Modular and Certified Resource-Bound Analyses

Abstract

La recherche de bornes statiques sur les besoins en ressources d'un programme, ou *analyse de ressource* est justifiée par de nombreux besoins:

Économie On veut éviter de gâcher des ressources, et donc de l'argent, à causes de programmes gourmands et les infrastructures qu'ils engendrent.

Sûreté L'épuisement des ressources des systèmes embarqués et/ou critiques peut mettre des vies humaines en danger. Les conditions exceptionnelles d'épuisements de mémoire par exemple, ne sont que rarement bien géré – que le premier qui sait que son code et ses dépendances transitives ne plantent pas quand `malloc` échoue à allouer lance la première pierre.

Sécurité Des attaques, dites par *analyse de consommation*, prennent parti du comportement des implémentations pour mettre en défaut des algorithmes pourtant corrects, en déduisant de l'information du temps de CPU ou de la consommation mémoire d'un système cible.

Interaction Le développeur d'un logiciel bénéficie déjà d'outils lui permettant d'avoir un retour sur le code qu'il écrit: typage statique, analyseurs pour des bugs courants, etc. Une information fiable et rapide sur la consommation de son programme est utile à la programmation.

L'approche à l'analyse de ressource développée ici se distingue par sa *modularité* et sa *certification*. Modularité, car elle permet de séparer l'analyse selon les composants du programme cible, d'implémenter des analyses selon des axes orthogonaux, et de combiner les preuves automatisés et le raisonnement manuel. Certification, car les bornes inférées sont augmentées de certificats permettant de garantir leur validité. Ces certificats peuvent être mis à l'épreuve par l'assistant de preuve *Coq* sans transformation préalable, et couvrent tout aussi bien les déductions automatiques que les raisonnement manuels mentionnés plus haut.

Les travaux de cette thèse montrent qu'il est possible d'analyser l'usage de ressource à partir du code source tout en garantissant les bornes sur les ressources réelles (et seulement asymptomatiques). Enfin, notre implémentation permet des interactions riches entre l'analyseur et son utilisateur, via des mécanismes d'annotations et d'insertions de raisonnement manuels.

Introduction

Composants clés de l'analyse de ressources

Sémantiques de coûts Une *sémantique de coûts* est une description mathématique de l'environnement d'exécution d'un programme permettant d'en décrire le comportement et l'usage de ses ressources. Il est courant d'utiliser plusieurs d'entre elles dans une même analyse: une pour le *langage source* et l'autre pour le *langage cible*, par exemple. Le lien est fait par un *compilateur sensible* à l'usage des ressources.

Compilateur sensible Les compilateurs classiques permettent de faire le lien entre deux sémantiques, mais sans garanties formelles sur leur résultat. Il est courant de trouver des bugs dans des compilateurs industriels qui influent sur leurs capacité à rendre compte de la consommation du code cible. On cherche alors à produire des compilateurs *vérifiés formellement*, comme CompCert (de *ANSI C* vers *assembleur PowerPC*). Hélas, afin d'accommoder les contraintes fortes de vérification, d'importantes propriétés de consommation ne sont ni modélisées ni préservées dans le code cible. Pour pallier à cela, on peut soit restreindre l'analyse de ressource au code cible, soit développer des *compilateurs sensibles à la consommation de ressources* capables de garantir des traductions entre des sémantiques de coûts.

Digression: *Pourquoi vérifier les compilateurs?* On prends par exemple le projet *Rust* de Mozilla. Rust est un langage fonctionnel de haut niveau sans ramasse-miette, qui gère la mémoire grâce un système de type garantissant le non-aliasing des objet selon les principe RAII. La chaîne de compilation classique est Rust $\rightarrow$ C $\rightarrow$ assembleur. Mais ni CLANG ni GCC ne peuvent compiler correctement le code C émit par le compilateur Rust, qui respecte cependant scrupuleusement les standard.

Pourquoi ? Comme les pointeurs Rust sont tous soit en lecture seule, soit non-aliasés, ils sont tous marqués par `const` ou `restrict` dans le code C émit, alors que les compilateurs C ne sont rompus qu'au code utilisant ces annotation avec parcimonie. Ils produisent alors du code objet *défectueux*. A l'heure où ces lignes sont écrites, Rust est contraint de ne pas marquer ses pointeurs avec `restrict`, et doit donc renoncer aux gains de performances que son modèle mémoire devait garantir. Un comble. *Fin de digression.*

Logiques formelles Pour raisonner sur les sémantiques de coûts, il faut développer des *logiques formelles* basées sur des *principes des raisonnement pertinent* à la recherche de bornes de consommation, et ce à deux fins. Premièrement, ces logiques permettent au programmeur d'inférer manuellement des bornes et des annotations sur la consommation des programmes. Deuxièmement, elles guident l'implémentation et la vérification des compilateurs sensible au ressources.

Outillage (semi)-automatique Même s'il est théoriquement impossible d'entièrement automatiser l'analyse de ressource (c'est un problème indécidable, par réduction du problème de l'arrêt), il est possible, et bien sûr désirable, de créer des outils permettant de déterminer des bornes de consommation pour grand nombre de programme. Quand l'automatisation totale n'est pas possible, la logique formelle partagée entre l'outil et le programmeur permet à ce dernier de combler les failles du raisonnement automatique, et donc de maintenir les garanties sur les bornes obtenues.

Certificats Les compilateurs et les analyseurs statiques sont des programmes complexes dont la correction n'est jamais triviale (voir la digression plus haut). Les bugs étant monnaie courante, l'obtention de vraie garanties ne peut pas se faire à la tête de l'analyseur. Celui-ci doit générer un *certificat*, c'est-à-dire un objet qui prouve la validité de son résultat indépendamment de sa correction. L'outil tiers qui vérifie le certificat devient alors le point faible de la chaîne de confiance des garanties de consommation. Heureusement, des logiciels comme vérifiés formellement de fond en comble comme Coq peuvent remplir ce rôle.

TODO Logiques d'invariants

Les logiques d'invariants forment un corpus théorique utile aux preuves de corrections des algorithmes que l'on va implémenter durant le stage. Il serait peu être bon d'y jeter un oeil ?

Travaux connexes

Logiques de programmes

À la suite des travaux séminaux de Hoare, il s'est posé la question de comment ajouter l'entrée-sortie à la **Logique de Hoare**. Les méthodes classiques se basent sur des *variables auxiliaires* fraîches, présentes dans les pré- et post-conditions des programmes et simulant des arguments formels. Dans la logique d'invariants, on se passe de ces variables auxiliaires en introduisant les arguments via des quantificateurs du formalisme logique ambiant. On note qu'au moment de l'écriture de la thèse, cette nouvelle méthode est la seule garantissant la de la méthode tout en ayant une gestion simple (sic.) de la séquence et des variables auxiliaires.

Pour les preuves de **cohérence et complétude**. On note l'utilité d'utiliser les techniques inventées par Wright et Felleisen dans *A Syntactic Approach to Type Soundness*, notamment la préservation des types par la réduction.

Des **logiques de séparation quantitatives** ont été utilisées avec succès pour manipuler les fonctions de potentiels dans des logiques de programmes. On note

l'existence d'outils permettant de quantifier l'usage de ressource de bytecode JVM avec un certain succès, en se basant sur la *Vienna Development Method*. L'article est Aspinall & al, sous la direction de Hoffman: https://doi.org/10.1016/j.tcs.2007.09.003

Analyses automatique de ressources

Les travaux présentés dans cette thèses sont conçus dans la lignée de ceux de Hoffmann, jusqu'au système *RAML* et l'article *Towards Automatic Resource Bound Analysis for OCaml* (https://doi.org/10.1145/3093333.300984). On note trois avancées sur ces travaux:

- Nos fonctions de potentiels sont plus générales: alors que celles de RAML sont des polynômes multi-variables, les nôtres sont des combinaisons linéaires de fonctions de croissance très diverses. Ce formalisme est plus expressif et permet à l'analyse de conclure sur des inductions bien-fondées plus complexes.
- Nous permettons à l'utilisateur d'interagir avec l'analyse par le biais d'annotations. L'introduction de logiques de programmes quantitatives et complètes nous permet de garantir la correction des bornes établies même en présence d'annotations utilisateur pour guider la recherche.
- Enfin, notre système est le premier à générer effectivement des certificats pour les bornes inférées. La possibilité existait pour des travaux précédent, mais n'avais pas été effectivement implémentée.

Dans le monde impératif, de nombreux projets permettent d'obtenir des bornes et des invariants de boucles, souvent sur des modèles abstraits des langages ciblés, et parfois de manière ad-hoc. L'intérêt pour les méthodes de potentiels est montant dans le domaine. Quelque projets notables (*CoFloCo*, *Loopus* et *Rank* ne peuvent conclure sur les procédures récursives générales):

Outil	Auteurs	Technique
CoFloCo	Flores-Montoya	relations de coûts
KoAT	Brockschmidt & al	ré-écriture + terminaison
Loopus	Sinn, Zuleger & al	contraintes sur différences
PUBS	Albert & al	relations de coûts
Rank	Alias, Darte & al	ré-écriture + terminaison
SPEED	Gulwani	interprétation abstraite

Encore plus proche du métal, l'analyse de *Worst Case Resource Usage* commerciale est bien développée, mais ne permet pas d'inférer des bornes paramétriques, ou de vérifier la validité des annotations utilisateurs.

Génération d'invariants

L'analyse de ressource et la génération d'invariants forment un système d'étoiles binaires: l'obtention d'invariant permet de borner l'usage de ressource, et la génération d'invariants comprend souvent une réduction vers un solveur externe dans l'esprit et le sillage de l'analyse de ressource. Cette complémentarité est par exemple visible dans *SPEED*, mentionné plus haut. Plus récemment, Kincaid & al ont combiné les registres abstraits accumulateurs et une interprétation abstraite complexe pour inférer les invariant *polynomiaux*. Hélas, la mise en forme fermée de ces invariants est un problème moins bien cerné que dans le cas linéaire.

Dans tout les cas, il semble se dégager un schéma général en trois étapes de l'analyse de programme:

1. Choix d'un "moule" pertinent pour les invariants recherchés;
2. Extraction de contraintes linéraires dans cet espace de travail;
3. Résolution par programmation linéaire.

Les "moules" des contraintes sont remplis avec les coefficients extrais des structures syntaxiques du programme cible. La sémantique du langage cible engendre les contraintes entre ces coefficients. La complexité de l'analyse est sensible à la logique interne des contraintes: les contraintes linéaires-par-morceaux avec quantificateurs ne peuvent pas être résolues en temps polynomial (pour le moment). Hors des contraintes linéaires et assimilées, Müller-Olm & Seild obtiennent des invariants sous forme de polynômes à degré borné en utilisant un domaine abstrait de hauteur finie.

Compilation vérifiée sensible au ressources

CompCert est un compilateur vérifié d'un fragment d'*ANSI C* vers l'assembleur *PowerPC*, qui fournit des certificats garantissant la conservation de la sémantique du code C en entrée. Les garanties de consommation de temps et de mémoire doivent néanmoins toujours être obtenues par un autre logiciel, par raisonnement sur le code assembleur produit – sauf pour des invariants portant sur les états mémoires indépendamment les uns des autres. Nous avons étendu *CompCert* avec une gestion de la pile selon de standard PowerPC, notamment en prenant en compte sa finitude et en garantissant le non-débordement de pile des programmes compilés.

De récents travaux ont permis de créer *CompCertS*, une version de CompCert qui prend en compte la finitude de la mémoire. La sémantique des opération allouantes inclut une vérification, sous hypothèses faibles, de la quantité de RAM disponible. L'utilisateur peut alors obtenir des garanties simultanées sur la *correction opérationelle* de son programme et la *sûreté* de son utilisation des ressources. On contraste ceci avec notre approche, qui garantie la sûreté et accorde à l'utilisateur le confort mental de pouvoir supposer la mémoire suffisamment grande.

Enfin, Jost & al ont développés une sémantique quantitative pour leur langage *Hume* et un compilateur sensible à l'usage des ressources de Hume vers l'assembleur des micro-contrôleurs *Renesas M32C-85U*, mais sans certification de la conservation sémantique par le compilateur.