliom et prouvé une relation de simulation. Ceci permet aux programmeurs de raisonner informellement par évaluation partielle, tandis que le compilateur implémente la sémantique statique qui est plus complexe mais plus efficace.
— Après s'être assuré que le langage Eliom était compatible avec la compilation et le typage statique, une nouvelle question apparaît : comment passer à l'échelle ? Les programmes sont composés de nombreux modules à la fois client et serveur et nous souhaitons les recomposer arbitrairement tout en préservant le typage fort et l'efficacité du langage. Pour résoudre ce problème, j'ai introduit une nouvelle notion de modules sans-étages qui combinent les modules à la ML avec des annotations client/serveur. Ces modules s'intègrent dans l'écosystème OCaml de façon transparente et supportent l'abstraction et la compilation séparée.
— Pour démontrer la practicalité de ces nouvelles constructions, l'équipe Ocsigen et moi-même avons développé une galerie d'exemples et de bibliothèques avec Eliom, y compris des constructions récurrente de la programmation Web tels que les RPCs, le HTML, et la programmation fonctionnelle réactive [15]. Nous avons également montré comment les modules sans-étages permettent de structurer les applications Web [14].

Publications La partie théorique de ce travail a été présentée partiellement à APLAS 2016 [16]. Le système de module est en cours de soumission [18]. Les présentations de nouveaux paradigmes de programmation ont été présentées à IFL 2016 [15] (Prix Peter Landin) et WWW 2018 [14]. L'intégralité est contenue dans ma thèse [10].

2.1 Contributions au projet Ocsigen (A5 SO4 SM4 EM4 SDL5/DA2 CD3 MS3 TPM3)

Bien que ma recherche se concentre sur le langage Eliom, le projet Ocsigen est composé de nombreux éléments pour former un framework Web complet. En tant que membre de l'équipe Ocsigen, j'ai participé au développement de plusieurs fonctionnalités dans ces logiciels. De plus, je suis mainteneur de deux composants en particulier.

TyXML (A5 SO3 SM5 EM4 SDL5/DA4 CD4 MS4 TPM4) Tyxml (`https://github.com/ocsigen/tyxml`) est une bibliothèque qui permet d'écrire du HTML et du SVG dont la validité est garantie par le système de type. Cette idée n'est pas complètement nouvelle, mais Tyxml est la

première bibliothèque qui la rend facilement accessible. Depuis 2014, je suis le mainteneur et le développeur principal de Tyxml. Depuis, j'ai tâché de rendre Tyxml accessible à un plus large public en multipliant ces domaines d'applications et en ajoutant une extension de syntaxe pour utiliser la syntaxe du HTML directement. Tyxml est couramment utilisé dans la communauté OCaml et est distribué dans de large distributions Linux telles que Debian.

Une nouvelle implémentation d'Eliom (A2 SO4 SM2 EM2 SDL1/DA4 CD4 MS4 TPM4) Pour accompagner la formalisation d'Eliom, j'ai développé une nouvelle implémentation du langage en modifiant directement le compilateur OCaml [6] (représentant un patch de +2300-300 lignes d'OCaml). Bien que ce soit toujours un prototype, cette nouvelle implémentation suit fidèlement la formalisation, couvre l'ensemble des fonctionnalités de l'ancienne implémentation et introduit toutes les nouveautés développées dans le cadre de ma thèse. À terme, cette implémentation remplacera l'ancienne version. Elle a également été l'occasion de contributions à la fois à l'ancienne implémentation d'Eliom [2] et au compilateur OCaml [3].

3 Systèmes d'exploitation modulaires

Les systèmes d'exploitation modernes sont gros : ils contiennent de nombreux drivers et interfaces pour satisfaire les usages les plus courants. Dans le contexte des services Internet, cette taille augmente le poids du service et sa surface d'attaque. Un unikernel [8] est une application qui contient uniquement les pièces de système d'exploitation dont elle a besoin, telles une pile réseau ou un système de fichier. Ils sont ainsi spécialisés et optimisés en efficacité et en taille.

MirageOS [4] est un « système d'exploitation - bibliothèque » pour construire des unikernels d'une façon modulaire. L'idée principale de MirageOS est d'utiliser des modules ML pour décrire des systèmes de composants via des functors : des modules paramétrés par d'autres modules. Une pile réseau est ainsi implémentée en termes d'une interface réseau arbitraire et d'une implémentation d'IP. Alternativement, la pile réseau Unix peut être utilisée directement. L'implémentation concrète de tous ces composants importe peu à l'application finale, tant qu'une pile réseau existe. MirageOS contient actuellement une centaine de composants avec de nombreuses dépendances et plus de 5 cibles de compilation. Dans ces conditions, la flexibilité de conception des unikernels engendre une explosion des configurations possibles qui est difficile à contrôler manuellement.

Pour maîtriser cette complexité, nous avons développé un langage de configuration qui permet aux utilisateurs de décrire leurs choix de configuration comme une composition de ces modules d'un point de vue très haut niveau et générique. Ceci forme un langage proche des calculs à composants, mais avec une sémantique très différente. Les calculs à composants considèrent souvent des composants dynamiques dont il faut gérer le cycle de vie complet pendant l'exécution. Au contraire, les composants de MirageOS n'existent que pendant la configuration et sont intégralement dépliés et spécialisés pendant la compilation afin de générer une application d'une taille minimale. L'idée novatrice est de traiter la configuration comme un problème de méta-programmation : un fichier de configuration est un programme dont l'évaluation ne génère que le code utile pour la configuration nécessaire au déploiement de l'application.

Le langage de configuration et l'outil associé sont en production depuis 5 ans et utilisés par tous les utilisateurs de MirageOS, y compris des spécialistes des systèmes d'exploitation avec

2. Par exemple #387
3. Par exemple #1699, #1703 and #1704
4. https://mirage.io/

une connaissance limitée d'OCaml.

Publication Les travaux décrits dans le rapport de recherche [17] sont en cours de soumission pour publication dans une conférence.

Production logicielle : Functoria (A5 SO3 SM4 EM4 SDL5/DA4 CD4 MS3 TPM3) Je suis l'auteur principal (60% du design, 80% du code) et le mainteneur de langage de configuration et de l'outil associé, nommé Functoria (https://github.com/mirage/functoria/, 2700 lignes d'OCaml plus environ 500 lignes de patch dans l'écosystème MirageOS).

4 Types linéaires, ownership et inférence

Une large part de la programmation système consiste en la manipulation de ressources tels les descripteurs de fichiers, les connections réseaux, ou encore la mémoire allouée dynamiquement. Chacune de ces ressources est associée à un protocole qui décrit son usage correct. Par exemple, un descripteur est créé lors de l'ouverture d'un fichier. Si il a été ouvert en lecture, les opérations de lecture vont réussir mais pas les opérations d'écriture. Une fois le descripteur fermé, il ne peut être ni lu ni écrit. De nombreuses ressources suivent un protocole similaire. Violer ces protocoles cause de nombreux bugs tels que des erreurs de types, des data-races, ou encore des usages-après-désalocation et des fuites mémoires.

La plupart des langages statiquement typés garantissent la sûreté du typage et des accès mémoires, mais ne considèrent pas l'usage incorrect des ressources. Les types linéaires permettent de contrôler le partage des objets, mais forcent l'utilisateur à adopter un style de programmation purement fonctionnel peu adapté à la programmation bas niveau. La notion d'*ownership* [3], popularisée par le langage Rust, garantie un contrôle du partage similaire mais promeut un style impératif, souvent plus naturel pour la manipulation des ressources. Malgré ses avantages, l'ownership n'est disponible que dans des langages bas niveau tels Rust. Ces langages sont difficile d'approche pour les programmeurs habitués a des langages haut niveau et une gestion de la mémoire automatique par défaut.

Peter Thiemann et moi avons développé un langage qui combine types linéaires, ownership et fonctionnalités de la famille ML : fonctionnel, inférence de types complète et gestion automatique de la mémoire par défaut. Ce langage permet d'utiliser un style de programmation mixte, similaire au langage OCaml, avec un contrôle fin du partage et de l'aliasing quand l'utilisateur le souhaite. Un challenge particulier à été de conserver l'inférence complète en présence des types linéaires et d'ownership. En effet, l'inférence complète, qui évite à l'utilisateur le besoin d'annoter les types des fonctions, mène à un style de programmation bien plus souple et facile d'accès. Pour ce faire, nous avons étendu l'inférence par contraintes [9] avec un système de *kinds* et un nouvel algorithme de résolution de contraintes capable de traiter les types linéaires.

Hannes Saffrich à maintenant commencé une thèse dans la suite directe de mon travail. Nous collaborons également avec Guillaume Munch-Maccagnoni dans le but d'intégrer ces constructions dans le langage OCaml.

Publication Un article [13] est actuellement en soumission à PLDI.

Production logicielle : Affe (A2 SO4 SM2 EM2 SDL1/DA4 CD4 MS4 TPM4) J'ai implémenté un typecheckeur prototype pour notre langage (https://affe.netlify.com/). Ce prototype

contient l'intégralité des features décrites dans l'article, ainsi qu'une extension aux types de données algébriques.

5 Analyse de terminaison en C

L'un des grands axes de recherche des dernières années est l'analyse statiquement des programmes C pour vérifier autant de propriétés que possible, comme démontré par des projets tels Frama-C. Dans le contexte de logiciels temps réel ou de la cryptographie, la terminaison de programme est une première étape essentielle pour prouver qu'une procédure respecte les contraintes temporelles. Bien sûr, la terminaison est indécidable en général, mais peut être vérifiée dans certains cas, par exemple dans le cas des fonctions de rang *linéaires*. Une fonction de rang est une fonction positive décroissante pendant l'exécution du programme. Naturellement, si une telle fonction existe, le programme termine. Malheureusement, les méthodes précédentes pour inférer ces fonctions de rang ne passaient pas à l'échelle.

David Monniaux, Laure Gonnord et moi-même avons proposé une nouvelle méthode qui utilise la méthode CEGAR [5] pour inférer des fonctions de rang linéaires. Cette méthode procède par itération : si à un point donné la méthode trouve un contre-exemple, le programme ne termine pas. Si tous les cas sont traités, il termine. CEGAR n'avait précédemment pas été appliqué à la terminaison de programme. Nous avons prouvé la correction et la terminaison de notre algorithme. Afin de rendre notre analyse facile d'utilisation et applicable à des programmes C réalistes, nous avons implémenté notre algorithme comme un plug-in pour le compilateur LLVM. Ceci nous fourni de nombreux avantages (notamment une architecture de compilation très robuste), mais également une contrainte : les programmes sont alors en forme SSA et avec des points de contrôle multiples. Ceci est inhabituel pour les systèmes embarqués et les analyses associées font souvent la supposition que les programmes ont un point de contrôle unique. Nous avons étendu notre approche à ce nouveau contexte et l'avons implémenté. L'analyse résultante est suffisamment précise pour vérifier la plus grande partie des suites de test de la littérature et est plus rapide que les analyseurs concurrents par plusieurs ordres de grandeur. En particulier, notre analyse est capable de prouver instantanément la terminaison de programme relativement subtil tel le tri par tas.

Publication David Monniaux, Laure Gonnord (80%) et moi (20%) avons publié ce travail à PLDI 2015 [7].

Production logicielle : Termite (A2 SO4 SM3 EM1 SDL4/DA4 CD4 MS4 TPM4) Sous la supervision de Laure Gonnord, j'ai implémenté (80%) l'analyseur Termite (`https://termite-analyser.github.io/`, environ 2400 lignes d'OCaml) et je l'ai intégré comme plug-in dans la chaîne de compilation LLVM.

6 Outils pratiques pour les expressions régulières

Les expressions régulières sont l'un des outils les plus communs dans la boite à outil du programmeur. Étonnement, et bien que le domaine soit très exploré, il reste de nombreux problèmes concernant l'utilisation des expressions régulières en pratique. Je me suis en particulier intéressé à deux problèmes : comment tester un moteur d'expression régulière et comment extraire les données après le matching d'une façon efficace et bien typée.

6.1 Tester les moteurs d'expressions régulières

Étant donné la longue histoire des automates à états finis, le matching des expressions régulières est un domaine étonnement florissant et qui donne lieu à de nouvelles implémentations. Dans ces conditions, il est désirable de tester un moteur d'expressions régulières nouvellement implémenté. Ce n'est cependant pas si simple : la génération de test aléatoire n'est pas suffisante car nous voudrions à la fois des tests positifs et négatifs, sans utiliser un oracle externe. Pour ce faire, Peter Thiemann et moi avons proposé un algorithme qui génère le langage reconnu par des expressions régulières étendues avec l'opérateur de complément. Ceci permet de générer à la fois des tests positifs et négatifs. Nous avons fourni deux implémentations (en Haskell et OCaml) avec des performances au moins un ordre de grandeur plus rapide que l'état de l'art.

Publication Peter Thiemann et moi (50%) avons publié ces travaux à GPCE 2018 [12].

Production logicielle : Regenerate (A3 SO4 SM3 EM3 SDL5/DA4 CD4 MS4 TPM4) J'ai implémenté notre technique dans une bibliothèque (`https://regex-generate.github.io/regenerate/`, 1500 lignes d'OCaml). Cette bibliothèque est maintenant utilisée pour tester plusieurs moteurs d'expressions régulières, notamment `ocaml-re`, le moteur de regex le plus populaire en OCaml.

6.2 Expressions régulières typées

Les expressions régulières sont souvent utilisée pour extraire des données d'une chaîne de caractères via matching. La plupart des moteurs d'expression régulière renvoient simplement une liste de `string`, sans détail sur les branches et les répétitions contenues dans l'expression régulière. Certains algorithmes, plus précis, renvoient un arbre de syntaxe complet mais sont souvent incompatibles avec les fonctionnalités avancées des moteurs d'expressions régulières modernes. J'ai crée une méthode qui permet l'extraction bien typée et efficace des données matchées par n'importe quel moteur de regex. D'un point de vue théorique, cela nécessite une réinterpretation typée de l'algèbre de Kleene. D'un point de vue pratique, pour garantir l'efficacité et la correction, il faut utiliser la sémantique des transducteurs plutôt que celle habituelle des automates.

Publication J'ai publié ces travaux à PEPM 2019 [11].

Production logicielle : Tyre (A4 SO3 SM3 EM4 SDL5/DA4 CD4 MS4 TPM4) J'ai implémenté cette technique dans une bibliothèque nommée Tyre (`https://github.com/Drup/tyre`, 1000 lignes d'OCaml). Cette bibliothèque est maintenant utilisée par des utilisateurs, notamment pour le routage d'URL.

7 Implication dans l'écosystème OCaml

Je suis un contributeur prolifique à l'écosystème OCaml et j'ai contribué, via des patchs ou en tant que développeur principal, à un grand nombre de bibliothèques. Certaines de ces bibliothèques ont été détaillées précédemment. Je présente maintenant d'autres contributions qui ne sont pas directement liés à mes activités précédentes. Toutes ces bibliothèques sont distribuées en tant que logiciel libre sur des forges logiciels telles que Github ou via des gestionnaires de paquets.

Le compilateur OCaml (A5 SO3 SM4 EM4 SDL5/DA2 CD3 MS2 TPM1) Suite à mon travail sur Eliom présenté dans la Section 2, je suis particulièrement familier avec l'implémentation du compilateur OCaml. Depuis, j'ai fait plusieurs contributions dans deux domaines : rendre la syntaxe du langage plus flexible et nettoyer l'implémentation du compilateur pour le rendre plus facile à étendre. Grâce à mon expertise dans les systèmes de type et de module d'OCaml, je suis également l'un des relecteurs principaux pour les changements sur le typechecker.

La grande migration PPX (A5 SO1 SM4 EM4 SDL5/DA4 CD3 MS4 TPM3) Historiquement, OCaml a utilisé un système nommé « camlp4 » pour créer des extensions de syntaxes. En 2015, celui-ci a été remplacé par un nouveau système plus stable nommé « PPX ». J'ai été l'un des promoteurs de ce système via la migration de nombreuse extensions de syntaxe (`js_of_ocaml`, `lwt`, `tyxml`, ...) et la création de nouvelles (`ppx_tyre`, `sedlex`, ...). J'ai également fait plusieurs contributions au compilateur dans ce domaine et je suis l'un des mainteneurs de `ppx_tools`, un outil pour la création de PPXs.

ocp-browser (A5 SO2 SM3 EM3 SDL5/DA3 CD4 MS4 TPM3) `ocp-browser` est un outil pour explorer de façon interactive les modules disponibles sur le système du programmeur. Il utilise l'infrastructure mise à disposition par l'outil `ocp-index`, un outil d'auto-complétion pour éditeur de texte. En 2013, j'ai réimplémenté et étendu `ocp-browser` à partir d'un prototype de Louis Gesbert pour le rendre plus ergonomique et plus rapide. Depuis, cet outil fait partie de l'outillage standard en OCaml et est régulièrement recommandé aux programmeurs débutants.

Conclusion

Mes activités de recherche ont toutes un but commun : améliorer la pratique de la programmation en la rendant plus sûre, plus ergonomique ou plus efficace. La large majorité de mon travail a été implémentée, soit en tant que prototype soit comme outil utilisable dès à présent. De plus, je contribue à la fois à différent domaines spécifiques et à la théorie des langages de programmation. Ceci me donne une perspective unique pour créer de nouveaux langages spécialisés à des domaines variés qui restent accessible aux programmeurs. Dans ce cadre, je promeus l'utilisation des méthodes formelles pour concevoir des outils pratiques de programmation, adaptés aux spécificités de chaque domaine et basés sur des fondements théoriques solides.

Références

[1] Flow typechecker. URL https://flow.org/.

[2] Hack programming language. URL https://hacklang.org/.

[3] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 283–295. ACM, 2005. doi : 10.1145/1040305.1040329. URL https://doi.org/10.1145/1040305.1040329.

[4] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann,

and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi : 10.1007/978-3-319-17524-9_1. URL https://doi.org/10.1007/978-3-319-17524-9_1.

[5] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. doi : 10.1007/10722167_15. URL https://doi.org/10.1007/10722167_15.

[6] EliomLang. *EliomLang Sources*, 2018. URL https://github.com/ocsigen/eliomlang.

[7] Laure Gonnord, David Monniaux, and Gabriel Radanne. Synthesis of ranking functions using extremal counterexamples. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 608–618. ACM, 2015. doi : 10.1145/2737924.2737976. URL https://doi.org/10.1145/2737924.2737976.

[8] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels : library operating systems for the cloud. In Vivek Sarkar and Rastislav Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 461–472. ACM, 2013. doi : 10.1145/2451116.2451167. URL https://doi.org/10.1145/2451116.2451167.

[9] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1) :35–55, 1999.

[10] Gabriel Radanne. *Tierless Web Programming in ML*. PhD thesis, Paris Diderot, November 2017. URL https://www.irif.fr/~gradanne/papers/phdthesis.pdf.

[11] Gabriel Radanne. Typed parsing and unparsing for untyped regular expression engines. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*. ACM, 2019.

[12] Gabriel Radanne and Peter Thiemann. Regenerate : a language generator for extended regular expressions. In Eric Van Wyk and Tiark Rompf, editors, *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming : Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, pages 202–214. ACM, 2018. doi : 10.1145/3278122.3278133. URL https://doi.org/10.1145/3278122.3278133.

[13] Gabriel Radanne and Peter Thiemann. Kindly bent to free us. *CoRR*, abs/1908.09681, 2019. URL http://arxiv.org/abs/1908.09681.

[14] Gabriel Radanne and Jérôme Vouillon. Tierless web programming in the large. In Pierre-Antoine Champin, Fabien L. Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 681–689. ACM, 2018. doi : 10.1145/3184558.3185953. URL https://doi.org/10.1145/3184558.3185953.

[15] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. Eliom : tierless web programming from the ground up. In Tom Schrijvers, editor, *IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, pages 8 :1–8 :12. ACM, 2016. ISBN 978-1-4503-4767-9. doi : 10.1145/3064899.3064901. URL http://doi.acm.org/10.1145/3064899.3064901.

[16] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom : A core ML language for tierless web programming. In Atsushi Igarashi, editor, *APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 377–397, 2016. ISBN 978-3-319-47957-6. doi : 10.1007/978-3-319-47958-3_20. URL https://doi.org/10.1007/978-3-319-47958-3_20.

[17] Gabriel Radanne, Thomas Gazagnaire, Anil Madhavapeddy, Jeremy Yallop, Richard Mortier, Hannes Mehnert, Mindy Preston, and David J. Scott. Programming unikernels in the large via functor driven development. *CoRR*, abs/1905.02529, 2019. URL http://arxiv.org/abs/1905.02529.

[18] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom : A language for modular tierless web programming. *CoRR*, abs/1901.11411, 2019. URL http://arxiv.org/abs/1901.11411.