Travaux antérieurs

ARMAËL GUÉNEAU

1 VÉRIFICATION FORMELLE DE LA CORRECTION ET COMPLEXITÉ DE PROGRAMMES

Collaborateurs :
- François Pottier (Inria Paris, équipe Gallium)
- Arthur Charguéraud (Inria et Université de Strasbourg, CNRS, ICube)
- Jacques-Henri Jourdan (CNRS, LRI, Paris-Saclay)

Publications :
- Un article [GCP18] à la conférence internationale ESOP'18
- Un article [GJCP19a] à la conférence internationale ITP'19
- Une présentation [Gué18a] au workshop Coq @ FLOC 2018
- Mon manuscrit de thèse [Gué19b]

Un bon algorithme doit être correct. Pourtant, il est courant que des erreurs se glissent lors de son implémentation concrète. L'objectif des approches de *preuve de programme* est de vérifier rigoureusement la correction d'un algorithme, en établissant la preuve que son implémentation (exprimée dans un langage de programmation) n'échoue pas, termine, et renvoie le résultat attendu, correspondant à la spécification de l'algorithme (donnée en langage mathématique).

Cependant, qu'un algorithme renvoie le bon résultat n'est pas la seule propriété désirable : on s'attend également à ce qu'il soit efficace. Si de nombreux problèmes admettent une solution simple et inefficace, l'art de l'algorithmique s'intéresse à la conception d'algorithmes plus complexes mais efficaces. La raison d'être de tels algorithmes est alors typiquement leur bonne *complexité asymptotique*. Par ailleurs, éliminer les *bugs* de complexité (particulièrement difficiles à détecter par le test) est tout à fait souhaitable, car cela permet généralement d'améliorer les performances du programme concerné.

Pendant ma thèse [Gué19b], j'ai travaillé à la conception d'une méthodologie permettant de vérifier formellement la complexité asymptotique de programmes. en même temps que leur correction fonctionnelle. L'approche que j'ai développée prend la forme d'un outil de vérification de programmes OCaml impératifs embarqué dans l'*assistant de preuve* Coq, ce qui permet d'avoir une confiance élevée en la correction des preuves établies dans l'outil.

Les aspects distinctifs principaux de l'approche sont les suivants :
- Une logique de programmes expressive permettant de vérifier des programmes OCaml complexes (combinant structures de données mutables et fonctions d'ordre supérieur) et des bornes de complexité asymptotique amorties non triviales (poly-log, $O(\min(m^{1/2}, n^{2/3}) + n)$, $O(\alpha(n))$). La preuve des bornes de complexité peut dépendre d'invariants fonctionnels établis lors de la preuve de correction.
- L'approche met l'accent sur l'utilisation de la notation O due à Landau pour spécifier et structurer la preuve de spécifications de la complexité de programmes. La majorité des approches existantes raisonnent à plus bas niveau, en terme d'un compte explicite d'instructions effectuées par le programme. Comme détaillé dans mon article à ESOP'18, de telles spécifications sont peu modulaires et réutilisables, en plus de ne pas correspondre à la pratique de la littérature en algorithmique.

Chronologiquement, l'article ESOP'18 [GCP18] introduit la méthodologie et l'illustre sur des exemples simples ; une des bibliothèques Coq développée pour supporter l'approche est présentée au workshop Coq [Gué18a] ; et finalement l'article ITP'19 [GJCP19a] est principalement consacré à présenter une étude de cas significative. Dans ce qui suit, je détaille les ingrédients clefs de l'approche dans son ensemble.

1.1 Une logique de programme expressive : Logique de Séparation avec Crédits Temps Négatifs

Mon outil de vérification se basait initialement sur le cadre existant de la logique de séparation avec crédits temps [Atk11, PP11], utilisée notamment par Arthur Charguéraud et François Pottier pour vérifier la correction et complexité d'une implémentation de l'algorithme d'union-find de Tarjan [CP17].

Certaines difficultés liées à l'utilisation pratique de cette logique m'ont poussé à en concevoir une extension, appelée "logique de séparation avec crédits temps *possiblement négatifs*" (présentée dans l'article ITP'19). Cette logique de séparation étendue hérite des qualités de la logique avec crédits temps "standard" (elle permet d'exprimer des invariants de complexité amortie, et de raisonner sur des structures de données mutables). En outre, elle supporte des règles de raisonnement plus agréables, qui facilitent la conception de procédures d'inférence de complexité semi-automatiques, cruciales pour rendre l'effort de preuve raisonnable. Finalement, la logique de séparation avec crédits temps négatifs permet d'énoncer des spécifications plus élégantes pour les programmes dont le coût est exprimé en fonction de la taille du résultat et non seulement de l'entrée, ce qui contribue à l'expressivité de l'approche.

1.2 Spécifications de programmes avec bornes de complexité asymptotique

La logique de séparation avec crédits temps permet d'exprimer au niveau logique le coût d'un programme donné, en terme du nombre d'appels de fonctions et de tours de boucle effectués. Cette notion de coût — exprimée en terme du programme source — permet donc déjà de s'abstraire des détails concrets de la machine. Cependant, j'argumente dans mon article ESOP'18 qu'un tel décompte n'est pas satisfaisant en tant que *spécification*, car trop lié à des détails d'implémentation. Intuitivement, on espère que la complexité d'un programme de recherche dichotomique soit en $O(\log n)$, mais on n'est pas intéressé par savoir si celui-ci effectue exactement $3 \log n + 4$ ou $5 \log n + 12$ appels de fonctions.

À la place, j'ai proposé [GCP18] un style de spécifications de programme où le coût concret d'un programme est rendu abstrait, et où seule une borne asymptotique est exposée, à l'aide de la notation O. Dans le cadre de la vérification de programmes, j'ai présenté et formalisé des règles pour composer des spécifications avec O. Par exemple, raisonner sur la complexité d'une boucle `for` peut être fait en fonction d'une borne en O pour le corps de la boucle (plutôt que de maintenir un invariant de sommation très explicite). Ces règles permettent d'énoncer et de réutiliser modulairement des spécifications de complexité. En particulier, je me suis intéressé au cas de bornes en O avec plusieurs paramètres, celles-ci venant avec un certain nombre de difficultés additionnelles qui sont typiquement passées sous silence par les ouvrages classiques d'algorithmique.

1.3 Techniques de vérification pour des preuves de complexité robustes

Lors de la preuve de la borne de complexité d'un programme, il est également désirable de ne pas compter manuellement le nombre d'étapes de calcul effectuées par le programme, ce qui rendrait la preuve pénible et fragile. J'ai développé deux mécanismes permettant l'écriture de preuves formelles de complexité *robustes*. D'une part, un mécanisme d'inférence de coût guidé par l'utilisateur (introduit par l'article ESOP'18 puis développé dans mon manuscrit de thèse), permettant de synthétiser une expression de coût pour du code sans récursion. D'autre part, un mécanisme permettant, dans une preuve Coq, d'introduire des constantes symboliques afin de collecter des contraintes à propos de ces constantes ; l'utilisateur décidant à la fin de la preuve d'une valeur qui satisfait les contraintes pour les constantes introduites. Ce mécanisme est alors utilisé pour inférer des équations de récurrence pour la complexité de programmes récursifs, et a fait l'objet d'une présentation au workshop Coq à FLOC'18.

1.4 Application à la vérification d'un algorithme de l'état de l'art

Je me suis appliqué à démontrer l'utilisation pratique de la méthodologie proposée. L'étude de cas la plus significative fait l'objet de l'article ITP'19, et concerne la vérification de la correction et complexité asymptotique amortie d'un algorithme de l'état de l'art, pour la détection incrémentale de cycles dans un graphe. L'algorithme en question, dû à Bender, Fineman, Gilbert et Tarjan [BFGT16, §2] n'a été publié que récemment (en 2016). Si établir une implémentation concrète cet algorithme était déjà non-trivial, la difficulté principale reste l'analyse de sa complexité : il est loin d'être évident de se convaincre qu'il satisfait la complexité attendue, à savoir, que construire un graphe à n nœuds et m arêtes tout en vérifiant qu'il reste acyclique à chaque étape est $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

Le travail de vérification a requis une étude approfondie de l'algorithme et nécessité d'exhiber des invariants de complexité plus fins que ceux qui étaient fournis dans la preuve informelle de l'article originel. Cela nous a conduit, avec Jacques-Henri Jourdan, à découvrir une amélioration pour l'algorithme lui même. L'algorithme de Bender et coauteurs dépend d'un certain paramètre qui influence sa complexité ; Bender et ses coauteurs montrent alors comment calculer une valeur donnant la complexité attendue en fonction de la taille *finale* du graphe. Malheureusement, cette taille n'est en général pas connue à l'avance ! Nous avons donc proposé une version modifiée de l'algorithme où cette valeur est calculée en fonction de la configuration *actuelle* du graphe, levant cette limitation ; et j'ai pu prouver formellement sa correction et complexité.

1.5 Production logicielle

Ce travail est supporté par deux bibliothèques Coq, disponibles publiquement sous license *open-source*, et dont je suis l'auteur principal et le mainteneur : les bibliothèques `bigO` [Gué18b] et `procrastination` [Gué18c].

J'ai ajouté le support pour les crédits temps négatifs à l'outil `cfml` [Cha19a], et formalisé leur méta-théorie dans `cfml2` [Cha19b] ; `cfml` et `cfml2` sont développés et maintenus par Arthur Charguéraud et disponibles librement.

L'implémentation de l'algorithme de détection incrémentale de cycles et sa preuve formelle sont également disponibles sous license *open-source* [GJCP19b], et le code OCaml est disponible en tant que bibliothèque réutilisable. Celui-ci a été en particulier intégré [Gué19a] au logiciel `dune` [dun18], un gestionnaire de compilation (*build system*) populaire de l'écosystème OCaml [1]. Mon implémentation vérifiée a remplacé dans Dune une implémentation non vérifiée du même algorithme et qui comportait plusieurs bogues de complexité ; on a pu mesurer la nouvelle implémentation être jusqu'à 7 fois plus rapide sur un scénario concret.

2 PROPRIÉTÉS DE SÉCURITÉ HAUT NIVEAU POUR DU CODE BAS NIVEAU SUR DES MACHINES À CAPABILITÉS

Collaborateurs :

- Lars Birkedal, Aïna Linn Georges, Alix Timany, Alix Trieu (Aarhus University, Danemark)
- Dominique Devriese, Sander Huyghebaert (Vrije Universiteit Brussel, Belgique)
- Thomas Van Strydonck (KU Leuven, Belgique)

Publications :

- Un article [GGvS+21b] à paraître à la conférence internationale POPL'21
- Un article [GGvS+21a] à paraître à la conférence nationale JFLA'21
- Une présentation [GGTB21] à paraître au workshop PRISC'21

Un grand nombre d'attaques de sécurité reposent sur un attaquant corrompant la mémoire du programme cible (par exemple via un *buffer overflow*), permettant alors de compromettre le flot de contrôle du programme ou d'écraser des données privées.

1. Parmi les 3156 paquets disponibles via `opam` [opa], 1638 dépendent de `dune`.

Les architectures matérielles conventionnelles permettent difficilement de se protéger contre ce genre d'attaques : un CPU traditionnel ne fournit que des mécanismes de cloisonnement à gros grain, typiquement utilisés par le système d'exploitation à l'échelle d'un processus entier. Des stratégies de défense variées ont été étudiées [WLAG93, ABEL05, MWCG99, FSA97], mais celles-ci ne permettent que de défendre contre une certaine catégorie d'attaquants ou viennent au prix d'un coût en performance et complexité du système accrue.

Les machines à *capabilités* (*capability machines*) sont une forme d'architecture matérielle conçues pour fournir par nature une sécurité accrue, au niveau du matériel. Une machine à capabilités associe des méta-données à chaque mot machine (stocké dans les registres ou la mémoire), distinguant tout d'abord entre entiers et *capabilités*, les capabilités jouant alors le rôle de pointeurs permettant d'accéder à la mémoire. Les méta-données associées à chaque capabilité restreignent son accès à une certaine région mémoire, pour une certaine permission ; la machine garantissant l'intégrité de ces méta-données, qui ne peuvent être manipulées que selon des règles bien précises. L'idée n'est pas nouvelle [CKD94, DVH66, SSF99], mais a vu récemment un regain d'intérêt, en particulier grâce aux efforts du projet CHERI [che], suivi dernièrement par ARM avec la fabrication d'un prototype industriel (Morello [mor]).

Les capabilités tirent leur force des différentes manières dont elles peuvent être combinées, permettant alors de garantir avec assurance des propriétés de sécurité complexes. Il se pose alors, d'une part, la question de quelle utilisation concrète des capabilités faire pour garantir telle ou telle propriété ; et d'autre part, de comment raisonner sur les garanties formelles obtenues, pour s'assurer qu'elles correspondent à ce que l'on espère.

Le projet CHERI s'est en particulier intéressé à concevoir et spécifier formellement des machines à capabilités comme extensions d'architectures existantes [WNW+20] ; et à leur utilisation dans le cadre de systèmes d'exploitation réalistes [WWN+15, WNW+16]. Par ailleurs, Skorstengaard *et al.* [SDB18, SDB19] se sont intéressés à l'utilisation de capabilités pour implémenter une convention d'appel sûre garantissant des propriétés d'encapsulation de l'état local et de flot de contrôle bien parenthésé — et développent un cadre formel pour raisonner sur ces propriétés.

La ligne de recherche à laquelle j'ai pris part pendant mon post-doc s'intéresse aux moyens de garantir et raisonner sur des propriétés de sécurité "haut niveau" pour des programmes bas niveau, en utilisant une machine à capabilités. Un des ingrédients clef de notre approche est l'utilisation de la logique de séparation Iris [JKJ+18] embarquée dans Coq. Celle-ci fournit des principes de raisonnement puissants, tout en permettant de relier formellement ceux-ci à l'exécution concrète de programmes sur un modèle de machine à capabilités. Un autre ingrédient clef est l'utilisation de *relations logiques* afin de capturer de manière sémantique les garanties formelles fournies par la machine à capabilités. Une relation logique nous fournit alors typiquement un principe de raisonnement très général, que l'on peut déployer pour raisonner sur différents scénarios subtils, et donc le bon fonctionnement (par exemple en présence de code malicieux) repose sur une bonne utilisation des capabilités.

Avec l'aide de mes collaborateurs, la réalisation principale de mon post-doc a été la conception et la formalisation d'une nouvelle convention d'appel sûre basée sur les capabilités, publiée à POPL'21 [GGvS+21b]. Dans ce travail, nous proposons l'utilisation de capabilités dites *uninitialized* pour améliorer la convention d'appel de Skorstengaard *et al.* [SDB18] qui reposait sur un mécanisme inefficace d'effacement mémoire (on passe alors d'un coût d'effacement de la pile quadratique à un coût linéaire). On définit ensuite notre principe de raisonnement clef, une relation logique, qui permet de raisonner sur les garanties obtenues lors de l'utilisation de cette nouvelle convention d'appel. Les interactions entre les différents types de capabilités entrant en jeu sont subtiles, mais on montre qu'il est possible d'énoncer un modèle logique suffisamment abstrait, grâce à une combinaison nouvelle des mécanismes fournis par Iris. Celui-ci nous fournit des principes de raisonnement supplémentaires et nous permet en particulier de raisonner à bien plus haut niveau (et dans un cadre mécanisé) que dans la formalisation (sur papier) de Skorstengaard *et al.*

J'ai contribué en particulier à l'élaboration du modèle au coeur de la relation logique, et également à la vérification concrète des scénarios que l'on a considéré pour illustrer les propriétés d'encapsulation de l'état

local et de flot de contrôle bien parenthésé. Il s'agit du premier développement formel établissant des propriétés sémantiques pour des programmes s'exécutant sur une machine à capabilités et interagissant avec du code adversaire inconnu.

J'ai également pris l'initiative d'explorer l'application de notre méthodologie à un mode d'utilisation plus simple des capabilitées, à l'occasion d'un article publié à la conférence JFLA'21 [GGvS+21a]. L'objectif était tout d'abord pédagogique. Si l'on ne cherche qu'à garantir l'encapsulation de l'état local (laissant donc tomber les propriétés de bon parenthésage), alors on obtient un mode plus simple d'utilisation des capabilités, sur lequel il est significativement plus simple de raisonner. Cet article était donc l'occasion d'illustrer en détails les éléments clefs de notre méthodologie, sous un format plus pédagogique.

Une question restante, à laquelle je m'intéresse avec mes collègues, est de savoir s'il est possible d'implémenter, à l'aide de capabilités, une convention d'appel fournissant exactement les mêmes garanties que la notion d'appel de fonction d'un langage de haut niveau. En ce sens, notre présentation à venir au workshop PriSC'21 [GGTB21] fait l'inventaire des approches existantes, et présente un nouveau candidat susceptible d'apporter les propriétés manquantes à la convention d'appel de notre article POPL'21 (notamment, des propriétés de type "confidentialité").

Production logicielle

Le travail décrit ici a été entièrement formalisé en Coq, et est disponible publiquement sous license *open source*. Le développement `cerise-stack` [GGvSTb] formalise la sûreté pour une machine à capabilités avec capabilités *uninitialized*, ainsi que la convention d'appel supportant l'article POPL'21. Le développement `cerise` [GGvSTa] correspond au cas d'étude plus simple étudié dans l'article JFLA'21.

Ces développements supportent non seulement les articles auxquels ils correspondent, mais sont aussi un excellent support d'expérimentation : les outils qui y sont déployés (logique de programme, relation logique) permettent non seulement de raisonner sur les conventions d'appels mentionnées précédemment, mais aussi, plus généralement, de prouver la correction et la sûreté de programmes arbitraires, et d'expérimenter avec de nouveaux types de capabilités ou différentes notions de sûreté.

3 LOGIQUE DE SÉPARATION POUR LA VÉRIFICATION DE PROGRAMMES CAKEML IMPÉRATIFS AVEC ENTRÉES-SORTIES

Collaborateurs :

- Magnus Myreen (Chalmers University of Technology, Suède)
- Ramana Kumar, Michael Norrish (Data61, CSIRO, Australie)

Publications :

- Un article [GMKN17] à la conférence internationale ESOP'17

Le compilateur CakeML [TMK+19, ckm] est un compilateur vérifié formellement pour un très large sous-ensemble de ML. CakeML implémente de nombreuses optimisations : les performances du code généré sont comparables aux compilateurs ML de l'état de l'art. Le théorème de compilation de CakeML, formalisé dans l'assistant de preuve HOL4, fournit ainsi une preuve que le comportement du code machine produit par le compilateur correspond bien au comportement d'un programme avant compilation, éliminant mathématiquement la possibilité d'un bogue dans le compilateur.

Je suis l'auteur initial d'un outil [GMKN17] permettant de prouver la correction fonctionnelle de programmes CakeML, d'une manière pouvant ensuite se combiner avec le théorème de correction du compilateur CakeML, afin d'obtenir un théorème à propos du code machine obtenu après compilation.

Historiquement, le compilateur CakeML disposait initialement d'un mécanisme, implémenté en HOL4, permettant d'obtenir des programmes CakeML vérifiés à partir de fonctions de la logique — ce schéma de traduction ne

pouvant produire que des programmes CakeML purement fonctionnels. En Coq, l'outil CFML [Cha19a] fournissait un mécanisme de preuve de programmes ML impératifs en logique de séparation, mais n'était pas vérifié de bout en bout (produisait des axiomes), et était limité à un langage noyau.

Lors de ce travail, sur l'échelle de 5 mois, j'ai adapté l'approche suivie par CFML et basée sur la génération de *formules caractéristiques* [Cha11] en la portant en HOL4, en l'intégrant à l'écosystème CakeML, et en la généralisant de deux manières essentielles : d'une part, en formalisant de bout en bout la méthodologie, prouvant ainsi une fois pour toutes que les formules caractéristiques générées par l'outil sont correctes (au lieu de devoir les admettre en axiome) ; et d'autre part, en la généralisant à un langage plus riche, en ajoutant notamment le support pour les exceptions et entrées-sorties. Ces développements ont par la suite motivé le développement par Arthur Charguéraud d'une refonte de CFML, intitulée CFML2 [Cha19b], vérifiée de bout en bout en s'inspirant du travail que j'ai réalisé pour CakeML.

Finalement, en combinant cet outil de vérification avec le compilateur CakeML et le théorème de compilation associé, nous montrons que l'on peut obtenir des binaires vérifiés de bout en bout effectuant des tâches réalistes : entrées-sorties pour communiquer avec l'utilisateur, gestion d'erreurs, etc.

Production logicielle

Ce travail a été intégré au développement du compilateur CakeML lui même [ckf]. Il a été par la suite utilisé pour la vérification de la bibliothèque standard de CakeML et d'applications [FPK+18], et par ailleurs étendu ou utilisé d'une manière ou d'une autre dans plusieurs travaux ultérieurs [HAK+18, BTM+18, PRM19].

4 AUTRES TRAVAUX

4.1 Itération et transfert de ressources en Mezzo

Collaborateurs :

— François Pottier, Jonathan Protzenko (Inria Paris, équipe Gallium)

Publications :

— Une présentation [GPP13] au workshop HOPE'13

Le langage de programmation Mezzo [BPP16] utilise un système de types particulièrement expressif, où les types ont une interprétation en terme de ressources, permettant ainsi de contrôler la possession de données en mémoire. Il fait partie d'une famille de langages expérimentaux avec des systèmes de types dits "sous-structuraux" ayant inspiré le langage Rust, en plein essor dans l'industrie récemment.

Lors d'un stage à l'équipe Gallium, j'ai utilisé Mezzo afin d'étudier, dans ce cadre, les façons possibles de programmer des itérateurs. Ce travail a mené à une présentation au workshop HOPE'13 [GPP13]. La difficulté principale est d'exprimer les différents transferts de ressources (pour la structure sur laquelle on itère et ses éléments) entre le code d'itération et le client qui reçoit les éléments un par un. En particulier, dans le cas d'un itérateur donnant le contrôle de l'itération au client, il faut exprimer que la possession unique de la structure sur laquelle on itère est d'abord transférée du client à l'itérateur ; l'itérateur redonne alors au client la permission vers chaque élément individuel de la structure, au fil de l'itération ; finalement, le client récupère la possession sur la structure entière à la fin de l'itération.

Le code issu de ce travail a été intégré dans la bibliothèque standard de Mezzo.

4.2 Contributions au compilateur OCaml

Je suis un des 26 mainteneurs du compilateur OCaml. J'ai notamment contribué à nettoyer et restructurer la partie du compilateur dédiée à l'affichage des messages d'erreur et leur emplacement. Ceci m'a ensuite permis d'implémenter un format d'affichage plus agréable des erreurs émises par le compilateur, qui affiche la portion

du programme concernée en plus des indications de numéro de ligne et colonne. J'ai également contribué à l'intégration d'un certain nombre d'améliorations aux messages d'erreurs qui avaient été proposées par Arthur Charguéraud [Cha15] comme étant susceptibles d'améliorer l'expérience d'utilisateurs débutants.

AUTO-RÉFÉRENCES

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars : Formalizing asymptotic complexity claims via deductive program verification. In *ESOP*, 2018.

[GGTB21] Aïna Linn Georges, Armaël Guéneau, Alix Trieu, and Lars Birkedal. Towards complete stack safety for capability machines. In *PriSC*, 2021.

[GGvS+21a] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cap' ou pas cap' ? Preuve de programmes pour une machine à capacités en presence de code inconnu. In *JFLA*, 2021.

[GGvS+21b] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. In *POPL*, 2021.

[GGvSTa] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, and Alix Trieu. The `cerise` Coq development. https://github.com/logsem/cerise.

[GGvSTb] Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, and Alix Trieu. The `cerise-stack` Coq development. https://github.com/logsem/cerise-stack.

[GJCP19a] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal proof and analysis of an incremental cycle detection algorithm. In *Interactive Theorem Proving (ITP)*, 2019.

[GJCP19b] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. The verified incremental cycle detection library, March 2019. https://gitlab.inria.fr/agueneau/incremental-cycles.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *ESOP*, 2017.

[GPP13] Armaël Guéneau, François Pottier, and Jonathan Protzenko. The ins and outs of iteration in Mezzo. Higher-Order Programming and Effects (HOPE), 2013.

[Gué18a] Armaël Guéneau. Procrastination, a proof engineering technique. Talk proposal for the FLoC Coq Workshop, 2018.

[Gué18b] Armaël Guéneau. The `bigO` library, January 2018. https://gitlab.inria.fr/agueneau/coq-bigO.

[Gué18c] Armaël Guéneau. The `procrastination` library, January 2018. https://github.com/Armael/coq-procrastination.

[Gué19a] Armaël Guéneau. Dune pull request #1955, March 2019. https://github.com/ocaml/dune/pull/1955.

[Gué19b] Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs.* PhD thesis, Université de Paris, 2019.

RÉFÉRENCES

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. CCS '05. ACM, 2005.

[Atk11] Robert Atkey. Amortised resource analysis with separation logic. 7(2 :17), 2011.

[BFGT16] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, 12(2) :14 :1–14 :22, 2016.

[BPP16] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, 38(4) :14 :1–14 :94, 2016.

[BTM+18] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy : verified controller executables from verified cyber-physical system models. In *Programming Language Design and Implementation (PLDI)*, 2018.

[Cha11] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. pages 418–430, 2011.

[Cha15] Arthur Charguéraud. Improving Type Error Messages in OCaml. In *ML Family/OCaml Users and Developers workshops*, Electronic Proceedings in Theoretical Computer Science, Vancouver, Canada, 2015.

[Cha19a] Arthur Charguéraud. The `cfml` framework, March 2019. https://gitlab.inria.fr/charguer/cfml.

[Cha19b] Arthur Charguéraud. The `cfml2` framework, March 2019. https://gitlab.inria.fr/charguer/cfml2.

[che] The CHERI project. http://cheri-cpu.org.

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. ASPLOS VI, New York, NY, USA, 1994. ACM.

[ckf] The CF verification framework for CakeML. https://github.com/CakeML/cakeml/tree/master/characteristic.

[ckm] The CakeML project. https://cakeml.org.

[CP17] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. In *Journal of Automated Reasoning (JAR)*, 2017.

[dun18] Dune : A composable build system, 2018. https://dune.build/.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, page 143–155, March 1966.

[FPK+18] Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program verification in the presence of I/O : Semantics, verified library routines, and verified applications. In *Verified Software. Theories, Tools, and Experiments (VSTTE)*, 2018.

[FSA97] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems, Workshop on*, page 67. IEEE Computer Society, 1997.

[HAK+18] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In *Automated Reasoning - 9th International Joint Conference (IJCAR)*, 2018.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28 :e20, 2018.

[mor] The Morello project. http://morello-project.org.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 1999.

[opa] The ocaml package manager. http://opam.ocaml.org/.

[PP11] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. 2011.

[PRM19] Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. Characteristic formulae for liveness properties of non-terminating CakeML programs. In *Interactive Theorem Proving (ITP)*, 2019.

[SDB18] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities - provably safe stack and return pointer management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 475–501, 2018.

[SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Stktokens : Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros : A fast capability system. SOSP '99. ACM, 1999.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. SOSP '93. ACM, 1993.

[WNW+16] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast protection-domain crossing in the cheri capability-system architecture. *IEEE Micro*, 36(5) :38–49, 2016.

[WNW+20] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions : CHERI Instruction-Set Architecture (Version 8). Technical report, University of Cambridge, Computer Laboratory, 2020.

[WWN+15] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri : A hybrid capability-system architecture for scalable software compartmentalization. SP '15, USA, 2015. IEEE Computer Society.