The Development of Chez Scheme

R. Kent Dybvig
Indiana University and Cadence Research Systems
dyb@cs.indiana.edu

Abstract

Chez Scheme is now over 20 years old, the first version having been released in 1985. This paper takes a brief look back on the history of Chez Scheme's development to explore how and why it became the system it is today.

Categories and Subject Descriptors D.3.4 [Programming languages]: Processors—compilers, incremental compilers, optimization, interpreters, memory management (garbage collection), runtime environments; D.3.2 [Programming languages]: Language classifications—applicative (functional) languages, Scheme

General Terms Algorithms, design, languages, performance, reliability

Keywords Chez Scheme, Scheme implementation

1. Introduction

Chez Scheme Version 1 was completed in 1984 and released in 1985. I am amazed to find myself working on it still more than two decades later. If asked in 1985 to look forward twenty years, I would have said that Chez Scheme and Scheme itself would have long since found their way into the bit bucket of history. After all, the oldest languages then were no older than Scheme is today, and many languages had come and gone. Many languages have come and gone since, but Scheme, with its roots in the circa-1960 Lisp, lives on. The user community now is larger and more diverse than ever, so with any luck, the language and implementation will last at least another two decades. It's a scary thought.

Longevity is tied to adaptability, and the current version of Chez Scheme is, to be sure, much different from its 1985 counterpart. It implements a much larger and different language and sports a much richer programming environment. The compiler is much more sophisticated, as is the storage management system. Whereas the initial version ran on one architecture and under one operating system, the system now supports a variety of different computing platforms and has supported many others at one time or another.

Still, the principles behind Version 7 are the same as those behind Version 1. Our primary objectives remain reliability and efficiency. A reliable system is one that correctly implements the entire language and never crashes due to a fault in the compiler or run-time environment. An efficient system is one that exhibits uniformly good performance in all aspects of its operation, with a fast compiler that generates fast code and does so for the widest variety of programs and programming styles possible. While we have added many new features over the years, and improved the system's usability with better feedback and debugging support, we have always done so in a way that took our primary objectives into account.

The paragraphs above first appeared in the preface of the *Chez Scheme Version 7 User's Guide* [21], which was published in 2005. The user's guide goes on, of course, to document the language as it exists today, and says no more about the history of the system. The purpose of this paper is to explore that history, to answer how and why the system came to be what it is today.

The remainder of the paper begins with a brief description of the systems that, in one way or another, were precursors to Chez Scheme (Section 2). It then describes the motivations behind the initial and successive versions of Chez Scheme and some of the more important new language features or implementation techniques that appeared in those versions (Sections 3 through 10). The paper concludes with some parting remarks (Section 11).

2. Precursors

Chez Scheme did not materialize out of a vacuum. What follows is a description of several Scheme or Lisp systems I worked on before Chez Scheme, systems that influenced the design and implementation of Chez Scheme in one way or another.

2.1 SDP

Scheme Distributed Processes [13] (SDP) was a multi-threaded implementation of Scheme written in 1980–81 by fellow Indiana University graduate student Rex Dwyer and me. The system was primarily a vehicle for investigating the Distributed Processes model of concurrency proposed by Per Brinch Hansen [34]. It grew out of an assignment given to us by Dan Friedman in his graduate programming languages seminar, which he taught using the book he and Bob Filman were writing on concurrent programming techniques [32].

SDP supported a large subset of the 1978 (revised report) version of Scheme [52]. In addition to its parallel processing extensions, SDP also supported arrays, a partial application mechanism, `dskin` and `dskout` functions for loading and saving definitions, and even a structure editor. SDP departed from Scheme's semantics by distinguishing false from the empty list and also by requiring that the cdr of a list also be a list. SDP was written entirely in Simula [3] and took advantage of Simula's run-time system, including most importantly its garbage collector.

Although none of the code of SDP survived into any of the Scheme systems I wrote later, it provided me my first experience with Scheme and with implementing a Lisp dialect of any kind.

ICFP '06 September 18–20, 2006, Portland, Oregon, USA.

2.2 Z80 Scheme

In 1981, while working as systems programmers in the academic computing center at IU, George Cohn and I decided to create an implementation of Scheme for the Z80 microprocessor. In so doing, George could teach me how to program in Z80 assembly, and I could teach him about Scheme. George was an incredible programmer, and I learned a lot from him in what I am sure was the better end of the bargain.

We coded the Z80 Scheme system entirely in Z80 assembly under the CP/M [48] operating system. The system supported most if not all of the 1978 version of Scheme, including full continuations. All values took up exactly 32 bits (two sixteen-bit words) or were made up of linked chains of 32-bit values. With all objects aligned on 32-bit boundaries, we were able to use the low-order two bits of each word for tagging and garbage collection. The system was interpreted and included a simple mark-sweep collector and free-list allocator.

About a year later, we created a second version of the Z80 Scheme system, making two major changes. First, we converted the collector into a mark-sweep-compact collector, reinventing the "two-finger" compaction algorithm apparently first proposed by Daniel Edwards [49]. This algorithm actually ran faster than the original, and it allowed us to use faster inline allocation. Second, we eliminated support for full continuations so that we could use a traditional recursion stack. Because the stack and heap grew toward each other, compacting the heap also ensured that the system did not run out of either stack or heap space before memory was truly exhausted. Overall, this system was much faster than the original, but I regretted losing support for full continuations.

We also attempted a compiler for the Z80 Scheme system. We designed a Scheme Assembly Language analogous to the Lisp 1.5 Lisp Assembly Language (LAP) [45], implemented it as a set of library routines to save code space, and generated from the Scheme source what amounted to a series of calls to these library routines. Sadly, because of the library-call overhead, the system ran no faster than the straight interpreter, and the generated code was larger than the original source code, so we abandoned the compiler.

2.3 C-Scheme

In 1982 I also began the implementation of a new dialect of Scheme, Curry Scheme (later abbreviated C-Scheme) [14]. The system employed a preprocessor, written in Scheme and bootstrapped via the Z80 Scheme system. The preprocessor performed macro expansion and also took care of currying both applications and `lambda` expressions. This was my first experience writing a macro expander and also my first experience with bootstrapping. The run-time system and interpreter were initially implemented in Pascal on the Z80 but were later recoded in C on the VAX. The storage management system employed the *big bag of pages* (BiBOP) representation of memory, in which memory is broken up into fixed-size segments and a separate segment table is used to identify the type of object contained in each segment [51]. Objects larger than a segment were supported by allocating two or more consecutive segments. Unboxed native integers (fixnums) were supported by leaving empty the lowest and highest portions of the virtual memory address space and setting the corresponding segment table entries to the type code for fixnums.

Although Chez Scheme had not yet even been conceived, C-Scheme was an important step toward Chez Scheme. Chez Scheme's initial run-time system borrowed heavily from that of C-Scheme, and C-Scheme was used to bootstrap the first version of the Chez Scheme compiler.

2.4 Data General Common Lisp

In an incredible coincidence, I happened to be in Dan Friedman's office just before heading to graduate school at the University of North Carolina in 1982 when Jed Harris of Data General in Research Triangle Park called to ask Dan if he knew of anyone who might be interested in coming to North Carolina to help them initiate a Common Lisp effort. Dan put me on the phone, and Jed and I arranged to meet after I got settled. I hired on as a contract employee and served by myself as the entire DG Common Lisp group for about a year, with Jed looking over my shoulder from time to time. The plan was to adapt the work I had done for C-Scheme into a run-time system, with an interpreter for the core of Common Lisp, then graft a compiler on at some later time. By the end of the year, I had written a storage management system, I/O system, interpreter, and various primitives in DG's proprietary systems programming language. At that point, several other people were hired on, the Spice Lisp compiler was brought in from CMU to be combined with the run-time system, and I backed off to occasional consulting so that I could focus more attention on my PhD research.

During the summer of 1984, I was asked to work full time at DG to repair the storage management system, which had been replaced with one that ran two orders of magnitude slower than my original and crashed during the second collection cycle. Simply reverting to my earlier code was not an option, as by now new object types had been added and the representations of some others had changed. Besides, although my old collector was faster than the one that replaced it, it was still pretty slow, taking something like one or two minutes to collect an 8MB heap on DG's flagship MV/10000 computer. (Don't laugh. We had reports of other, much slower collectors outside of DG.) So I was asked to make one that was much faster. Unfortunately, I had only until the end of July to make it all happen, as DG had plans to demo their Common Lisp at the co-located Lisp and Functional Programming and AAAI conferences in early August. On the other hand, I was given an outstanding partner, Rob Vollum, which made the task a lot easier, and more pleasant. Working 18 hour days, we had a solid storage management system with a fast collector just in time for the demo.

We gained most of the performance by leaving in place (but still tracing) all of the system data structures in what became known as the static part of the heap. This was a poor variant of generation scavenging [43, 54], about which I had not yet heard, but it was still fairly effective. The BiBOP representation inherited from C-Scheme allowed us to avoid tracing segments that contain no pointers, e.g., those containing strings, and we also collected these segments infrequently to avoid bringing them into memory if they had been paged out. The end result was a collector that averaged around 15 seconds to collect an 8MB heap—very slow by today's standards, but respectable at the time.

By all accounts, the demo at LFP and AAAI was a success. Unfortunately, as I understand the situation, the marketing division within DG that had bid and won the right to sell the Common Lisp product shortly thereafter received a lucrative contract for some different effort, and the Common Lisp product never really saw the light of day. I learned a lot from the project, however, that I was able to use in my work on Chez Scheme. In addition to gaining valuable experience with storage management, I remember being impressed by the seriousness with which DG approached quality assurance and became a firm believer in the development of extensive test suites. Also, although I did not have a chance to work with the compiler directly, I did write code for handling `lambda` lists, an unpleasant task that helped push me toward a more minimalist approach to language design in later years.

3. Chez Scheme Version 1

During the relative respite from DG in Fall 1983 and Spring 1984, I worked in earnest on the design of a parallel implementation of Scheme for my advisor's (Gyula Mago's) cellular computer [44], which had been my intent from the day I decided to go to school at UNC. Since there was actually no machine yet, I also started writing a simulator (in Scheme, naturally). Unfortunately, C-Scheme wasn't fast enough, although it was faster than the other Scheme systems I tried. I ported the simulator to Franz Lisp [33], which was a good system, but was frustrated by its poor handling of function arguments—which the compiler seemed to punt to the interpreter—and by a lack of consistent semantics between the compiler and interpreter. So I decided to undertake, in parallel with my research work, the design and construction of a compiler for Scheme, which eventually became Chez Scheme.

As part of the design process, I profiled the C-Scheme implementation and discovered that most of the time was spent in variable lookups and stack-frame creation. It dawned on me that the typical implementation model for Scheme was all wrong: by heap allocating environments and call frames, it made closure creation fast at the expense of the more common variable references, and it made continuation operations fast at the expense of the more common procedure calls. In our second Z80 Scheme implementation, we opted to sacrifice full continuations to enable stack allocation of stack frames, and the designers of T had done likewise [47], but I wasn't willing to go that route with Chez Scheme. Instead, I started to think about ways to make continuations "pay their own way," and at the same time, shift the burden somehow from closure access to closure creation.

The solution for continuations seemed obvious: use a stack for procedure calls, implement continuation capture by copying the stack into a heap-allocated data structure, and implement continuation reinstatement by copying the stack copy back to the stack. With environments still heap allocated, variable values would never be stored directly on the stack, and there would be no concern about making multiple copies of mutable variables. On the other hand, my ultimate goal was to use traditional stack frames in which local variables were stored on the stack, and I wasn't sure how this was going to work out.

Solving the closure issue was a bit trickier. While researching how other systems had coped with similar problems, I ran across a book on the implementation of Algol 60 by Brian Randell and Lawford Russell [46], which described the use of displays for speeding access to the free variables of a local function. A *display* is a bank of memory locations or registers, each pointing to one of the frames whose variables make up the current lexical environment. Displays weren't directly usable for my purposes, but I was able to make several adjustments and from the display model derived the notion of a *display closure*, a heap-allocated vector-like object holding a code pointer and the values of the free variables [16]. In addition to allowing constant-time access to all variables, it had the added benefit that closures hold on to no more of the environment than they require, which had the potential to make garbage collection more effective.

Assigned variables were a problem with this representation, since a variable's value could potentially appear in multiple closures. I dealt with this by "boxing" assigned variables, i.e., replacing each assigned variable's value with a pointer to a heap-allocated single-celled object, or box, holding the actual value. (A variable is assumed to be assigned if it appears on the left-hand side of an assignment somewhere in its scope.) I learned later that Luca Cardelli used a similar flat representation of closures in his ML implementation [11]. In ML, variables are immutable, so there is no need for the compiler to introduce boxes. One can view the introduced boxes as a form of ML *ref* cell, however, the difference being that, in ML, the programmer must introduce the *ref* cells explicitly, whereas, in Scheme, the compiler introduces the boxes implicitly.

Boxing assigned variables also solved the stack problem, because it allowed the values (or boxed values) of local variables to be stored directly in a stack frame without concern for the fact that the frame might be copied as a result of a continuation capture.

With the new closure and continuation models, the cost of creating a closure became proportional to the number of free variables, but the cost of accessing a variable's value became small and constant—one memory reference if no assignment to the variable appears within its scope, otherwise two. The cost of creating or reinstating a continuation became proportional to the size of the stack, but call frames became stack rather than heap allocated, saving linkage overhead, reducing the frequency of garbage collection, and allowing common portions of a frame to be reused when multiple non-tail calls were made. The use of assigned variables also became more costly, but assigned variables are (and should be) used infrequently in Scheme, so that was not much of a concern. A bigger concern was that proper treatment of tail calls became potentially more costly. The callee's arguments must be placed in the same locations as the caller's local variables to prevent the stack from growing, but the caller's locals are generally needed to produce the callee's arguments. The simple solution used in Version 1 was to place the callee's arguments above the caller's locals and shift them down just before transferring control to the callee.

As I became excited by the new closure and continuation models, my primary motivation for developing Chez Scheme shifted from building a tool for use in my research to proving that my new ideas could be used to build a fast and reliable Scheme implementation for use by others as well as me. Because I had found other systems to be lacking in reliability as well as speed, I refocused my efforts on building a system that was not only fast but also reliable, with full type and bounds checking, including stack overflow checking, even in compiled code. The new focus turned out to be a strong motivator, and by the start of Summer 1984 I had an interpreter and run-time system, much of which was copied from C-Scheme but incorporating ideas borrowed from my DG Common Lisp storage manager. I had also written the compiler's front end, including the assignment and closure conversion passes needed to implement the new models. Unfortunately, I had to take a break when the call came from Data General to work on their storage manager, and I was not able to work on it again until that fall.

When I returned to work on Chez Scheme that fall, I shored up the run-time system and, with the help of fellow UNC graduate student Bruce Smith, added a bignum arithmetic package. This left only one major task, the construction of the compiler back end. I had originally intended to follow Lisp tradition and provide an interpreter for interactive use, but Luca Cardelli visited and showed off his slick interactive, incremental compiler for ML, and I was inspired to go the same route for Chez Scheme. This made the back end more difficult, since I could not use the system assembler and linker, which would have been too slow even without the process-creation overhead, but instead had to write my own. In the end, these turned out to be pretty straightforward, but at the time it seemed an immense challenge, and I put it off for quite a while. Finally, however, by the end of 1984, Version 1 of Chez Scheme was complete.

The compiler and portions of the run-time system were written in Scheme and bootstrapped using C-Scheme. The remainder of the run-time system was written in C and assembly. The system supported only one architecture, the VAX, and one operating system, BSD Unix.

In a high-level sense, the compiler was naive. It handled a small set of core forms and did only one thing that could charitably be called a high-level "optimization:" it treated the variables bound by

a direct lambda application as local variables to avoid the cost of allocating a closure and calling that closure. As I tell my compiler students now, there is a fine line between "optimization" and "not being stupid." This was really an instance of the latter.

My focus was instead on low-level details, like choosing efficient representations and generating good instruction sequences, and the compiler did include a peephole optimizer. High-level optimization is important, and we did plenty of that later, but low-level details often have more leverage in the sense that they typically affect a broader class of programs, if not all programs.

One important representation hack was the inclusion of a code-pointer slot as well as a value slot in each symbol to improve the speed of calls to globally bound procedures, including primitives. By maintaining a separate code-pointer slot through which global calls jump unconditionally, the compiler doesn't have to generate a `procedure?` check at each global call. The slot is initialized when the symbol is created or assigned to the address of a trap handler that patches up the code pointer and completes the call, if the value is a procedure, or signals an error, if the value is unbound or not a procedure. After the first successful call, the trap handler is thus bypassed and control passes directly to the called procedure.

It is possible to extend this idea to multiple code pointers for different interfaces, thus avoiding argument count checks as well. I decided not to do this in Version 1 for several reasons: (a) the extra code pointers would bloat the size of a symbol or require the use of an auxiliary data structure, (b) handling multiple code pointers would add complexity to the compiler and storage management system, (c) the savings in run time would have been less significant, since argument-count checks were cheaper than type checks, which involved a segment-table indirect, and (d) the savings in code size would have been less significant, since the argument-count checks were performed at the entry point, not at the call site, and call sites are typically more numerous than entry points.

This is the way that decisions with Chez Scheme have usually gone, especially in the early days. We pick the low-hanging fruit, that which is easy and yields the biggest savings, leave the rest for later, if ever, and move on to pick the low-hanging fruit on some other tree. There's no point in trying to achieve perfection in one aspect of performance when something easier of even greater benefit is waiting to be done.

Although I don't recall the details now, Version 1 outperformed the other Scheme and Lisp systems to which I had access, in spite of the complete type and bounds checking, so I felt that the new closure and continuation representations had proven themselves. I distributed a half dozen copies of Version 1 in Spring 1985, through UNC, and feedback from the users was positive as well.

4. Chez Scheme Version 1.1

During Spring 1985, Bruce Smith and I created a reference manual [31]. I converted many of the example programs in the reference manual into the start of a test suite and then augmented the test suite with many additional tests. I found and fixed a few bugs during this process, and by summer completed Version 1.1 of the system. At the same time, my wife Susan Dybvig and I started Cadence Research Systems to distribute and further develop the system. Susan's MBA degree and prior experience with a small software company complemented my training and experience, and we managed to make our first commercial shipments of Version 1.1 that summer. All of our profits then and since have been reinvested to pay for the costs associated with development—mostly labor.

Susan also obtained a contract with Prentice-Hall to publish our reference manual. Prentice-Hall chose the title *The Scheme Programming Language*, in an obvious attempt to capitalize on the success of *The C Programming Language* [39], which they also published. I was pleased with this, but it raised the bar considerably

Release year: 1984 (Version 1.0), 1985 (Version 1.1)

Language: R2RS compatibility, fixnums, bignums, ratios, flonums, simple macros, saved heaps, saved executables, engines [27, 35], trace support, multiple waiters and cafés, primitive macro system, timer and keyboard interrupts, property lists, primitive `format`, `dynamic-wind`, fluid binding

Implementation: incremental compiler (no interpreter), custom linker, flat ("display") closures, stack-based representation of control, boxing of assigned variables, BiBOP typing with reserved (16-bit) fixnum range, stop-and-copy collector, code pointers cached in symbol code-pointer slot, peephole optimization

Documentation: *Chez Scheme Reference Manual Version 1.0* [31], Unix manual page

Platforms: Vax BSD Unix

Version 1 Highlights

in my mind, and I ended up rewriting most of the text before the book was finally published in 1987. Unfortunately, my coauthor, Bruce Smith, was not able to spend much time on the effort, and I eventually asked for and received his permission to complete the project on my own. I am eternally grateful for his critical early contributions both to the system and to the book.

Initially, I wrote all of our code and documentation and we administered our business using a home-built Z80 PC, which with its 8Mhz Z80H processor, 128K of RAM, and 20MB hard disk was actually faster for those purposes than either the new IBM PCs or even the VAX systems I had used at UNC. (To be fair, I had customized my copy of Gosling's Emacs to emulate WordStar, the word processing system we used on the Z80. That slowed Emacs down considerably and probably accounts for why the VAX seemed slower.) Builds and testing were done on a VAX computer at the Microelectronics Center of North Carolina (MCNC), to whom, in exchange, we provided a free license for Chez Scheme. We eventually purchased a Sun workstation and moved our development work to that platform, and we bought or were provided with other systems from time to time, but we continued to barter for or borrow time from clients for builds and testing on platforms we did not have in house.

The first Sun workstation we bought was their lowest-end system. I would have liked a faster system with more memory, of course, had we been able to justify the cost. On the other hand, I considered having a low-end machine an advantage of sorts: if I could make Chez Scheme run well on it, Chez Scheme would run well on anything.

5. Chez Scheme Version 2

In Fall 1985, we moved to Bloomington Indiana, where I joined the faculty at Indiana University. Development was largely put on hold during the academic year while I prepped and taught a full-year compiler course and one other course. Sometime during the spring semester, however, George Davidson of Sandia National Laboratories, who had been using Chez Scheme for several months, asked us to create a cross-compiler from the VAX to an embedded MC68000-based system. It was a great opportunity to work on an MC68000 code generator, which later allowed us to port the system to the Sun, Apollo, and a couple of other platforms. I decided to hire a graduate student from my compiler class to help me over the summer with the project.

The compiler class was full of really good students, but one stood out (literally as well as figuratively) above the rest. Bob Hieb

Release year: 1987

Language: R3RS compatibility, internal definitions, EPS macros, `extend-syntax`, `case-lambda`, optimization levels, fixnum operators, 19-bit fixnums, pretty-printer, user-defined structures, user-defined error handlers

Implementation: multiple back ends, multiple operating systems, automatic closing of files by the collector, optimizing `letrec` expressions and loops, inlining of primitives, destination-driven code generation [30] (obviating peephole optimizer), faster compiler, faster collector

Documentation: *The Scheme Programming Language* [15], Unix manual page

Platforms: Vax BSD Unix, Apollo Domain/IX, Sun-3 SunOS, Alliant, VAX VMS

Version 2 Highlights

was 6 feet 6 inches tall, with broad shoulders, bushy hair, a full beard, sharp features, and a typical outfit consisting of flannel shirt, jeans, and boots. He looked more like a lumberjack than a computer scientist (I learned later he had spent ten years as a carpenter), but his performance in the class demonstrated great potential, so I offered him the job. He gladly accepted, and we ended up working together for seven fruitful years before his untimely death in 1992.

After completing the MC68000 cross compiler, Bob and I worked together on other aspects of the implementation. We implemented different optimization levels, which were really just flags telling the compiler it could do certain things. At optimization level 1, it was allowed to spend more time. At optimization level 2, it was allowed to assume that the global names of primitives were indeed bound to those primitives. At optimization level 3, it was allowed to generate unsafe code. We also introduced some code into the compiler to optimize letrec expressions, optimize loops, and inline most simple primitives

Since the compiler was coded in Scheme and benefited from its own optimizations after bootstrapping, these improvements made the compiler itself faster. In fact, the optimizations more than made up for the extra work done by the compiler to implement the optimizations. This was a good thing, because we were concerned about compiler speed (the compiler was, after all, used interactively and for loading source files) as well as effectiveness. At some point we actually instituted the following rule to keep a lid on compilation overhead: if an optimization doesn't make the compiler itself enough faster to make up for the cost of doing the optimization, the optimization is discarded. This ruled out several optimizations we tried, including an early attempt at a source optimizer.

We also worked on the collector to see if we could improve its performance. There were slim pickings in terms of the algorithm, which was already as tight as we knew how to make it. We were, however, able to pick up (according to a comment in the code) from "10 to 33 percent improvement" by defining a key routine as a C preprocessor macro instead of as a C function.

In addition to work on porting and making the system faster, we also adopted some new language features, including support for low-level expansion-passing-style macro definitions [24, 25], and high level `extend-syntax` macro definitions [41]. We did not adopt hygienic macro expansion [41, 42] until much later because of the quadratic expansion cost and other limitations of the mechanism that we did not solve until much later.

We also added a new feature for creating "variable arity" procedures, i.e., procedures with multiple interfaces, a generalization of optional arguments. A `case-lambda` expression is like a `lambda` expression but has multiple clauses pairing a formal-parameter list with a body. When a procedure created with `case-lambda` is called, the appropriate clause is selected based on the number of actual parameters received. Our original design called for the replacement of the dot interface with a different syntax that allowed one or more of the cases to accept an indefinite number of arguments without committing to any particular representation of these arguments. This effectively removed lists from the interface, along with various difficulties that lists can cause with optimizing calls to procedures of indefinite arity [26, 28]. We backed off of this feature and instead settled on a less radical version in which the formal-parameter list of each clause is a normal `lambda` formal-parameter list.

Version 2 was released in 1987, about the same time as *The Scheme Programming Language* was published by Prentice-Hall.

6. Chez Scheme Version 3

Between the releases of Version 2 and Version 3 we continued to tweak the compiler and run-time system to improve performance, but most of our time was spent on ports to a pair of new RISC architectures, improving the interoperability of Chez Scheme with other languages and processes, and improving the overall usability of the system.

One change that improved both performance and usability was the adoption of a new continuation mechanism. In Versions 1 and 2, capturing a continuation meant copying the entire stack, and reinstating a continuation meant copying it all back, which meant that continuation operations could become inordinately expensive. We solved this problem with a new segmented stack approach that Bob and I developed with Carl Bruggeman, another of my graduate students [36]. This mechanism eliminates the need to copy the stack when a continuation is captured and reduces the amount copied to a small constant number of words (or at most the size of the largest frame) when a continuation is reinstated, without adding any overhead to normal procedure calls and returns. The mechanism also supports automatic stack-overflow recovery so that stack space is never exhausted as long as heap space remains available to be allocated.

By good fortune rather than design, the new continuation mechanism also enabled us to modify the trace package, in response to a request by Olivier Danvy, to reflect the difference between non-tail calls (by increasing the trace display nesting level and displaying the return value) and tail-calls (by doing neither). The challenge is recognizing which calls to traced procedures are non-tail calls and which are tail calls, and at first we were stumped. If the trace package were built into an interpreter, the interpreter could track this information, but we had no interpreter. Instead, the trace package operates by embedding each traced procedure in another procedure, a trace wrapper that takes care of printing the trace information. The solution is for the trace package to maintain a trace-continuation variable that always holds the continuation of the last traced call. Each trace wrapper compares its own continuation against the trace continuation. If they are the same, a tail call has occurred, and if they are different, a non-tail call has occurred. For non-tail calls only, the trace wrapper sets the trace continuation to the new current continuation before applying the embedded procedure and restores the old trace continuation after. With the segmented-stack representation of continuations, the comparison of continuations can be done with a single `eq?` test, i.e., a pointer comparison.

The first of our RISC ports was to the Motorola MC88000 and was done at the behest of Sam Daniel at Motorola, who championed Chez Scheme at Motorola for many years and even managed to convince Motorola to market Chez Scheme on their Delta Series MC68000 and MC88000 series machines for a short while. The

MC88000 is one of the most difficult ports we have undertaken, and it took the combined efforts of three people (I had recruited Carl as well as Bob to help) over most of a summer to do the job. This was partly because its RISC architecture was radically different from the CISC VAX and MC68000 architectures, but mostly because the operating system, C compiler, and even the hardware were still under active development at the time of the port. Naturally, this meant dealing with several compiler and operating system bugs, but the biggest challenges were hardware bugs. One was challenging, in part, because it went into hiding when we set breakpoints or single-stepped the code. It turned out to be a race condition with a missing scoreboard (busy) check on a move from the hardware return-address register into our own return-address register. By the time we proceeded from the breakpoint or single-stepped the code, the register was no longer busy and our code ran without problems. When run at full speed, however, our code would occasionally try to access the register too soon and, instead of being forced to wait, would get the wrong (old) value, usually causing a procedure to return (eventually) to the caller's caller rather than to its own caller.

The other port was to the Sparc architecture, and that went much more smoothly, because we had already ported to one RISC architecture, the operating system was essentially the same as for the MC68000 Sun systems, and the tools and hardware were more stable.

Interoperability with other processes and other languages within the same process was a priority for some of our users. This prompted us to add tools for loading foreign object files and invoking entry points in those files, with automatic conversion of argument and return value data types to and from the C and Scheme representations. These tools were modeled after similar features in the implementations of other Lisp dialects. We also added support for running other programs in subprocesses and interacting with them via Unix pipes.

Most of the new code we wrote for Version 3, and for all of the other new versions, was written in Scheme, and some that was originally written in C has since been converted to Scheme, where it can be coded in a more abstract style that is less likely to be affected by changes in the system, e.g., to object representations. The largest piece of code we converted while working on Version 3 was the Chez Scheme reader, which implements the Scheme `read` procedure and is also used when loading or compiling source programs. We expected that the reader would be somewhat slower, but not by enough to justify leaving the reader in C. The Scheme-coded reader actually turned out to be faster, despite the fact that the Scheme version used mutually tail-recursive routines for lexical analysis while the time-critical portions of the C version used `while` loops. Although the difference was probably partly due to the relative ease with which we could tune the Scheme code, it was nice to know that Chez Scheme's compiler was starting to get to the point where it could actually compete with C compilers, which have a much easier job to do. The Scheme-coded reader became even faster in the next release, when we made significant improvements in register allocation, including allocating registers to procedure arguments.

Version 3 was released in 1989.

7. Chez Scheme Version 4

Shortly after Version 3 was released, we began a major overhaul of the system, changing the way it represents Scheme values and rewriting major portions of the compiler and storage management system. In so doing, we intended to remove some object-size limitations and, as always, to improve performance and reliability. We also wanted to leave the system with a stronger foundation that took into account the system modifications we had already made or knew we wanted to make in the future. A system that isn't overhauled from time to time can fall under its own weight as new features are grafted onto a foundation that wasn't originally designed to support them.

Release year: 1989

Language: foreign procedure interface, subprocess creation and communication, tail-recursive trace, SICP [1] compatibility mode

Implementation: constant-time continuation operations and stack-overflow recovery, various new compiler optimizations, faster compilation, RISC architecture ports

Documentation: *The Scheme Programming Language* [15], *Chez Scheme System Manual* [8], Unix manual page

Platforms: Vax Ultrix and BSD Unix, VAX VMS, Sun-3 SunOS, Sun-4 SunOS, Motorola Delta MC680X0 and MC88000 SVR3, Intel NeXT Mach

Version 3 Highlights

We had stuck with the BiBOP mechanism adopted from C-Scheme through the first three major releases of Chez Scheme. Because type information is stored in a separate table, the BiBOP mechanism allowed us to use native representations for fixnums and pointers and to avoid taking up space in any data structure for type information. On the other hand, the cost of doing a type check was rather high. It involved extracting a segment address from the address of an object, using this as an index into the segment table, and doing a test and branch.

We were also just feeling the effects of another problem, caused by rapid increases in typical physical memory and backing store sizes. These increases created corresponding pressure to increase the maximum sizes of strings and vectors. This meant increasing the size of the fixnum range, since indices into these objects are represented as fixnums for efficiency. At the same time, the increases in typical physical memory and backing store sizes also meant that processes were using a larger percentage of the virtual memory address space. Unfortunately, our fixnum range was obtained, as described in Section 2.3, by sacrificing a portion of this address space. Increasing the fixnum range to allow larger strings and vectors would decrease the amount of space to put them in. This problem forced us to consider moving away from the BiBOP model and toward a tagged pointer model.

The BiBOP model has many benefits that we were reluctant to lose, however. By segregating objects containing pointers from objects not containing pointers, it allows the garbage collector to avoid sweeping objects, like strings, that don't contain pointers. More generally, it allows the collector to use different methods for sweeping different types of objects, which in turn permits the use of more clever representations for some objects, like I/O ports containing "next" pointers into the middle of their string buffers or closures containing code pointers into the middle of a code object. It allows dynamically generated or loaded code to be placed in different segments within the heap, so that code pages can be given different protections from pages containing only data. It also allows the storage manager to handle "holes" in the virtual address space gracefully; holes can occur when the O/S or some other language run-time has reserved space for some other purpose.

Ultimately, we decided to switch to a hybrid model [23] that uses tagged pointers to distinguish specific types of objects and also BiBOP for its various benefits. We no longer used the BiBOP mechanism to segregate objects by specific type but rather by characteristics of interest to the collector, such as whether they contain pointers and whether they are mutable. For our implementation of tagged pointers, we adopted the low-tag model used by T [47], with

a different assignment of tags. The hybrid mechanism allowed the fixnum size to be increased to 30 bits and type checking overhead to be reduced in many cases, without sacrificing virtual address space or the benefits of the BiBOP mechanism described above.

Changing object and pointer representations affected the compiler in various predictable ways. For example, it had to generate different code for accessing or mutating objects, performing arithmetic, and type checking. The change also allowed us to switch to inline allocation. Because different types of objects were no longer distinguished by the type of segment they resided in, we were able to allocate most objects (all but code objects) into a single area of memory, then segregate the objects that survive their first collection. This meant we needed only one allocation pointer, which we could put into a register, and the allocation code sequences became small enough to justify inlining certain allocation operations, such as pairs and closures. This gave a significant boost to the performance of many programs.

These changes were just one task of several we undertook while rewriting the compiler. Another task was to replace the list structures used to represent intermediate code with compiler-specific record structures, or c-records. Because each c-record is a flat structure rather than a linked list, this reduced the size of the intermediate code and the cost of accessing the subexpressions of each intermediate form. At the same time, it allowed us to reduce dispatching overhead. Reliability increased as well because the shape of each record is checked statically at the creation and dispatch sites and also because the c-records are immutable, preventing inadvertent modification.

Another task was to replace our local register allocator with an intraprocedural register allocator. At the same time, we altered Chez Scheme's calling conventions to allow procedure arguments to be passed in registers. Prior to Version 4, all arguments were passed in stack locations. While this was common at the time on CISC architectures, which supported many operations directly on memory and did so with relative efficiency, it was certainly not a good idea on the RISC architectures. With a language that emphasized procedure calls over loops as the basic mechanism for repetition, we needed a mechanism that handled calls well. Unfortunately, the register allocation literature was focused almost exclusively on generating good code for traditional (Fortran) loop-based programs and either did not address procedure calls or did so in a cursory manner. We briefly considered adapting graph coloring register allocation [12] to our needs, but it offers no special help for calls, and the potential compile-time cost was beyond what was reasonable for an interactive system based on incremental compilation.

So Bob and I developed our own register-allocation algorithm. The algorithm assigns registers first to the incoming arguments, then on a first-come, first-served basis to the bindings found in a bottom-up pass starting with the leaves of a procedure's abstract syntax tree. So that registers containing values that are no longer needed can be used for other values, the algorithm employs an inexpensive form of live analysis for the register values only, allowing the use of cheap fixnum logical operations. The initial version of the algorithm saved and restored live register values around each non-tail call. Once the algorithm was working, we gathered various sorts of dynamic information for a set of benchmark programs, including our own compiler. Based on this information, we switched to a "lazy save" strategy in which live register values are saved as soon as but not before a call is inevitable. We also added a shuffling mechanism used at each call. The shuffler reorders arguments at each call site to reduce the number of register saves and allow argument values to be placed directly in their outgoing register or stack locations, with few or no extra moves. Shuffling obviated the shifting of tail-call arguments described in Section 3. The register allocation algorithm, though designed primarily to be fast, was surprisingly effective. A refined version of the algorithm was published several years later [7].

Release year: 1991

Language: IEEE Scheme standard compatibility, R4RS compatibility, parameters, complex numbers, flonum/complexnum operators, IEEE signed 0, infinities, NaNs, distinct empty list and false objects, void object, printer cycle detection, `compile-file` and linker preserve shared structure and cycles, collection of code and symbols, inspector

Implementation: hybrid tagging model, inline allocation, closure length moved from closures to code headers, literal references moved from closures to code stream, flonum size cut in half, generational garbage collection, intraprocedural register allocation

Documentation: *The Scheme Programming Language* [15], *Chez Scheme System Manual, revision 2.0* [9], Unix manual page

Platforms: Vax Ultrix, Sun-3 SunOS, Sun-4 SunOS, Decstation Ultrix, Motorola Delta MC680X0 and MC88000 SVR3, Intel NeXT Mach, SGI IRIX

Version 4 Highlights

Yet another task was to improve the compiler's support for inlining primitive operators to allow partial inlining of operators that cannot be fully inlined. We used this to inline safe versions of a number of primitives that defer to out-of-line error handlers and to inline the fixnum case for various generic numeric operators, like `+`. The same mechanism allowed us to introduce some mundane but useful program transformations, like one that replaces `eqv?` calls with cheaper `eq?` calls when one argument is constant and comparable with `eq?`.

The final compiler task was to improve support for floating-point operations and add support for complex operations. We added a set of new flonum operators, like `fl+`, which are inlined in unsafe code and partially inlined in safe code. We added a similar set of "complex flonum" operators, like `cfl+`, that work on combinations of flonums and inexact complexnums. In supporting complex operations, a major challenge was to devise an efficient representation of inexact complexnums. We didn't want the extra indirects involved with representing them as a pair of pointers to flonums, but we also wanted to avoid the cost of allocating a flonum object when extracting the real or imaginary part.

We were able to avoid both the indirects and the allocation overhead with a simple hack that also cut the size of our flonums in half. Because the collector moves objects as it works, it needs space in each object to leave behind a forwarding marker and address. This is a problem for floating-point numbers, because any marker we choose may be indistinguishable from raw floating-point data. On systems supporting IEEE floats, we considered encoding forwarding addresses as bit patterns that correspond to NaNs, but found that some architectures and operating systems don't document the set of NaNs that might be produced by their instructions and libraries. Thus, additional space had been included in each flonum for the forwarding marker and address. The hack was to remove this space and modify the collector so that it never stores a forwarding address in a flonum. A flonum might be replicated, of course, if two pointers to it exist, but causes no particular difficulties. The replication is detectable via `eq?`, but this does not violate the semantics of `eq?`, which is always allowed to return `#f` when given

two numeric arguments. Once we made this change, we were able to represent inexact complexnums as pairs of double floats (aka. flonums) and employ simple pointer arithmetic for extracting the real and imaginary parts.

The changes in representation of Scheme values, the addition of new complex number types, and the elimination of forwarding addresses from flonums caused us to make a few adjustments to the collector, but these were relatively minor. The major challenge we set for ourselves was to convert our stop-and-copy collector into a generational collector [43, 54]. As with our value-representation changes, this conversion was motivated by the increase in typical physical memory sizes. As memory sizes increased, the size of a typical heap grew, and the cost of collection grew with it. We turned to generational collection to reduce the cost. Generational collection is based on the theory that older objects, having survived through multiple collections, are less likely to be garbage than younger objects, and thus need not be collected as often.

Our variant [23] employs several generations, the number of which is determined when the system is built. Five generations are used by default, where generation 0 is the youngest, generation 3 is the oldest collectible generation, and 4 is a static generation containing code in the initial heap, after system and application "boot" code is loaded. New objects are allocated into generation 0, and objects are promoted to a higher generation when they survive a certain number of collections in a younger generation. The frequency with which each generation is collected and the number of collections an object must survive before being promoted to a higher level can be set by the programmer. The default is to collect generation n every 4^n times the collector is run, so generation 0 is collected every time, generation 1 every fourth time, generation 2 every 16th time, and generation 3 every 64th time. This rather arbitrary strategy was initially just a hack for testing, but it turned out to work well, indeed better than several more elaborate strategies we tried, so it has been the default ever since. It does a good job in practice of reducing the average time for a collection without the potential problems associated with the "premature tenuring" [54] of dynamically allocated objects into a generation that is never collected.

I have no record of the performance benefits from the various changes we made during the overhaul, but the overall improvement in run-time speed was about 50%. Perhaps more impressively, the speed of the compiler improved by 30%, despite the addition of the new register allocation passes.

Although most of our changes for Version 4 had to do with the overhaul, we also added several new features. One of these features was a new inspector, with both programmatic and interactive interfaces. The inspector allows programmers to view and modify, where appropriate, the contents of any object, including most importantly the continuation of an error or keyboard interrupt. To support the listing of source information and the proper labeling by variable names of values saved in a continuation or closure, we had to make additional changes to the compiler so that it would track source information through the various passes. This turned out to be a mostly straightforward process of adding a source field to each c-record, propagating the information through each pass, and finally associating the information with the addresses of the entry and return points in the generated code object. Support for inspection was made without any sacrifice in performance, so we were able to remain true to our priorities while addressing an obvious deficiency in the system.

8. Chez Scheme Version 5

We were all set to begin work on Version 5 over the summer of 1992 when Bob Hieb was killed in a car accident that also claimed the life of his daughter Iva. This tragic event could have derailed the development of Chez Scheme, but I dealt with my grief partly by engaging in a mad rush to complete the development work we had started or planned for Chez Scheme as well as some unfinished research work at IU.

Release year: 1994

Language: `syntax-case` macros, multiple values, guardians [22] and weak pairs, generic ports, saved (full and incremental) heaps, better foreign interface support

Implementation: local call optimizations, more compile-time checks, improved register allocation [7], improved code generation, 25X faster gensym

Documentation: *The Scheme Programming Language* [15], *Chez Scheme System Manual, revision 2.5* [10], Unix manual page

Platforms: Sun Sparc SunOS/Solaris, DEC Alpha AXP OSF/1, Decstation Ultrix, Motorola Delta MC88000 SVR3 & SVR4, Intel NeXT Mach, SGI IRIX, Intel BSDI, Intel Linux

Version 5 Highlights

Carl Bruggeman was particularly helpful with the latter, as he stepped in to work with me on finishing the `syntax-case` macro system which had been at the core of Bob's PhD research. We completed work on the system and a pair of technical reports describing the system [17, 37] that same year and published a journal article that combined parts of the two technical reports a year later [29]. I incorporated the `syntax-case` system into Chez Scheme and also published a portable version, which has been kept in sync with the Chez Scheme version ever since.

Most of the other system changes I made in 1992 were routine improvements in code generation, compile-time checking, and the like, but one major task was the optimization of local calls. Up to that point, most local calls involved a `procedure?` check and an indirect jump through the procedure's closure. The only exceptions were direct calls recognized as the equivalent of `let` expressions and calls participating in loops that the compiler recognized as such. Global calls were actually cheaper, because the code-pointer hack described in Section 3 obviated the `procedure?` check.

To optimize local calls, I modified the compiler to record when an unassigned variable is bound by `let-` or `letrec` directly to a `lambda` or `case-lambda` expression, then to generate more efficient code for calls made through such variables. In particular, the compiler eliminates the `procedure?` check, replaces the closure indirect with a jump into the `lambda` body (past the argument-count check or directly into the appropriate `case-lambda` body), passes a closure pointer only if the procedure has free variables, and, for "rest" arguments, allocates the list at the call site where the number of arguments is known and no loop is needed. In many cases, a procedure is called only in this manner. If it has no free variables, the closure is never used, so the compiler eliminates both the closure-creation code and the `let-` or `letrec`-binding for it. The compiler arranges for more procedures to fall into this category by recognizing when the only need for the closures in a group of mutually recursive procedures is to hold onto the closures of the other procedures.

The local-call optimizations made a large difference—from 15–50%—with versions of our benchmarks in which global calls to user-defined procedures had been converted into local calls by wrapping each benchmark in a `let` expression. Once again, the bootstrapped compiler benefited from its own optimizations, as the compiler became about 25% faster on average.

In retrospect, these optimizations don't seem very difficult, and the students in my graduate compiler class are able to implement

them over the course of a couple of weekly assignments. At the time, however, I found them extremely challenging. Of course, I didn't have an assignment description telling me how to proceed, and my intermediate language was more complex. Another reason for the difficulty is that I shoe-horned the optimizations into existing compiler passes, when I probably should have added one or more separate passes. Extra passes aren't as costly as I believed at the time, and the reduced complexity and relative ease with which the compiler could have been modified in later years would have more than made up for the modest compile time cost.

In 1993, I recruited one of my graduate students, Mike Ashley, to work with me on support for multiple return values. We wanted a mechanism that, as with our continuation mechanism, pays its own way in the sense that the efficiency of normal, single-value returns and return points is unaffected. We also wanted the mechanism to signal an error whenever the wrong number of values is received. We were able to accomplish both goals with a mechanism that handles both multiple- and single-value returns and return points efficiently [2].

I also recruited another of my graduate students, Oscar Waddell, to port Chez Scheme to the Alpha processor running Digital's OSF/1 operating system. This was the first of many projects that Oscar and I worked on together. In addition to the Alpha port itself, the most useful thing to come out of this project was the elimination of assembly code from the system source. We had been using a combination of machine-dependent m4 [40] macros and mostly machine-independent assembly files to reduce the amount of assembly code involved in porting to a new architecture, but this was difficult to work with and did not completely relieve us from having to deal with the many idiosyncrasies of assembler syntax. We replaced the assembly code with code written in a machine-independent assembly-like language implemented by feeding code into our compiler's code generator.

One other Version 5 change is worth mentioning, for those who like simple but effective hacks. This particular hack reduced the cost of creating a generated symbol, or `gensym`, by a factor of 25. The hack is to delay generation of the gensym's name until the first time it's printed, so that `(gensym)` becomes a cheap inline allocation operation. Most gensyms created, e.g., during macro expansion, are never printed, so the savings are real and have a significant impact on programs that use gensyms heavily.

9. Chez Scheme Versions 6

System development continued at a rapid pace between Versions 5 and 6. We added support for tracking source-file information through the reader, expander, and compilation process so that the inspector can display original source code rather than expander output. We also added support for disjoint record types, one-shot continuations [4], modules [56], and one of my favorite new language features, datum comments. A datum comment, using the syntax `#;`, comments out an entire S-expression, so that, for example, `(a #;b c)` reads as `(a c)`. The need for such a thing occurred to me after watching people use quote marks to comment out top-level expressions and be frustrated when the same trick didn't always work away from the top level.

We also added support for calling Scheme from C, which set up the possibility of nested Scheme/C calls. We avoided difficulties with continuations by saving the C stack context (using `setjmp`) before calling into Scheme. This context is restored (using `longjmp`) on the first return from Scheme, at which point the context is also marked invalid along with any dynamically subordinate contexts. An attempt to return to an invalidated context causes an error to be signaled. This allows arbitrary continuation operations to be performed on the Scheme side as long as no attempt is made to return a second time to a C frame or to a frame nested dynamically within it on the C stack.

Release year: 1998

Language: R5RS compatibility, Petite Chez Scheme, datum comments, modules, source-file information tracked through compilation, profiling, disjoint record types, support for C calls into Scheme, continuation handling with nested C/Scheme calls, one-shot continuations

Implementation: fast interpreter, `letrec` optimization, procedure inlining, faster continuation handling, improved float printing

Documentation: *The Scheme Programming Language, 2nd Edition* [18], *Chez Scheme User's Guide* [19] (both documents available online), Unix manual page

Platforms: Sun Sparc SunOS/Solaris, DEC Alpha AXP OSF/1, HP PA-RISC HPUX, Motorola PowerPC AIX, Motorola PowerPC NT, SGI IRIX, Intel Linux, Intel Windows 95/NT

Version 6 Highlights

Bob Burger, another of my graduate students, was hired to port Chez Scheme to two new architectures, the HP PA-RISC architecture under HP/UX and the PowerPC architecture under AIX. I also incorporated several refinements that Bob made to a floating-point printer I had written in 1990 based on a paper by Guy Steele and Jon White [53]. Bob's improvements made the algorithm simpler and more efficient [5]. Bob's PhD research was on profile-driven dynamic recompilation [6], and though we have never adopted the dynamic recompilation support, we did incorporate Bob's profiling support into Chez Scheme with the help of Mike Ashley.

One of our goals for Version 6 was to provide a complete run-time system to simplify the delivery of compiled Chez Scheme programs. No run-time system for Scheme would be complete without `eval`, so we decided to include an interpreter that we had quietly slipped into the system back in Version 2 to help with cross compilation. With the interpreter and the rest of the run-time system, we had 99% of a working Scheme system, so we decided to include the read-eval-print loop and release the run-time system with interpreter as a complete, separate system. The system is built from the same sources as Chez Scheme, with only the compiler left out. We named the system Petite Chez Scheme, since without the compiler it is smaller than the system as a whole. We couldn't resist making a few tweaks to the interpreter to speed it up before releasing the system. We get a lot of positive feedback about the interpreter's speed, and people are often surprised to learn that it is written in Scheme. For some reason, they think an interpreter written in C should be faster, although an interpreter written in Scheme and compiled by a good Scheme compiler can leverage off of the built-in support for proper treatment of tail calls and continuations, as well as compiled library code.

Of course, as always, we also worked on improving the speed of compiled code. We had written a source optimizer many years earlier but abandoned it because it failed our performance test, i.e., did not make the bootstrapped compiler's code enough faster to overcome the extra passes involved. In early 1996, however, an advance copy of the paper, "Flow-directed inlining," by Suresh Jagannathan and Andrew Wright [38] spurred us into action. The paper was interesting because it claimed some impressive improvements in speed for a set of substantial Scheme programs. What made it especially interesting was that these improvements were accomplished with an optimizer that ran as a prepass to the Chez Scheme com-

piler. Some of the gains turned out to be due to the conversion of global calls into local calls, triggering the local-call optimizations described in Section 8, but even after adjusting for this, the results were too good to ignore. While the compile-time cost of their analysis was impractically large for our compiler, we now knew that good gains were there to be had, if we could figure out how.

Mike Ashley had already been working on a flow analyzer of his own, and he adapted it to implement an algorithm similar to that of Jagannathan and Wright's. He was actually able to improve slightly on their results, but although his flow analyzer was substantially faster, it was still impractical for our purposes.

So Oscar Waddell and I set out to create our own inliner, with the constraint that it should be fast and linear. We hoped it would also yield results "in the same ballpark" as those that had been achieved with the flow-directed inliners. After months of work, we had an inliner that performed even better than we'd hoped, generating at least comparable code for all of the benchmarks and actually beating the flow-directed inliners soundly in a few cases [55]. One reason was the "on-line" nature of our algorithm, which allowed the inliner to make decisions based on subexpressions that it had already transformed, while the flow-directed inliners ran the analysis "off-line," i.e., entirely before any transformations occurred. Another reason was the approach we used in deciding when to inline and when not to. The inliner simply makes all inlining attempts and cuts them off when the size of the residual code or the time taken on the attempt exceeds a predetermined limit. Before settling on this no-nonsense approach, we tried several heuristics to limit code size and inlining time, but heuristics inevitably inhibit or allow more inlining than they should. We incorporated the inliner into the Chez Scheme compiler and also use it as an interpreter prepass in Petite Chez Scheme.

During this time, Oscar Waddell had been tapped to lead the development of a GUI API to support the use of Scheme in teaching as part of an NSF-supported educational infrastructure project. The initial name for this system was "Bob." I think this was a takeoff on the short-lived Microsoft product of the same name, but I'm not sure. In any case, the name didn't stick, and the name Scheme Widget Library was adopted instead. Abbreviated SWL and pronounced "swill," the name still reflected Oscar's poor opinion of the system in the early going. The system turned out really nice in the end, however, and includes an interactive development environment for Scheme as well as tools to build graphical applications. Oscar put in the bulk of the work, with several other people contributing as well, especially Carl Bruggeman, Bob Burger, Erik Hilsdale, and John Zuckerman. John LaLonde of Abstrax, Inc., also supported the project, and we borrowed some ideas from a system he had written some years earlier while working at Motorola. The system is now distributed with Chez Scheme, and I have made a few minor contributions of my own. Many programmers still use Chez Scheme with `emacs`, and true power users like me use it with `vi`, but many others use SWL as their interface to Chez Scheme or Petite Chez Scheme, especially under Windows and the Mac.

10. Chez Scheme Version 7

Although many years passed between the releases of Version 6 and Version 7, we were not idle and put out several minor releases. We added a variety of new features, like logical operations on bignums, file compression, support for Scheme shell scripts including script compilation, Windows registry primitives, `eq?` hash tables, and `apropos`. We extended gensyms to allow the creation of globally unique identifiers, added record inheritance, and added support for automatically loading heap and boot files based on application executable names. In a fit of madness, I also implemented Common Lisp `format`, which is potentially quite useful, but even after implementing it I can never remember which directive is which and especially which combinations of colon and at-sign modifiers to use to accomplish a particular task. I had intended to implement something less elaborate but didn't know where to stop.

Release year: 2005

Language: multithreading, `expand/optimize`, `letrec*`, `apropos`, `meta` definitions, `import` extensions, `eq?` hash tables, heap/boot search paths, registry primitives, `format`, support for scripts, `compile-script`, record inheritance, nongenerative record types, `let-values`, bignum logical operations, generalized `syntax-case` patterns, `visit`/`revisit`, more foreign/external interface improvements, file compression, gensyms UIDs

Implementation: thread support, incompatible record checks, `letrec`/`letrec*` violation checks [57], compile-time format-string checks and compilation, `unquote` and `unquote-splicing` extension to zero+ subforms, various run-time library and code generation improvements, `syntax-case` performance improvements, improved bignum arithmetic speed

Documentation: *The Scheme Programming Language, 3nd Edition* [20], *Chez Scheme Version 7 User's Guide* [21] (both documents available online), Unix manual page

Supported machines: Sparc Solaris (32- and 64-bit, threaded and nonthreaded), Intel 80x86 Linux (threaded and nonthreaded), Intel 80x86 Windows 95/98ME/NT/2000/XP (threaded and nonthreaded), Apple PowerPC MacOS X (nonthreaded)

Version 7 Highlights

In addition to these relatively minor changes, Oscar and I ported the system for the first time to a 64-bit architecture and created a multithreaded version of the system based on Posix Threads that allows applications to make use of multiple processors and processor cores. The 64-bit port, which targeted the 64-bit Sparc architecture, should have been relatively straightforward, but we were tripped up by a number of 32-bit dependencies. You might think someone who experienced the transition from 16-bit microprocessor address spaces to 32-bit address spaces would have avoided such dependencies, but of course, I never dreamed Chez Scheme would be around long enough for this to be a concern. At least we're ready now for the transition to 128-bit address spaces.

The thread system posed numerous challenges, particularly in making portions of the system thread-safe. Fortunately, several of our earlier design decisions simplified other aspects of the project. Our decision to stick with BiBOP and segmented memory made doling out individual allocation areas to each thread trivial. Individual allocation areas are essential to avoid synchronizing on every allocation operation, which would be a performance disaster. Our segmented representation of stacks also helped, since the stack-overflow recovery mechanism allows each thread to start with a small stack that grows by adding new segments as needed. Our model for initiating collections also helped. Collections are triggered via an interrupt, and the interrupt is set by the segment-level allocator based on the number of bytes that have been allocated since the last collection. This generalized nicely to allow multiple threads to synchronize for collection. When a thread receives the collect interrupt and other threads are still active, it waits for the other threads. When the last active thread synchronizes, blocks, or exits, the collection is initiated.

Tracking assignments into older generations turned out to be a problem, however. Generational collectors must be informed when-

ever a location within an older-generation object contains a pointer to a younger generation so that it can be traced during a collection of the younger generation. Our collector uses a modified card-marking [50] system in which this information is stored in a global table. Unfortunately, storing the information is not an atomic operation, and synchronizing on every mutation operation is too expensive, even if one believes programs should do without mutation whenever possible. We addressed this issue by maintaining a log of mutated locations at the end of the local allocation area, with the log pointer growing toward the allocation pointer, as the second version of the Z80 Scheme system had done for the stack and allocation pointers. Synchronization need not occur until the log and allocation pointers meet up, at which point the logged entries are scanned to record information about pointers from older to younger generations, and a new allocation area is obtained if necessary.

11. Parting remarks

Developing a system like Chez Scheme is a process of successive refinement. Even had I known in 1983 what I know today, it would still have been unimaginably difficult to create Chez Scheme, as it exists today, in one continuous development effort. Had I tried anything like that, I would likely have abandoned the effort before it was fairly begun. Instead, I started with a simple model for representing Scheme values, a small compiler that performed little optimization, a simple stop-and-copy garbage collector, and a small run-time system. From there it grew, and shrank, as features were added, extended, removed, and replaced.

Of course, for this decades-long development effort to succeed, we have had to do some often unpleasant things, like discard code we thought was clever, rewrite code we hoped never to look at again, fix bugs promptly even when inconvenient, and extend the test suite whenever a new feature is added. Had we not done so, the system would have become filled with useless, crufty, buggy, and untested code, making it difficult to maintain and extend, and making even the fun parts of the job, like writing new code, unpleasant.

Our priorities have become more refined over time as well, especially where efficiency is concerned. I've always believed that efficiency is not merely a matter of the speed of compiled code, but also a matter of compile time, memory usage, and overall system responsiveness. Over the years, I have come to believe that uniformity and continuity of performance are important as well. I have also become much more concerned about handling large programs or programs that operate on large amounts of data. Beyond efficiency and reliability, additional priorities have emerged, such as standards conformance, ability to interact gracefully with code written in other languages, and overall system usability.

Even after two decades of refinement, the system is no where near what I'd like it to be. There are numerous ways in which the performance can be improved, numerous places where the code could be cleaner, and numerous features I'd like to add or extend. Our to-do list has hundreds of entries, ranging from straightforward chores to research projects. I'm not complaining. If the to-do list is ever empty, I'll know it's time to pack it in.

Acknowledgments

In addition to the people, mentioned in this paper, who have worked directly on Chez Scheme, many more, too numerous to mention, have supported the development in other ways. Without their contributions, Chez Scheme would not be what it is today. This paper would not exist were it not for the kind invitation from Julia Lawall and the rest of program committee to speak on this topic at ICFP '06. Suggestions by Oscar Waddell, Julia Lawall, Susan Dybvig, Dan Friedman, and Olivier Danvy led to many improvements in the paper.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, MA, USA, 1985.

[2] J. Michael Ashley and R. Kent Dybvig. An efficient implementation of multiple return values in Scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 140–149, June 1994.

[3] G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979.

[4] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, May 1996.

[5] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, May 1996.

[6] Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proceedings of the IEEE Computer Society 1998 International Conference on Computer Languages*, pages 240–251, May 1998.

[7] Robert G. Burger, Oscar Waddell, and R. Kent Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138, June 1995.

[8] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme System Manual*, August 1989.

[9] Cadence Research Systems, Bloomington, Indiana. *Chez Scheme System Manual, Rev. 2.0*, December 1990.

[10] Cadence Research Systems. *Chez Scheme System Manual, Rev. 2.5*, October 1994.

[11] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, New York, NY, USA, 1984. ACM Press.

[12] Gregory J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 ACM SIGPLAN symposium on Compiler construction*, pages 98–105, New York, NY, USA, 1982. ACM Press.

[13] Rex A. Dwyer and R. Kent Dybvig. A SCHEME for distributed processes. Computer Science Department Technical Report #107, Indiana University, Bloomington, Indiana, April 1981.

[14] R. Kent Dybvig. C-Scheme. Master's thesis, Indiana University Computer Science Department Technical Report #149, 1983.

[15] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[16] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina Technical Report #87-011, Chapel Hill, April 1987.

[17] R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report 356, Indiana University, June 1992.

[18] R. Kent Dybvig. *The Scheme Programming Language*. Prentice Hall, second edition, 1996.

[19] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[20] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2003.

[21] R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[22] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 207–216, June 1993.

[23] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BiBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report 400, Indiana Computer Science Department, March 1994.

[24] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 143–150, 1986.

[25] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[26] R. Kent Dybvig and Robert Hieb. A variable-arity procedural interface. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 106–115, July 1988.

[27] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[28] R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, September 1990.

[29] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[30] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-driven code generation. Technical Report 302, Indiana Computer Science Department, February 1990.

[31] R. Kent Dybvig and Bruce T. Smith. *Chez Scheme Reference Manual, Version 1.0*. Cadence Research Systems, Chapel Hill, North Carolina, May 1985.

[32] Robert E. Filman and Daniel P. Friedman. *Coordinated computing: tools and techniques for distributed software*. McGraw-Hill, Inc., New York, NY, USA, 1984.

[33] John K. Foderaro, Keith L. Sklower, and Kevin Layer. *The Franz LISP Manual*. University of California, Berkeley, 1983.

[34] Per Brinch Hansen. Distributed processes: a concurrent programming concept. *Communications of the ACM*, 21(11):934–941, 1978.

[35] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[36] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[37] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Syntactic abstraction in Scheme. Technical Report 355, Indiana University, June 1992.

[38] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, 1996.

[39] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[40] Brian W. Kernighan and Dennis M. Ritchie. *The M4 Macro Processor*, 1979.

[41] Eugene Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, August 1986.

[42] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[43] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[44] Gyula Mago. A cellular computer architecture for functional programming. In *Proc. COMPCON Spring, IEEE Comp. Soc. Conf.*, pages 179–187, 1980.

[45] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Mass., 1966. second edition.

[46] Brian Randell and Lawford J. Russell. *ALGOL 60 Implementation*. Academic Press, London, 1964.

[47] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of Lisp or LAMBDA: The ultimate software tool. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 114–122, New York, NY, USA, 1982. ACM Press.

[48] Digital Research. *CP/M Operating System Manual*. Pacific Grove, CA, USA, 1976.

[49] Robert A. Saunders. The LISP system for the q-32 computer. In Edmund C. Berkeley and Daniel G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications.* Information International, Inc. and MIT Press, 1964.

[50] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. Thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science Department, Cambridge, MA., September 1988.

[51] Guy L. Steele Jr. Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, September 1977.

[52] Guy L. Steele Jr. and Gerald J. Sussman. The revised report on Scheme, a dialect of Lisp. MIT AI Memo 452, Massachusetts Institute of Technology, January 1978.

[53] Guy L. Steele Jr. and Jon L White. How to print floating-point numbers accurately. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):112–126, June 1990.

[54] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167, New York, NY, USA, 1984. ACM Press.

[55] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In *Proceedings of the Fourth International Symposium on Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, September 1997.

[56] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of the Twenty Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 203–213, January 1999.

[57] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-order and and symbolic computation*, 18(3/4):299–326, 2005.