Optimisations pour l'exécution symbolique Wasm

Léo Andrès
OCamlPro
leo@ocamlpro.com

Ce stage offre à un·e étudiant·e curieux·se et motivé·e l'opportunité de participer à un projet de recherche combinant **programmation avancée** (en OCaml) et enjeux concrets d'analyse logicielle appliqués à des programmes WebAssembly, C et Rust.

Contexte

Le contexte de ce stage est celui de l'analyse statique de programmes WebAssembly (Wasm) [8] et plus précisément de l'outil **Owi** [1] développé chez OCamlPro en collaboration avec l'INESC-ID/Instituto Superior Técnico de l'Université de Lisbonne. Owi permet d'effectuer de l'**exécution symbolique** [10] de programmes Wasm. C'est-à-dire que l'utilisateur peut fournir un programme et obtenir automatiquement les valeurs d'entrées pour lesquelles le programme lève une erreur. Il est également possible d'analyser des programmes écrits dans d'autres langages tels que Rust ou C en les compilant vers Wasm, ce que l'outil est capable de faire directement. L'exécution symbolique dans Owi est particulière en ce qu'elle permet d'analyser des programmes inter-langages[1], qu'elle est effectuée en parallèle[2] et qu'aucune approximation n'y est faite.[3] Une description détaillée d'Owi est disponible dans plusieurs publications [2, 3, 9].

Sujet

Ce projet de recherche s'inscrit dans le prolongement des travaux menés sur Owi, son but est d'**optimiser les programmes Wasm afin d'améliorer l'efficacité de l'exécution symbolique**. Nous avons remarqué que les optimisations prévues pour une exécution concrète ont également un impact bénéfique sur l'exécution symbolique (notamment du fait de la réduction du nombre de branchements). En particulier, lors de l'exécution symbolique de programmes C, la compilation vers Wasm avec l'option `-O3` de `clang` permet de trouver 10% de bogues de plus que la compilation avec `-O0` sur les benchmarks issus de Test-Comp.

Dans cette optique, nous proposons d'intégrer **une passe d'optimisation dédiée au sein d'Owi**. Contrairement aux optimisations réalisées au niveau du langage source, celles appliquées directement sur le code Wasm sont indépendantes du langage d'origine et s'inscrivent mieux dans la perspective d'un outil générique capable de traiter différents langages. De plus, développer ces optimisations directement dans Owi présente plusieurs avantages par rapport à l'utilisation d'un outil externe comme Binaryen[4] : cela permet en particulier de concevoir des optimisations spécifiques à l'exécution symbolique, offrant un contrôle précis et adapté à ce cas d'utilisation.

Actuellement, Owi intègre un optimiseur naïf, accessible via la sous-commande `owi opt`. Cet optimiseur effectue des passes simples, comme l'élimination de code mort et la propagation de constantes, mais son implémentation reste naïve. Le stage vise à transformer cet optimiseur en un outil efficace, particulièrement adapté aux besoins de l'exécution symbolique.

Pour atteindre cet objectif, les étapes proposées sont les suivantes :

[1] Par exemple, un programme fait à la fois de code Rust et de code C.
[2] Grâce à OCaml 5.
[3] Si l'analyse termine, on a la garantie d'une absence de bogue.
[4] Un optimiseur de programmes Wasm.

- **implémentation d'optimisations simples directement sur la représentation sous forme de pile** de Wasm, en s'inspirant des techniques utilisées pour des langages similaires comme Lisp [13], Forth [4], ou STAL [14] ;
- conversion entre Wasm et une représentation intermédiaire dédiée à l'optimisation, telle que la **forme SSA**[5] [7], [15] (en commençant par explorer des structures comme le **graphe de flot de contrôle** [11]) ou **Sea of Nodes** [5] ;
- **développement d'optimisations avancées** sur cette représentation intermédiaire (voire pendant la transformation), incluant :
  - simplification et propagation de constantes ;
  - élimination de code mort et de lectures redondantes ;
  - simplifications algébriques ;
  - déplacement de code hors des boucles.
- **évaluation de l'impact des optimisations sur l'exécution symbolique**, en s'appuyant sur les benchmarks déjà intégrés à Owi ;
- comparaison des performances des programmes optimisés en termes de temps d'exécution, dans des contextes concrets (non symboliques) et en les comparant avec les optimisations de Binaryen et les programmes non optimisés.

Si le temps le permet, des extensions seront envisagées, telles que l'implémentation d'un moteur d'interprétation abstraite [6] pour Wasm ou l'intégration de ce dernier à la passe SSA, comme proposé par Lemerre [12].

📚 Pré-requis

- niveau M1/M2 informatique ou équivalent
- bonne maîtrise d'un langage fonctionnel (idéalement OCaml)
- notions d'algorithmique des graphes et de compilation

🏆 Compétences développées

- découverte approfondie de WebAssembly
- programmation OCaml avancée
- techniques d'exécution symbolique
- techniques d'optimisation modernes

📍 Lieu du stage

L'étudiant·e effectuera les travaux dans les locaux d'OCamlPro, à l'adresse suivante :

```
OCamlPro SAS
21 rue de Châtillon
75014 Paris
```

📅 Dates

Le stage pourra avoir lieu entre mars et septembre.

👩‍🏫 Encadrants

- Léo Andrès (Ph. D.), ingénieur R&D ;
- Pierre Chambart (Ph. D.), CTO et ingénieur R&D.

Le télétravail est possible, généralement deux jours par semaine.

📜 Bibliographie

[1] Léo Andrès, Pierre Chambart, Filipe Marques, Eric Patrizio, Arthur Carcano, et Zhicheng Hui. 2021. Owi. Consulté à l'adresse https://github.com/ocamlpro/owi

[2] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Femenin dos Santos, et Jean-Christophe Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming* 9, 2 (2024). https://doi.org/10.48550/arXiv.2412.06391

[5] Static Single Assignment.

[3] Léo Andrès. 2024. Exécution symbolique pour tous ou Compilation d'OCaml vers WebAssembly.

[4] Christopher Bailey. 1996. Optimisation Techniques for Stack Based Architectures.

[5] Cliff Click et Keith D Cooper. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17, 2 (1995), 181-196.

[6] Patrick Cousot et Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977. 238-252.

[7] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, et F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989. 25-35.

[8] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, et JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017. 185-200. https://doi.org/10.1145/3062341.3062363

[9] Zhicheng Hui et Léo Andrès. 2025. Cross-Language Symbolic Runtime Annotation Checking. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*, janvier 2025. Roiffé, France. Consulté à l'adresse https://inria.hal.science/hal-04798756

[10] James C. King. 1976. Symbolic Execution and Program Testing. *Communications of the ACM* (1976). https://doi.org/10.1145/360248.360252

[11] Daniel Lehmann, Michelle Thalakottur, Frank Tip, et Michael Pradel. 2023. That'sa Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Symposium on Software Testing and Analysis (ISSTA'23)*, 2023.

[12] Matthieu Lemerre. 2023. SSA translation is an abstract interpretation. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1895-1924.

[13] Larry M Masinter et L Peter Deutsch. 1980. Local optimization in a compiler for stack-based Lisp machines. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, 1980. 223-230.

[14] Greg Morrisett, Karl Crary, Neal Glew, et David Walker. 1998. Stack-based typed assembly language. In *International Workshop on Types in Compilation*, 1998. 28-52.

[15] Barry K Rosen, Mark N Wegman, et F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988. 12-27.