**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 abstraction 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 garbage 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.

   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.

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.


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 Gaschikler 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 still-born, 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