ICLP 2006 DC Research Summary

Static Memory Management for Logic Programming Languages

Quan Phan

Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B-3001 Heverlee, Belgium,
`{quan.phan}@cs.kuleuven.be`

Introduction. Logic programming (LP) languages aim to free programmers from procedural details such as memory management tasks. One classical, automatic memory management technique in logic programming is to use a heap memory for all the structured terms and rely on backtracking and on a runtime garbage collector to reclaim memory. While efficient implementation of garbage collectors for LP languages can reuse more than 90% heap space, they introduce performance penalties to the execution of a program because the collectors need to temporarily stop the main program to do their job.

Background Literature. To remedy this shortcoming there has been a lot of research on compile-time memory management techniques, which automatically enhance programs with instructions to reuse memory. This static method generally follows two approaches: region-based memory management (RBMM) and compile-time garbage collection (CTGC). The basic idea of RBMM is to divide the heap memory into different regions. The dynamically created terms and their subterms have to be distributed over the regions in such a way that at a certain point in the execution of the program all terms in a region are dead and the region can be removed. CTGC detects when allocated memory cells are no longer used and instructs the program to reuse those cells to construct new terms, reducing memory consumption and in some cases achieving faster code. RBMM has long been a topic of intensive research for functional programming languages [7, 1, 4] and more recently also for procedural languages [3, 2]. For LP languages, there has been only one attempt to make RBMM work for Prolog [5]. The idea of CTGC has been used to reuse memory cells locally in the procedures of the LP language Mercury [6].

Goal of the Research. RBMM achieves competitive memory consumption for many programs due to timely removal of dead regions and memory management operations are time bound. CTGC can exploit many reuse opportunities. Taking those advantages to have a system in which reusable memory can be reused and non-reusable dead memory can be deallocated timely is the motivation for our current research. The ultimate research goal is to investigate the possibility and practicality of a hybrid static memory management technique, which combines RBMM and CTGC.

Current Status. Developing an algorithm that combines the nontrivial program analyses of CTGC in [6] and the type-based region inference in [5] has not been straightforward. Its main difficulty comes from the fact that those analyses

were developed for different programming languages. The CTGC system is for Mercury, while the RBMM system is originally designed for a first-order subset of ML and extended to work in XSB Prolog. More recently an analysis and transformation algorithm for RBMM, which relies on region points-to graphs and region lifetime analysis, has been reported for Java [2]. We found several similarities in terms of design and the basic concepts used between the algorithm and the analyses in CTGC. This makes it feasible to reformulate the algorithm to work for Mercury and to integrate it later on with CTGC analyses. In the initial step towards the research goal, we have developed an extensible RBMM analysis and transformation algorithm in the context of deterministic Mercury programs. Currently we are working on improving the precision of the algorithm and also on integrating the algorithm into a working Mercury compiler and extending the Mercury runtime to support regions. After having both RBMM and CTGC systems in the context of Mercury the idea of integrating them will have the necessary background to be developed.

Open Issues and Expected Achievements. There are several issues that need further investigation, such as supporting modular region analysis, both the theory and practice of the operation of CTGC when memory is organised in terms of regions, and the interaction of RBMM, CTGC and runtime garbage collection. The combination of RBMM and CTGC may lead to the “reuse inside regions” technique, which is useful when in a program there are procedures containing cells that die but cannot be reused locally by CTGC. We expect that dead, reusable cells can be allocated in separate regions from the regions of dead, non-reusable cells, which can be reclaimed timely by RBMM when they cease to be live.

References

- [1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1995.
- [2] S. Cherem and R. Rugina. Region analysis and Transformation for Java. In *Proceedings of the 4th international symposium on Memory management*, pages 85–96. ACM Press., October 2004.
- [3] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.*, pages 282–293, 2002.
- [4] F. Henglein, H Makhholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming.*, pages 175–186, 2001.
- [5] H. Makhholm and K. Sagonas. On enabling the WAM with region support. In *Proceedings of the 18th International Conference on Logic Programming*, 2002.
- [6] Mazur N. *Compile-time garbage collection for the declarative language Mercury*. PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, May 2004.
- [7] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation.*, 132(2):109–176, February 1997.