proposition de stage de M2 :
Garanties d'exécution de programmes OCaLustre et OCaml pour microcontrôleurs

Encadrants :
Emmanuel Chailloux (APR-LIP6-UPMC) - Emmanuel.Chailloux@lip6.fr
Steven Varoumas (APR-LIP6-UPMC) - steven.varoumas@lip6.fr
Lieu : LIP6 - Campus Jussieu - 4, place Jussieu 75005 Paris
Durée et dates : 5 à 6 mois à partir de février 2020
Rémunération : standard

Description : La programmation des architectures à base de microcontrôleur est toujours difficile tant par les ressources limitées accessibles que par les modèles de programmation proposés. L'intégration électronique poussée d'un microcontrôleur permet de diminuer la taille, la consommation électrique et le coût de ces circuits. La taille des programmes et la quantité de mémoire vive sont « faibles », le tas peut être de quelques kilo-octets seulement. Ils doivent communiquer directement avec les dispositifs d'entrées/sorties (capteurs, effecteurs, ...) via les pattes du circuit principal, et ne possèdent pas les périphériques classiques (souris, clavier, écran). La mise au point d'un programme devient plus difficile de par ce manque d'interaction classique. Néanmoins il est toujours possible de leur connecter un écran Lcd de quelques lignes d'affichage et des dispositifs de saisie spécifiques pour une application donnée. Pour pouvoir utiliser le plus précisément possible ces faibles ressources, les microcontrôleurs sont le plus souvent programmés en C ou en assembleur.

Plusieurs développements ont proposé des langages de plus haut niveau pour la programmation de microcontrôleurs, comme Picobit en Scheme, MicroPython pour Python ou OMicroB en OCaml [10]. OMicroB est un ensemble d'applications et de bibliothèques permettant de programmer plusieurs familles de microcontrôleurs en OCaml. Il permet de bénéficier de l'intégralité du langage OCaml en traduisant un programme en code-octet de la machine virtuelle OCaml en un programme C embarquant le dit code-octet, l'interprète de cette machine et la bibliothèque d'exécution. Le programme C résultat est portable sur Arduino, Micro :bit et PIC32. Les expériences avec OMicroB ont montré qu'il était beaucoup plus facile d'écrire et de mettre au point des programmes dans un langage de haut niveau comme OCaml qui intègre des traits fonctionnels, impératifs, modulaires et objets dans un cadre de typage statique avec une bibliothèque d'exécution gérant automatiquement la mémoire.

L'augmentation du nombre d'appareillages contrôlés par des micro-contrôleurs nécessite des garanties sur leur fonctionnement. Parmi celles-ci, le fait qu'il y ait assez de mémoire pour que le programme s'exécute est essentielle ainsi que le fait de garantir que les interactions seront bien prises en compte. Dans le premier cas, obtenir la quantité de mémoire nécessaire au bon fonctionnement du programme est un problème difficile et nécessite certaines restrictions. Par exemple, l'allocation dynamique est proscrite pour garantir l'absence de débordement de tas dans le développement d'applications critiques. Dans le second cas les modèles de programmation synchrone permettent de discrétiser le temps en une suite d'instants tout en garantissant le respect des échéances de chaque instant par des analyses de pire temps d'exécution (WCET), ce qui permet de garantir le comportement temps-réel de tels programmes.

OCaLustre est une extension synchrone à flots de données du langage OCaml destinée à la programmation de microcontrôleurs. Cette extension de langage permet d'offrir un modèle de programmation simple et peu gourmand pour le développement de comportements concurrents d'un système embarqué. Un programme OCalustre est transformé en programme OCaml qui peut ensuite être compilé et le byte-code passé à OMicroB pour son exécution sur microcontrôleur. Deux modèles de compilation sont offerts dont un optimisant l'utilisation mémoire en allouant puis modifiant l'état complet des flots scalaires et des registres mémoire. L'analyse du WCET [9] s'effectue sur le byte-code de cette version pour la partie synchrone. Elle est ensuite directement projetable sur l'architecture matérielle ciblée. Les appels aux fonctions hôte ne sont pas prises en compte, autant le traitement des fonctions non-allouantes semble classique, autant cela devient délicat pour les fonctions allouantes (pile et tas).

On cherche alors à apporter des garanties sur la consommation de ressources, principalement de la mémoire (pile et tas), en visant un compromis entre l'expressivité des algorithmes et des structures de données avec les capacités des analyses de consommation mémoire et de WCET. Dans le cadre d'OCalustre il est possible de donner une garantie au niveau de l'instant synchrone en cadençant le déclenchement du GC par l'extension synchrone. Pour cela il est nécessaire d'avoir une analyse de consommation mémoire pour la partie fonctionnelle/impérative d'OCaml.

Les analyses de consommation de ressources se sont tout d'abord intéressées au calcul du pire temps d'exécution. Les premiers travaux de calcul du pire temps d'exécution étaient basés sur la résolution de récurrence, ils s'adaptent à l'analyse de consommation mémoire. Depuis d'autres méthodes ont été développées et appliquées telles que les sized types, l'analyse amortie automatisée ou encore les invariants d'itérations. Parmi les travaux actuels, on trouve dans le cadre des langages fonctionnels les langages Hume [11] et RAML [5, 4] basés respectivement sur les sized types [6] et l'analyse amortie [8]. Et dans le cadre des langages objets impératifs, les projets Costa [1] et JConsume [2] basés respectivement sur la résolution de récurrences et les invariants d'itérations.

À l'exception de JConsume, les outils précédents ne tiennent pas compte d'un mécanisme particulier de libération de la mémoire. Il est difficile en présence de garbage collector de savoir quelles zones mémoires sont libérables statiquement. Pour résoudre cette difficulté, une possibilité est d'ajouter un mécanisme offrant plus d'informations statiques comme les régions. À l'origine, les régions sont ordonnées en pile et permettent d'introduire une notion de portée lexicale aux valeurs allouées dans le tas. La précision de ce mécanisme repose en partie sur le nombre de régions créées à l'exécution. Plusieurs travaux ont montré que cette discipline de pile rendait difficile une gestion fine de la mémoire. De récentes avancées [3] ont montré qu'il était possible de se passer de cette pile en adoptant un mécanisme de capacités pour la gestion des régions. À la différence des régions usuelles, celles-ci servent avant tout à l'analyse statique du calcul de borne supérieure d'occupation mémoire à l'exécution. Elles n'apparaissent donc pas forcément au niveau de l'environnement d'exécution. Ainsi [7] propose le développement conjoint d'un langage fonctionnel impératif typé statiquement muni sur un mécanisme de régions et d'une analyse conçue pour déterminer la quantité maximale de mémoire utilisée lors de l'exécution d'un programme.

Dans le cadre de la programmation fiable de microcontrôleurs, et autres matériels, on cherche à définir pour un langage de haut niveau (au moins fonctionnel/impératif, statiquement typé à la OCaml) sur lequel une analyse statique de consommation mémoire garantit que l'espace mémoire est suffisant durant l'exécution. Un premier travail est donc de faire l'état de l'art des techniques d'analyse de consommation mémoire tout d'abord sans tenir compte du gestionnaire automatique de mémoire (GC), et dans un second temps avec GC. Au moins deux voies sont alors possibles et intéressantes à explorer.

- La première est de ne pas autoriser le déclenchement de GC lors d'un appel de fonction classique OCaml à partir d'un nœud OCaLustre, et pour cela connaître la consommation mémoire via un système à la RaML. C'est l'instant synchrone qui pourra déclencher le GC et en tenir compte lors de calcul de WCET.
- La seconde est de construire une analyse de consommation mémoire en présence d'un GC, au moins pour un noyau fonctionnel/impératif à la OCaml, quitte à ajouter des informations l'aidant c'est-à-dire en annotant les programmes. Cette analyse pourra se limiter à des fonctions travaillant sur des structures dynamiques simples (listes, arbres, références, dots) pour lesquelles il sera possible de connaître la consommation mémoire. Ce bornage mémoire permettra de garantir que le programme ne dépassera pas la capacité du tas en présence de GC. Il sera utilisable pour des programmes OCaml tournant sur des processeurs classiques permettant ainsi d'analyser statiquement leur consommation mémoire.

Dans les deux cas la partie calcul du langage hôte en tenant compte des temps de GC (dépendant de préférence du nombre de valeurs restant vivantes) est à intégrer au calcul du WCET.

Au final, on obtient une programmation sûre pour la partie synchrone, mais aussi pour la partie algorithmique avec appel de fonctions OCaml, qui garantit le bon déroulement du programme tant en taille mémoire, avec réutilisation de celle-ci, que pour le non-chevauchement d'instants synchrones. Cela au niveau du code-octet, et donc facilement transposable sur des architectures de microcontrôleurs simples.

Références

[1] Elvira Albert, Puri Arenas, Samir Genaim, et Germán Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Autom. Reasoning*, 46(2) :161–203, 2011.

[2] Víctor A. Braberman, Diego Garbervetsky, Samuel Hym, et Sergio Yovine. Summary-based inference of quantitative bounds of live heap objects. *Sci. Comput. Program.*, 92 :56–84, 2014.

[3] Matthew Fluet, Greg Morrisett, et Amal J. Ahmed. Linear Regions Are All You Need. In *Proceedings of the 15th European Symposium on Programming*, ESOP 06, pp. 7–21, 2006.

[4] Jan Hoffmann, Ankush Das, et Shu-Chun Weng. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th Symposium on Principles of Programming Languages*, POPL'17, pp. 359–373. ACM, 2017.

[5] Martin Hofmann et Steffen Jost. Type-Based Amortised Heap-Space Analysis. In *Proceedings of the 15th European Symposium on Programming*, ESOP 06, pp. 22–37, 2006.

[6] John Hughes, Lars Pareto, et Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd Symposium on Principles of Programming Languages, POPL 96*, pp. 410–423. ACM, 1996.

[7] Salvucci J. et Chailloux E. Memory Consumption Analysis for a Functional and Imperative Language. In *RAC 2016 - Resource Aware Computing*, volume 330 of *Electronic Notes in Theoretical Computer Science*, pp. 27 – 46, April 2016.

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

[9] Steven Varoumas et Tristan Crolard. WCET of ocaml bytecode on microcontrollers : An automated method and its formalisation. In *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *OpenAccess Series in Informatics (OASICs)*, pp. 5 :1–5 :12. Schloss Dagstuhl, July 2019.

[10] Steven Varoumas, Benoît Vaugon, et Emmanuel Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project. In *Proceedings of the 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, 2018.

[11] Pedro Vasconcelos. *Space Cost Analysis Using Sized Types*. PhD thesis, University of St Andrews, 2008.