Vérification formelle d'analyses de complexité

Proposition de sujet de stage de M2

12 décembre 2014

Directeur : François Pottier, directeur de recherche, projet Gallium, Inria Paris-Rocquencourt

Co-directeur : Arthur Charguéraud, chargé de recherche, projet Toccata, Inria Saclay.

Situation du sujet

Grâce aux progrès de la vérification de programme au cours de la dernière décennie, il est désormais possible de vérifier formellement la correction de programmes impératifs, y compris ceux dont les invariants sont particulièrement complexes.

Les techniques de vérification sont parfois appliquées pour prouver la terminaison du code, mais, à de rares exceptions près, elles ne sont jamais utilisées pour prouver que le code admet la complexité asymptotique espérée. Pourtant, un « bug de complexité » peut se révéler dans certains contextes tout aussi problématique qu'un bug de correction. En effet, si un algorithme n'a pas la complexité souhaitée, il peut, en pratique, échouer par manque de temps ou de mémoire.

La vérification formelle est principalement motivée par le fait qu'il est très facile de se tromper lorsqu'on fait des preuves « à la main ». Pour certains programmes dont la complexité est non triviale à justifier, comme par exemple ceux nécessitant des analyses amorties, le même argument s'applique. Il serait donc fortement souhaitable d'être capable de compléter les preuves de correction par des preuves de bornes de complexité. Les travaux existants autour de la formalisation de bornes de complexité se sont presque exclusivement concentrés sur des techniques d'analyse automatique, permettant de gérer les cas les plus simples mais incapables de formaliser les arguments de complexité amortie et/ou des arguments de complexité qui s'appuient sur une propriété de correction non triviale.

Pendant sa thèse [2], Arthur Charguéraud a conçu et implémenté un système de preuve de programmes, nommé CFML [1], pour un fragment substantiel du langage Caml. CFML, qui s'appuie sur l'assistant de preuves Coq, permet ainsi de réaliser des preuves de correction modulaires pour des programmes arbitrairement complexes. Il permet en particulier de raisonner sur les données mutables allouées dynamiquement, ainsi que sur les fonctions de première classe. CFML comporte un outil qui transforme un programme Caml en une formule logique, dite formule caractéristique, qui en décrit de façon exacte le comportement ; puis un jeu de définitions et de tactiques Coq permet à l'utilisateur d'exploiter cette formule caractéristique pour démontrer que le programme satisfait une certaine spécification, énoncé par l'utilisateur en Coq. Le système utilise la Logique de Séparation [9] et les prédicats abstraits pour modéliser l'organisation de la mémoire.

Sous-jacente à ce projet de stage est l'idée qu'avec une petite extension de CFML, on devrait pouvoir être capable de vérifier des analyses de complexité, y compris les plus complexes. L'approche envisagée consiste à ajouter à la Logique de Séparation des ressources appelées « crédits-temps ». Cette notion informelle remonte à Tarjan [10], et a été plus récemment introduite de façon formelle dans des systèmes de types [3, 6, 4, 8] ou de preuve [1]. Cependant, les crédits-temps n'ont à ce jour jamais été utilisés dans le contexte d'un système de preuve complet. Les crédits-temps sont des entités qui n'existent pas pendant l'exécution du programme, mais apparaissent dans le raisonnement à propos du programme. Un crédit-temps représente un droit à effectuer une étape élémentaire de calcul (par exemple, un appel de fonction). Les crédits-temps sont des permissions au sens de la

1. http:arthur.chargueraud.org/softs/cfml/

Logique de Séparation : ils peuvent être transmis, stockés, mais non dupliqués. Du fait que chaque appel de fonction consomme un crédit-temps, il résulte que le nombre de crédits consommés par un programme correspond (à une constante près) à sa complexité (amortie, et dans le cas le pire).

L'extension de CFML visée permettrait ainsi de raisonner simultanément à propos de la correction du programme et à propos de sa complexité. On notera que le premier de ces deux types de raisonnement est indispensable au second : pour établir la complexité d'un programme, il faut d'abord en établir certaines propriétés (par exemple, « tel arbre est équilibré », « la longueur de telle liste est au plus tant », etc.). Le défi, en ce qui concerne l'analyse de complexité formelle, vient du fait que cet exercice est relativement nouveau et du fait qu'il faudra comprendre comment reproduire formellement le raisonnement asymptotique informel à base de « grands O ». En principe, on peut modéliser les « grands O » en termes de quantification existentielle ; mais le jeu de l'alternation entre quantificateurs universels et existentiels est subtil, et il reste à déterminer comment mettre en œuvre cette approche en pratique.

Si l'extension avec les crédits-temps est un succès, on pourra dans un second temps s'intéresser à la notion de « débit temps ». Cette notion permet de raisonner à propos de la complexité des structures de données persistantes qui s'appuient sur l'évaluation paresseuse. Elle est décrite en détail dans les travaux d'Okasaki [7] et de Danielsson [4].

Objectifs

L'objectif de ce stage est d'étendre CFML avec des crédits-temps et des débits temps, et d'arriver à formaliser des analyses de complexité pour un certain nombre d'exemples de structures de données et d'algorithmes. Les exemples sélectionnés, dont la liste apparaît ci-dessous, couvrent la plupart des types de raisonnement à propos de la complexité. Notez que, pour chacun de ces exemples, il existe déjà une preuve en CFML de correction fonctionnelle. Ainsi, le travail de stage pourra être entièrement concentré sur les propriétés spécifiques à l'analyse de complexité, en réutilisant directement tous les invariants déjà établis pour la preuve de correction.

1. Formalisation d'analyses de complexité amortie basée sur les crédits-temps, parmi :
   - file fonctionnelle représentée par une paire de listes,
   - tableau redimensionnable (*vector*),
   - algorithme de Dijkstra utilisant une file de priorités sans opération *decrease-key*,
   - compteur binaire représenté sous forme d'une liste de booléens, et sa généralisation à la structure des tas binomiaux purement fonctionnels (binomial heap [7]),
   - séquence représentée à l'aide d'un arbre de *buffers* de taille fixe (*chunked sequence*).
2. Formalisation d'analyses de complexité amortie basées sur l'analyse par débits, parmi :
   - file fonctionnelle (physicists queue [7]),
   - file fonctionnelle à deux extrémités (banker's deque [7]),
   - file fonctionnelle avec accès aléatoire (implicit queue [7]),
   - séquence fonctionnelle avec accès aléatoire, concaténation et scission (*finger trees* [5]).

Notez qu'il sera tout à fait satisfaisant de couvrir seulement une petite partie des exemples ci-dessus. Pour réaliser ce travail, il faudra en particulier développer une représentation formelle de la notation « grand-O », ainsi qu'une tactique permettant d'automatiser le raisonnement sur les crédits-temps et les débits. Si le temps le permet, on pourra prolonger ce travail dans les directions suivantes :

3. (Optionnel.) Étudier la méta-théorie de CFML et établir (sur papier) la correction de l'extension de CFML avec crédits-temps.
4. (Optionnel.) Formaliser des analyses de complexité plus complexes, telles que l'algorithme de raffinement de partition de Hopcroft, ou bien la structure d'union-find de Tarjan.

Pré-requis

Des bases solides en algorithmique et en programmation, ainsi qu'une familiarité certaine avec le langage Objective Caml et l'assistant de preuve Coq, sont souhaitables.

Détails pratiques

Le stage se déroulera à l'Inria, sur le site de Rocquencourt, sous la direction de François Pottier et d'Arthur Charguéraud, de mars à juillet 2015 environ. Notez qu'Arthur Charguéraud est présent le mardi à Rocquencourt, et qu'il est également possible d'aller travailler avec lui à l'Inria Saclay.

Références

[1] Robert Atkey. Amortised resource analysis with separation logic. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.

[2] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification.* PhD thesis, Université Paris 7, 2010.

[3] Karl Crary and Stephanie Weirich. Resource bound certification. In *Principles of Programming Languages (POPL)*, pages 184–198, 2000.

[4] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *Principles of Programming Languages (POPL)*, 2008.

[5] Ralf Hinze and Ross Paterson. Finger trees : a simple general-purpose data structure. *J. Funct. Program*, 16(2) :197–217, 2006.

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

[7] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999.

[8] Alexandre Pilkiewicz and François Pottier. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*, 2011.

[9] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[10] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2) :306–318, 1985.