Orbit: An Optimizing Compiler for Scheme

David Kranz
Curl Corporation
kranz@curl.com

Richard Kelsey
kelsey3@s48.org

Jonathan Rees
Millennium Pharmaceuticals, Inc.
jar19@mumble.net

Paul Hudak
Yale University
paul.hudak@yale.edu

James Philbin
Data Management Solutions
james@philbin.name

Norman Adams
Consultant
nadams@acm.org

ABSTRACT

Orbit was an optimizing compiler for T, a dialect of Scheme. Its aggressive use of CPS conversion, novel closure representations, and efficient code generation strategies made it the best compiler for a Scheme dialect at the time and for many years to come. The design of T and Orbit directly spawned six PhD theses and one Masters thesis, and influenced many other projects as well, including SML of New Jersey.

1. SETTING

When this paper was written in 1986, Scheme [6] was a twinkle in the eye of a few language purists, Common Lisp did not exist, ML was confined mostly to AI labs in Europe, Haskell did not exist, C++ had not yet become more popular than C, and Pascal was still viewed as a decent implementation language. Wow! Much has happened since then in language design and implementation.

All Lisp dialects before Scheme used *dynamic scoping* for the semantics of procedures and functions, and thus all implementations used some kind of dynamic binding mechanism for variables. A lot of effort was put into doing this well, so people felt pretty good about it, as in, how could you do much better, and why would you want anything but dynamic scoping? Then along came Scheme.

2. NEW IDEAS

One of the new things about Scheme was its use of *lexical scoping* (hardly new in language design – cf. Algol [4] – but certainly new to the Lisp community), and the emphasis on functions and procedures – i.e. *lambda* – as the ultimate abtraction mechanism (as reflected in a couple of MIT AI memos by Guy Steele [16, 17]). Another innovative feature was its (admittedly controversial) notion of a *first-class continuation*. Of course, Scheme also inherited many of Lisp's other features, most notably runtime typing, a rich library, and automatic storage management.

In any case, there was tangible hope that, if one could only implement Scheme well, it would be the language of choice for most applications, and in particular would be a better choice than C. All that we needed was a good compiler! Then along came Rabbit.

3. NEW HOPE

Guy Steele's Master's thesis [18] was about a prototype compiler called *Rabbit* that used CPS (continuation passing style) conversion to linearize programs, much like a continuation semantics. With respect to procedures, the nifty idea was that procedures did not return – they just jumped to a continuation that was handed to them. The connection between continuations and return pointers in a compiler was what made compilation based on this idea feasible. Furthermore, lexical scoping was implemented using closures – ala Algol – so that there was hope that lexical scoping – surely the preferred choice for the language connoisseur – might really work.

Still, much skepticism remained. A full implementation of Steele's proof of concept was needed to silence the remaining skeptics. Then along came Orbit.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

4. SALVATION

T [13, 15] is a Scheme dialect with several innovative features (more on this later). Orbit was an implementation of T based on the ideas in Rabbit and ideas from the authors' experience with other Lisp implementations. Orbit was noteworthy in its aggressive use of CPS conversion, the use of a CPS intermediate form throughout the compiler, and its clever use of data representations, in particular that for closures, to make function calls very efficient. It also had a novel garbage collector, which was actually written in a subset of T that was known to produce straight-line code (see later discussion about Pre-Scheme). Finally, compiled T programs had the interesting property that an interrupt could occur between any two lines of user code (see later discussion about operating systems).

The resulting compiler was the best implementation of a Scheme dialect at that time, and remained so for many years to come. It was competitive with C on chosen benchmarks (benchmarking was not a science), and outperformed Apollo's Pascal implementation.

5. IMPACT

Of course, ultimately C won the language war. One could say the deck was stacked against T/Scheme from the beginning, for purely technical reasons: the reliance on GC, the funny syntax, the emphasis on first-class procedures, and so on. The truth is, there are other reasons why one language wins out over others, but to pursue that here would be a digression.

Nevertheless, the impact of T/Orbit was significant. For starters, there were no fewer than six PhD dissertations and one Master's thesis that grew directly out of the effort:

1. David Kranz distilled the code generation ideas in Orbit and wrote a dissertation [8] at Yale in 1988 whose title was the same as the paper for which we are writing a retrospective. His work involved rather sophisticated data analysis and register allocation techniques.

2. Richard Kelsey's dissertation [7] at Yale in 1989 took the

idea of CPS transformation to the limit, transforming a source program into code that was closer and closer to assembly language until, well, that's what it was: assembly language written with parentheses. Kelsey showed the generality of the approach by also building a front-end for Pascal.

3. Eric Mohr's dissertation [10] at Yale in 1991 described a parallel implementation of T called *Mul-T* that used a notion of "lazy tasks" that relied heavily on Orbit's light-weight closures. Mohr and Kranz later used their expertise to help Bert Halstead build more parallel Lisp systems at MIT [9].

4. James Philbin saw the potential of using T and Orbit to define an operating system, one that could efficiently support languages requiring closures and garbage collection. The use of "one-shot continuations" to capture threads, and T's fine-grained notion of interrupts, were key enabling technologies in this development, as described in his dissertation [11] at Yale in 1993 and several other papers [5].

5. Jonathan Rees also wrote a dissertation [12] that used a Scheme-like language as the basis for an operating system, where the emphasis was more on OS protection mechanisms. Rees wrote his dissertation at MIT in 1995.

6. Olin Shivers' dissertation [14] at CMU in 1991 described a method to do control-flow analysis in the presence of first-class procedures, based on his experience working on an early version of Orbit in the summer of 1984, and on Paul Hudak's work on flow analysis for pure functional languages.

7. Norman Adams designed the assembler for Orbit, which became his Master's thesis [1] from Yale in 1986. It generated machine code for a variety of machine architectures and was driven by declarative descriptions of those architectures.

Orbit also influenced Andrew Appel, whose dissertation [3] described compilation techniques for SML, and formed the basis of "SML of New Jersey," for a long time *the* standard implementation of SML. Interestingly, Appel's PhD student Zhong Shao returned this thread of compiler development full circle back to Yale, when he became a professor there in 1994 and established the now flourishing FLINT group.

Orbit's source-to-source optimizer has been in continuous use by Kelsey since he left Yale in 1988. He used it to write the Pre-Scheme compiler circa 1990 (Pre-Scheme essentially allows you to write machine code in Scheme), which is still used to compile the Scheme 48 virtual machine from Scheme to C. Over the last year and a half Kelsey has been working with Michael Sperber and Martin Gasbichler on using his compiler to produce a Scheme 48-compatible native-code compiler for Scheme.

Perhaps the most innovative design feature of T, and surely the most under-appreciated, was its fine-grained notion of *objects* [2, 15]. This mechanism was cleverly tied into the assignment mechanism, so that `(set! (car x) y)` was shorthand for `((setter car) x y)`. This, in turn, worked as follows: `car` is sent the `setter` message; it returns the `set-car!` procedure, which is then applied to arguments `x` and `y`. This mechanism was completely general, so that the setter procedure could do arbitrarily sophisticated things, not just overwrite a data structure. Note that this mechanism bears strong resemblance to C#'s *properties* with its "getter and setter" methods.

In a broader sense, Orbit's demonstration that Lisp could be compiled into efficient machine code was very influential in the development of other functional languages, including Haskell. More specifically, Orbit was the first compiler to demonstrate that procedure calls based on lexical closures did not have to be slow.

One might further argue that in the absence of such demonstrations the whole object-oriented movement would have been stillborn, depending as it does on the notion that you can have lots of small methods attached to private data and still get good performance. Thus we see a progression from Steele's Lambda papers, to T and Orbit, to C compilers with fast procedure calls, to C++ and beyond. (Sometimes the road to hell is paved with good intentions.)

REFERENCES

[1] Norman I. Adams. Assembling from machine descriptions. Master's thesis, Yale University, 1985.

[2] Norman I. Adams and Jonathan A. Rees. Object-oriented programming in Scheme. In *Proc. of ACM Conference on LISP and Functional Programming*, pages 277–288, 1992.

[3] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[4] R.M. de Morgan, I.D. Hill, and B.A. Wichmann. Modified report on the algorithmic language ALGOL 60. *Computer Journal*, 19(4):364–379, 1976.

[5] Suresh Jagannathan and James Philbin. A customizable substrate for concurrent languages. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 345–356, June 1992.

[6] Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Comp.*, 11(1):7–105, 1998.

[7] Richard Kelsey. *Compilation by Program Transformation*. PhD thesis, Yale University, Department of Computer Science, 1988.

[8] David Kranz. *ORBIT: An Optimizing Compiler For Scheme*. PhD thesis, Yale University, Department of Computer Science, 1988.

[9] David Kranz, Richard Halstead, and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[10] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs*. PhD thesis, Yale University, Department of Computer Science, October 1991.

[11] James Philbin. *The Design of an Operating System for Modern Programming Languages*. PhD thesis, Yale University, Department of Computer Science, 1989.

[12] Jonathan A. Rees. *A Security Kernel Based on the Lambda-Calculus*. PhD thesis, Massachusetts Institute of Technology, 1995.

[13] Jonathan A. Rees and Norman I. Adams. T: a dialect of LISP or, Lambda: the ultimate software tool. In *Proceedings 1982 ACM Conference on LISP and Functional Programming*, pages 114–122. ACM, August 1982.

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

[15] Olin Shivers. History of T, 2001. Appears on Paul Graham's Lisp website, http://www.paulgraham.com/thist.html.

[16] Guy L. Steele, Jr. Lambda — the ultimate declarative. AI Memo 379, MIT, November 1976.

[17] Guy L. Steele, Jr. Lambda — the ultimate imperative. AI Memo 353, MIT, March 1976.

[18] Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Master's thesis, MIT, 1978. AI Memo 474.