Tableaux et raisonnement arithmétique dans le compilateur Mezzo

Proposition de stage de recherche (niveau M2)

25 février 2013

Encadrant

François Pottier, directeur de recherche, INRIA Paris-Rocquencourt (francois.pottier@inria.fr)

Résumé

Mezzo est un langage de programmation conçu par Jonathan Protzenko et moi-même [4]. Il s'apparente au langage OCaml, mais propose un système de types et de « permissions » plus puissant, qui permet d'éviter certains des dangers associés à la concurrence et au partage accidentel de structures modifiables.

Pendant ce stage, je vous propose, en bref, d'étendre le langage *Mezzo* et son système de types pour leur ajouter une notion de « tableau », ou zone de mémoire homogène de longueur statiquement inconnue. On souhaite que le système de types soit suffisamment puissant pour :

1. garantir que les accès aux tableaux respectent bien les bornes, de façon à interdire les erreurs de type « buffer overflow » ;
2. permettre de découper par la pensée un tableau en plusieurs segments qui pourront être manipulés indépendamment (par exemple, par plusieurs threads distincts).

Pour atteindre ces objectifs, il est nécessaire que le programmeur puisse écrire au sein des types des formules arithmétiques (par exemple, $i < j$, ou encore $i + j < \mathit{length}(a)$) dont le compilateur vérifiera la validité, en s'appuyant sur un solveur SMT existant.

Situation du sujet

On peut situer *Mezzo* dans la famille des langages dits « fonctionnels », dont les principaux représentants sont Standard ML, OCaml, et Haskell. Il hérite en effet de ceux-ci plusieurs traits, parmi lesquels on trouve les fonctions en tant que valeurs de première classe, les types de données algébriques, le filtrage (« *pattern matching* »), et la gestion automatique de la mémoire. Par ailleurs, comme les langages cités plus haut et comme les langages impératifs et orientés objets traditionnels (C, Java, C#, etc.), *Mezzo* autorise la programmation « impérative », c'est-à-dire la modification en place d'objets alloués dans le tas.

Dans un langage de programmation traditionnel, les types reflètent la structure des objets, mais pas la façon dont ces objets sont partagés et modifiés. Ainsi, certaines erreurs de programmation ne sont pas détectées par le compilateur. Par exemple, si un composant A modifie le contenu d'un objet auquel un composant B a également accès, le fonctionnement de B peut être perturbé de façon inattendue ; ou encore, si deux « *threads* » T et U tentent de modifier au même moment le contenu d'un objet, on a une « *race condition* », et le comportement du programme peut devenir imprévisible.

Dans *Mezzo*, au contraire, la manière dont les objets sont partagés et modifiés est contrôlée par le compilateur. Celui-ci manipule non seulement des types, mais aussi une notion plus générale de « permission » à accéder à un objet. Ainsi, le compilateur sait « qui possède quoi » et interdit l'accès à un objet lorsque celui-ci n'est pas justifié par une permission.

L'intérêt potentiel de cette approche est double. D'une part, elle permet de rejeter certains programmes erronés qui seraient acceptés par un compilateur traditionnel. On peut ainsi gagner en sûreté. D'autre part, elle permet d'accepter certains programmes qui seraient interdits dans un langage de programmation traditionnel : par exemple, en *Mezzo*, le type d'un objet peut évoluer au cours du temps, sans que cela compromette la sûreté. On peut ainsi gagner en efficacité.

Le compilateur *Mezzo* existe aujourd'hui à l'état de prototype préliminaire, et (nous l'espérons) sera bientôt utilisable pour l'écriture de programmes et de librairies de petite taille.

Objectifs

Le programme de travail est le suivant.

1. Se familiariser avec *Mezzo* [4].
2. Ajouter à *Mezzo* une notion « simple » de tableau mutable, sans chercher à atteindre les deux objectifs énoncés plus haut. (Donc, si on accède à un tableau en dehors des bornes, cela sera détecté non pas pendant le typage, mais pendant l'exécution, comme en OCaml.)
3. Raffiner la notion précédente pour distinguer, au moment du typage, les tableaux immutables et les tableaux mutables. Certaines opérations (lecture, calcul de la longueur, etc.) seront possibles sur ces deux sortes de tableaux, mais l'opération d'écriture exigera un tableau mutable.
4. Écrire une librairie qui fournit un ensemble d'opérations courantes sur les tableaux mutables et immutables, analogue à la librairie « Array » d'OCaml.
5. Imaginer quels types précis on pourrait donner aux fonctions de cette librairie, et comment le typeur pourrait analyser de façon précise le code de cette librairie, afin de garantir que tous les accès aux tableaux respectent les bornes. On définira un langage de formules arithmétiques, comme $i + j < \mathit{length}(a)$, où i et j sont des variables du programme, de type entier, et a est une variable du programme, de type tableau. Ces formules logiques pourront être considérées comme des permissions et (par conséquent) pourront apparaître dans les types.
6. Étendre le typeur de *Mezzo* pour produire des obligations de preuve, c'est-à-dire des énoncés de théorèmes que l'on doit démontrer pour vérifier que les affirmations du programmeur sont correctes ; et, à l'aide d'une librairie comme Alt-Ergo [1], vérifier automatiquement ces théorèmes. Notons qu'il existe déjà des langages de programmation qui permettent l'écriture de formules logiques au sein des programmes, et qui en vérifient la validité en s'appuyant sur un solveur externe [3, 2]. On pourra s'inspirer de ces systèmes ou même (pourquoi pas) s'appuyer directement sur eux, en traduisant *Mezzo* vers l'un de ces langages.
7. Mettre à jour la librairie pour profiter de ce mécanisme nouveau. Écrire des algorithmes qui l'illustrent également (quicksort, heapsort, multiplication de matrices par blocs, etc.).
8. Jusqu'ici, on aura utilisé des permissions monolithiques, qui donnent accès à l'intégralité d'un tableau. Raffiner cette notion pour obtenir une notion de permission à accéder à un segment de tableau. Les opérations de découpage et de recollage de ces permissions donneront lieu à une obligation de preuve. Ces opérations pourront se faire soit de façon explicite (à la demande du programmeur), soit (peut-être) de façon implicite.
9. Écrire des algorithmes qui illustrent l'utilisation de ce mécanisme nouveau. En particulier, les algorithmes cités plus haut pourront recevoir des types plus précis, et on pourra développer des versions parallèles de ces algorithmes.

Si cette proposition de stage vous intéresse, n'hésitez pas à me contacter pour en savoir plus.

Pré-requis

Des bases solides en algorithmique, en programmation, et en logique sont indispensables. Une familiarité certaine avec le langage OCaml est très souhaitable.

Détails pratiques

Le stage se déroulera à l'INRIA, sur le site de Rocquencourt, sous la direction de François Pottier, de février ou mars à juillet 2013.

Références

[1] Sylvain Conchon and Evelyne Contejean. The Alt-Ergo automatic theorem prover, 2012. http://alt-ergo.lri.fr/.

[2] Jean-Christophe Filliâtre *et al.* Why3, 2012. http://why3.lri.fr/.

[3] Nikhil Swamy *et al.* F*, 2012. http://research.microsoft.com/en-us/projects/fstar/.

[4] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Manuscrit non publié, octobre 2012.