[Sas94] A.V.S. Sastry. *Efficient Array Update Analysis of Strict Functional Languages.* PhD thesis, Computer and Information Science, University of Oregon, 1994.

[SC94] A.V.S. Sastry and William Clinger. Parallel destructive updating in strict functional languages. In *ACM Conference on Lisp and Functional Programming*, Lisp Pointers 7(3), pages 263–272, 1994.

[SCA93] A.V.S. Sastry, William Clinger, and Zena Ariola. Order-of-evaluation analysis for destructive updates in strict functional languages with flat aggregates. In *Conference on Functional Programming Languages and Computer Architecture*, pages 266–275, 1993.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193. IEEE, April 1998.

# Optimizing Memory Usage in Higher-Order Programming Languages: Theoretical and Experimental Studies[36]

Mitchell Wand and William D. Clinger
Northeastern University
Boston, MA 02115
({wand,will}@ccs.neu.edu)
www.ccs.neu.edu/home/{wand,will}

This project is concerned with techniques for optimizing the use of memory in higher-order programming languages, using a combination of theoretical and experimental techniques. Specific activities include: proof of correctness of destructive update insertion and their associated transformations, analysis and experimental investigation of new algorithms for generational garbage collection, and development of theoretical methods for proving the correctness of storage optimizations in compilers.

Destructive update insertion is a compiler optimization for functional languages that transforms functional array updates into imperative updates whenever the analysis determines that the old value of the array being updated is never needed following the update. Clinger and his student Sastry developed the first interprocedural destructive update analysis that runs in polynomial time [SCA93, SC94, Sas94]. For typical programs, its performance is nearly linear. This optimization is also more effective than previous analyses because it has the ability to choose a good order of evaluation. This analysis is a significant advance over the well-known optimizations used in the Sisal compiler [Can92]. Wand and Clinger have proven the correctness of this transformation [WC98]. This transformation is difficult to prove correct because it radically changes the storage behavior of the program. The proof uses a formulation of the analysis via set constraints and several levels of operational semantics at different levels of detail.

In the area of garbage collection, Clinger has described a new algorithm for generational garbage collection and used this algorithm to show that generational garbage collection can outperform nongenerational collection even for the radioactive decay model of object lifetimes, which had been conjectured to be a theoretical worst case for generational garbage collection [CH97]. He and his student Lars Hansen have demonstrated the viability of this new algorithm by implementing it in the Larceny runtime system for Scheme, where the new algorithm usually performs at least as well as Larceny's standard generational collector.

Another topic we have studied is the efficiency of allocating continuations on a heap instead of a stack. This has been a controversial topic, which has turned on disputed estimates of the indirect costs for various implementation strategies. Clinger has used Twobit and Larceny, our optimizing Scheme compiler and runtime system, to attach instrumentation to a set of benchmarks, and used these measurements to compute upper bounds for the disputed indirect costs. He also discovered a significant but previously unknown indirect cost of heap allocation when continuations are used to implement coroutines or concurrent threads [CHO99].

Clinger has formalized Scheme's requirement for proper tail recursion as a safety property that places an asymptotic bound on the amount of space that is used by implementations of Scheme. He also formalized several related safe-for-space-efficiency properties, and proved that they form a hierarchy of concrete complexity classes. This work clarified the distinction between tail call optimization and proper tail recursion, and explained how proper tail recursion depends upon garbage collection [Cli98].

At the end of 1998 Clinger and his student Lars Hansen made Twobit and Larceny available via the Internet. We believe that this is the first public release of source code for an implementation of Scheme that combines an optimizing native-code compiler with a generational garbage collector.

## References

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, May 1991.

[SW97] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, pages 48–86, January 1997. Original version appeared in Proceedings 21st Annual ACM Symposium on Principles of Programming Languages, 1994.

[Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):365–387, July 1993. preliminary version appeared in *Conf. Rec. 20th ACM Symp. on Principles of Prog. Lang.* (1993), 137–143.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proc. IEEE Conf. on Computer Languages '98*, pages 184–193. IEEE, April 1998.

[WS97] Mitchell Wand and Gregory T. Sullivan. Denotational semantics using an operationally-based term model. In *Proceedings 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 386–399, 1997.

[WS99] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[36]This work is supported by the National Science Foundation under grant CCR-9804115.