Chez Scheme Version 4 Release Notes

September 1991 (revised 3/2/92)

Overview

This document outlines the changes made to *Chez Scheme* for Version 4 since Version 3. This version includes the following enhancements:

- a new object inspector with support for stack walkbacks,
- complete support for complex numbers,
- full implementation of and comformance to the IEEE Scheme Standard,
- major improvements in performance,
- generational garbage collection,
- support for Decstation and Decsystem Mips-based computers, and
- support for Silicon Graphics Mips-based computers.

Overall performance has increased by around 50 percent over Version 3 at all optimization levels. Some programs are likely to run little or no faster (notably I/O intensive programs), while others may run as much as two or three times faster. Floating-point intensive programs are typically about two times faster. Compile time is approximately 30 percent faster.

Refer to the *Chez Scheme System Manual*, Revision 2.2 for detailed descriptions of the new or modified procedures and syntactic forms mentioned in these notes.

Version 4 is available for the following platforms:

- Decstation and Decsystem computers
- Motorola Delta Series MC680X0-based computers
- Motorola Delta Series MC88000-based computers
- NeXT (Mach 2.0 or higher) computers
- Silicon Graphics (IRIX 3.3.2 or higher) computers
- Sun-3 (SunOS 3.X, 4.X) computers
- Sun-4, Sparcstation, and Sparcserver (SunOS 4.X) computers
- Vax (Ultrix 2.0 or higher)

This document contains four sections describing (1) functionality enhancements, (2) performance enhancements, (3) bugs fixed, and (4) compatibility issues.

1. Functionality Enhancements Since Version 3

1.1. Inspector

An object inspector has been added to the system. By default, interrupts and explicit calls to `break` enter a break handler. The break handler is also entered whenever `debug` is invoked after an error occurs. Once inside the break handler, it is possible to reset to the current cafe (read-eval-print loop), enter a new cafe, obtain statistics for the interrupted computation, exit from the handler and continue, abort Scheme, or inspect the continuation of the interrupted computation. The inspector allows the programmer to "walk" back through the continuation to see what procedures were active at the point where the computation was interrupted. The inspector provides information about variable names and macro-expanded source code in continuations and procedures unless the parameter `generate-inspector-information` is set to `#f`.

The inspector also allows the programmer to examine circular objects, objects such as ports and procedures that do not have a print syntax, and objects such as variables that are not otherwise accessible, as well as ordinary printable Scheme objects. It can be invoked outside of the break handler via the single-argument procedure `inspect`.

1.2. Complex numbers

Chez Scheme now supports all standard operations on complex numbers. These operations include the procedures `make-rectangular` and `make-polar`, which are used to construct complex numbers, and `real-part`, `imag-part`, `magnitude`, and `angle`, which are used to decompose complex numbers. In addition, all appropriate numeric operators have been extended to handle complex arguments.

The *Chez Scheme* reader and printer as well as `string->number` and `number->string` have been extended to read and write complex numbers according to the syntax described in the IEEE Scheme Standard and Revised^4 Report on Scheme.

Two additional procedures, `magnitude-squared` and `conjugate`, have also been added.

Inexact complex numbers are represented by inexact complexnums, which contain two 64-bit floating point numbers, and exact complex numbers are represented by exact complexnums, which contain two exact ratios or integers.

1.3. IEEE Scheme Standard and Revised^4 Report

Chez Scheme fully implements all features of and conforms fully to the IEEE Standard for Scheme as documented in IEEE Std 1178–1990, *IEEE Standard for the Scheme Programming Language.*

As stated in the IEEE Scheme Standard, "Conforming implementations may have extensions, provided they do not alter the behavior of any conforming program." Since Chez Scheme defines several nonstandard syntactic forms that may interfere with conforming programs,[1] a "subset mode" has been implemented that disables these syntactic forms and allows any conforming program to run. The application `(subset-mode 'ieee)` will place Chez Scheme into this mode.

Chez Scheme also implements *all* required *and* optional features of the Revised^4 Report on Scheme. The application `(subset-mode 'r4rs)` will place Chez Scheme into a mode similar to the ieee mode that supports only those syntactic extensions documented in the Revised^4 Report.

The application `(subset-mode #f)` returns the system to its default state.

1.4. Limits lifted or eliminated

The range of fixnums (integers directly encoded in *Chez Scheme* pointers) has been increased from $-2^{18}..+2^{18}-1$ to $-2^{29}..+2^{29}-1$, resulting in improved performance for computations in the extended range.

More importantly, string and vector lengths are limited only by the range of nonnegative fixnums. Because the range of nonnegative fixnums has been increased to $0..2^{29}-1$, the maximum size of a string or vector is now effectively limited only by available virtual memory. The same is true for the maximum size of a bignum, which is represented internally as a sequence of words.

The maximum size of the code generated for a procedure was previously limited to 64K bytes; as for strings, vectors, and bignums, this limit has been removed. Various restrictions enforced by the assemblers on branch displacement and similar displacements have been removed as well.

An internal limitation of 32MB for the size of the Scheme image has been eliminated, so the maximum size of a Scheme image is now limited only by available virtual memory, *i.e.*, by operating system per-process limits and available swap space.

1.5. Distinct empty list and false objects

Chez Scheme now recognizes a distinction between the empty list (`()`) and the false object (`#f`). One consequence of this change is that `(eq? '() #f)` evaluates to `#f`. Furthermore, the empty list no longer serves as boolean false in conditional expressions, so, for example, `(if (cdr '(a)) 'yes 'no)` evaluates to `yes`.

1.6. Void object

A new void object has been introduced. This object is returned as the result of most operations that return an "unspecified" value. It is obtainable directly by invoking the procedure `(void)`. The void object prints as

[1] Any complete implementation of Revised^4 Report Scheme will necessarily contain at least one such syntactic form, namely `delay`.

#<void>, but the default waiter-write procedure suppresses printing of the void object, so that assignments and calls to procedures such as load that return the void object do not result in any output to the console.

1.7. Specifying built-in identifier bindings

Built-in syntactic forms that expand into calls to primitives, such as case, which expands into calls to memv?, use the syntactic form (|#primitive| *identifier*) to specify that the original top-level value of *identifier* should be used rather than the current lexical or top-level value. (|#primitive| *identifier*) can also be written as #%*identifier*.

|#primitive| was chosen rather than something more readable so that it would not conflict with symbols that can appear in a IEEE Scheme Standard conforming program. (The vertical bars are not part of the name; |#primitive| is Chez Scheme's print syntax for (string->symbol "#primitive").)

1.8. Collection of code objects and symbols

Code objects and interned symbols are now eligible for collection by the storage management system. In prior versions, once compiled code for a procedure was generated or loaded from an object file, it was permanently resident in the system. Now, this code is released and collected by the garbage collector when it is no longer accessible.

Also, in prior versions, interned symbols were retained indefinitely, even if no pointers from outside the oblist (symbol table) to the symbol existed and the symbol had no top-level bindings or properties. Now, interned symbols are released when the system can safely do so whenever the oldest generation, the generation in which the oblist resides, is collected (see "Garbage Collection" under "Performance Enhancements").

1.9. New printer controls

Two new printer controls have been added, primarily to support portability between Chez Scheme and other Scheme systems. The parameter (see "Parameters") print-brackets determines whether or not the pretty printer uses brackets in place of parentheses for readability around, for instance, let bindings or cond clauses. When set to #f, brackets are not used.

The parameter print-vector-length controls whether the length of a vector is printed, *e.g.*, #3(1 2 3), or omited, *e.g.*, #(1 2 3). If #f, the length is omited.

The initial value for both parameters is #t. Both are set to #f by subset-mode when the argument is either ieee or r4rs (see "IEEE Scheme Standard and Revised^4 Report").

1.10. Graph printing

The reader now accepts out-of-sequence graph references and marks. For example, '(#1# #1=(a)) now results in the same object as '(#1=(a) #1#). The printer now prints marks in the most "natural" order, *e.g.*, marks always come before references, and marks always appear in increasing order from left to right (or top to bottom) in the output. For example, '(#0=(a) #1=(b) #1# #0#) will print as (#0=(a) #1=(b) #1# #0#), not as, say, (#1=(a) #0=(b) #0# #1#) or (#1# #0=(b) #0# #1=(a)), although each of these forms are recognized as the same object by read.

The printer now enters graph printing mode temporarily when a cycle is detected that would otherwise cause infinite output.

1.11. Shared structure in compiled files

Shared structure (including cycles) within quoted objects is now maintained as such through the compilation and subsequent loading process by compile-file and load.

1.12. Numeric type predicates

The numeric type predicates integer?, rational?, and real? have changed to adhere to the IEEE Standard for Scheme and the Revised^4 Report and to accommodate the addition of complex numbers. The integer? predicate now returns true if its argument is an exact *or inexact* integer. For example, (integer? 3.0) and (integer? 3.0+0.0i) now return #t. The rational? predicate now returns true if its argument is an exact or inexact rational number. Furthermore, since *Chez Scheme* has no representation for irrational numbers, this means that real? and rational? now behave the same. For example, (rational? 3.2) and (real? 3.2) both return #t. The complex? and number? predicates continue to return #t for all numeric types.

The exact? and inexact? predicates have been extended to handle complex numbers: Also, both predicates now signal an error for any nonnumeric argument type.

1.13. Exactness

The following procedures are now "exactness preserving" (or more accurately, "inexactness preserving"), to adhere to the IEEE Scheme Standard and the Revised^4 Report on Scheme: quotient, truncate, floor, ceiling, min, max, and rationalize.

Also, the following procedures, which were previously defined only for exact integer or rational arguments, are now defined for inexact integer or rational arguments as well: numerator, denominator, even?, odd?, gcd, lcm, and expt-mod.

1.14. IEEE signed zero, infinities, and not-a-number

On machines that support IEEE floating point arithmetic, Chez Scheme now properly reads, prints, and maintains signed zero, infinities, and NaNs. The syntax -0.0 produces a negative zero internally that prints as -0.0 on output. Similarly, the syntaxes -inf.0, +inf.0, and +nan.0 are used to represent positive and negative infinity and IEEE NaN. Many floating point operations have been carefully coded to handle these values in the spirit of the IEEE floating point standard.

There is a slight inconsistency in our treatment, in that inexact real numbers are represented as flonums with an implicit exact zero imaginary part, whereas inexact imaginary numbers (such as +1.0i) are represented as complexnums with an inexact positive zero real part.

1.15. Printing of flonums

Inexact numbers, represented internally as flonums, are now printed with complete accuracy and with the fewest number of significant digits possible.[2] One consequence of the increased accuracy is that flonums printed by the system are always read as the same number on any other machine with the same internal representation for floating point numbers.

1.16. Type-specific arithmetic operators

Chez Scheme Version 4 includes support for various new fixnum-specific, flonum-specific and mixed flonum and inexact complexnum operators, in addition to the set of fixnum-specific operators supported by Version 3. The new operators include the type predicates flonum?, which returns #t if its argument is a flonum and #f otherwise, and cflonum?, which returns #t if its argument is a flonum or inexact complexnum. (This differs from inexact? in that inexact? signals an error if its argument is not a number, whereas cflonum? simply returns #f.)

The new fixnum operators are fxsll (shift left logical), fxsra (shift right arithmetic), and fxsrl (shift right logical).

The new flonum operators are fl+, fl-, fl*, fl/, fl=, fl<, fl<=, fl>, fl>=, and flabs.

The flonum/complexnum operators accept mixed flonum and inexact complexnum arguments, and return either booleans, flonums or inexact complexnums. The new flonum/complexnum operators are cfl-real-part, cfl-imag-part, cfl=, cfl+, cfl-, cfl*, cfl/, cfl-conjugate, and cfl-magnitude-squared.

The procedure fixnum->flonum may be used to convert a fixnum into a flonum, and the procedure fl-make-rectangular may be used to create a complexnum from two flonum arguments.

[2] These are requirements of the IEEE Scheme Standard but, interestingly, not of the IEEE floating point standard.

1.17. Waiter customization

Chez Scheme waiters may be customized via the parameters `waiter-prompt-string`, `waiter-prompt-and-read`, and `waiter-write`. Also, the ports used by the waiter for input and output may be changed by altering the values of the parameters `console-input-port` and `console-output-port`. The action taken by the system when a reset, exit, or abort occurs within a cafe can also be modified via the parameters `abort-handler`, `exit-handler`, and `reset-handler`.

1.18. Transcripts

The procedure `transcript-on` no longer enters a new cafe, but instead simply opens a transcript file and enters a transcript mode where all input and output to the console are recorded onto the file. The procedure `transcript-off` must be used to terminate the transcript mode and close the transcript file.

The procedure `transcript-cafe` behaves like the old `transcript-on`, *i.e.*, it starts up a new cafe and records all console input and output activity to the transcript file until the cafe is exited. Transcript-off may be used to terminate transcript mode and close the transcript file from within the new cafe, but it no longer exits from the new cafe. Invoking `exit` or typing the end-of-file character will exit from the cafe, terminate transcript mode, and close the transcript file.

1.19. `list?` predicate

The procedure `list?` has been altered to match the new Revised[4] Report and IEEE Standard definition. It now returns `#t` if its argument is a proper list, *i.e.*, if it is noncircular and terminated by the empty list. Otherwise, it returns `#f`.

1.20. `oblist` procedure

A new procedure, `oblist`, has been added that returns a list of the interned symbols currently known to the Scheme system. The variable `*oblist*`, which previously held a pointer to the systems internal oblist, is no longer bound in the initial heap.

1.21. `property-list` procedure

A new procedure, `property-list`, returns a list representing the properties associated (via `putprop` with its symbol argument. A property list is structured as an alternating list of keys and values, (*key value* ...).

1.22. `isqrt` procedure

A new procedure, `isqrt`, has been added that returns the integer square root of its integer argument.

1.23. Top-Level values for uninterned symbols

The compiler now allows uninterned symbols (created with `string->uninterned-symbol` and `gensym`) to have top-level values. In previous versions, the compiler would signal an error ("cannot reference uninterned symbol" or "cannot set! uninterned symbol") whenever a reference or assignment to an uninterned symbol was detected in the source code.

1.24. Warnings

Warning messages may now be issued by the compiler or run-time system via a warning mechanism essentially similar to the error mechanism. After a warning message is issued, the default behavior of the system is to continue. Warnings are signaled by user code via the procedure `warning`, whose interface is identical to that of `error`.

The behavior of the system when a warning is issued may be changed by defining a new warning handler. Warning handlers have the same interface as error handlers. A new warning handler may be defined by altering the value of the parameter `warning-handler`. For example, to cause warnings to be treated as errors, define the warning handler to be the same as the error handler as follows:

```
(warning-handler (error-handler))
```

1.25. Foreign procedure prototypes

The `foreign-procedure` syntactic form now evaluates the foreign name subexpression. Previously, the foreign name was syntactically restricted to be a string, *e.g.*, `(foreign-procedure "incr" (integer-32 integer-32) integer-32)`. The generalization allows foreign procedure *prototypes* to be defined. For example:

```
(define foreign-intXint->int
  (lambda (x)
    (foreign-procedure x (integer-32 integer-32) integer-32)))
(define incr (foreign-intXint->int "incr"))
(define decr (foreign-intXint->int "decr"))
```

In the example, two foreign procedures are constructed from the same prototype. (The strings `"incr"` and `"decr"` are assumed to name C procedures already loaded or linked into the Scheme image.)

Foreign procedure prototypes provide a simple abstraction mechanism, and they can result in less code generation as well, since a single foreign-procedure body can be shared by many foreign procedures.

1.26. `file-position` and `file-length` procedures

Two new procedures operating on ports have been added: `file-position` and `file-length`.

The procedure `file-length` accepts one argument, which must be a port, and returns the current length in bytes of the file (or string for string ports) to which the port refers.

The procedure `file-position` accepts one or two arguments. The first argument must be a port, and the second, if present, must be a nonnegative integer. When the second argument is omitted, `file-position` returns the position of the port in the file (or string for string ports) to which the port refers, measured in bytes from the start of the file (or string). When passed two arguments, `file-position` sets the file position to the value of the second argument.

1.27. Parameters

All user-modifiable *Chez Scheme* system variables, such as `*print-length*`, have been converted to *parameters*. A parameter is a procedure that encapsulates a single variable. When invoked without arguments, a parameter returns the value of its encapsulated variable. When invoked with one argument, the parameter changes the value of its encapsulated variable to the value of its argument. A parameter may signal an error if its argument is not appropriate.

New parameters may be created with `make-parameter`, a procedure of one or two arguments. The first argument is the initial value of the internal variable, and the second, if present, is a *filter* applied to the initial value and all subsequent values. The filter should accept one argument, signal an error if the value of the argument is not appropriate, and return the value if it is appropriate.

For example, `(print-length)` is defined in the system as follows:

```
(define print-length
  (make-parameter
    #f
    (lambda (x)
      (unless (or (not x) (and (fixnum? x) (fx>= x 0)))
        (error 'print-length "~s is not a positive fixnum or #f" x))
      x)))
```

```
(print-length) ⇒ #f
(print-length 3)
(print-length) ⇒ 3
```

It is also possible to temporarily change the value of a parameter in a manner analogous to `fluid-let` for ordinary variables, using the new syntactic form `parameterize`. For example, the expression below:

```
(parameterize ([print-length 3] [print-level 3])
  (format "~s" '((a (b (c)) d) ((e f) g) h i j k)))
```

evaluates to `"((a (b (...)) d) ((e f) g) h ...)"`.

The following system variables have been converted to parameters:

old variable name	new parameter name
`*case-sensitive?*`	`case-sensitive`
`*collect-notify*`	`collect-notify`
`*collect-request-handler*`	`collect-request-handler`
`*collect-trip-bytes*`	`collect-trip-bytes`
`*eval*`	`current-eval`
`*standard-input*`	`current-input-port`
`*standard-output*`	`current-output-port`
`*error-handler*`	`error-handler`
`*gensym-count*`	`gensym-count`
`*gensym-prefix*`	`gensym-prefix`
`*keyboard-interrupt-handler*`	`keyboard-interrupt-handler`
`*optimize-level*`	`optimize-level`
`*pretty-line-length*`	`pretty-line-length`
`*pretty-one-line-limit*`	`pretty-one-line-limit`
`*print-gensym*`	`print-gensym`
`*print-graph*`	`print-graph`
`*print-length*`	`print-length`
`*print-level*`	`print-level`
`*print-radix*`	`print-radix`
`*timer-interrupt-handler*`	`timer-interrupt-handler`

The following system variables have been converted to *read-only* parameters; they can be invoked only without arguments:

old variable name	new parameter name
`*most-negative-fixnum*`	`most-negative-fixnum`
`*most-positive-fixnum*`	`most-positive-fixnum`

The `random-seed` procedure has been converted into a parameter, and the `set-random-seed!` procedure has been eliminated.

The `default-foreign-libraries` procedure has also been converted into a parameter, which can be used to access or change the default list of libraries.

The following parameters have been added:

- `abort-handler`, `exit-handler`, `reset-handler`: see "Waiter customization."
- `break-handler`: see "Inspector."
- `console-input-port` and `console-output-port`: see "Waiter customization."
- `current-directory` (extends and replaces `set-working-directory`).
- `generate-inspector-information`: see "Inspector."
- `machine-type` (read-only): returns the current machine type, used as the default value for the third argument to compile-file.
- `print-brackets` and `print-vector-length`: see "New printer controls."
- `subset-mode`: see "IEEE Scheme Standard and Revised^4 Report."
- `warning-handler`: see "Warnings."
- `waiter-prompt-and-read`, `waiter-prompt-string`, and `waiter-write`: see "Waiter customization."

1.28. Support for Decstation/Decsystem computers

Version 4 includes support for Digital Equipment Corporation Decstation and Decsystem computers, including the Decstation 2100 and 3100 and the Decsystem 5000 series.

1.29. Support for Silicon Graphics computers

Version 4 includes support for Silicon Graphics Mips-based computers running IRIX 3.3.2 or higher.

1.30. Support for NeXT computers

Version 4 includes support for NeXT mc68040- and mc68030-based computer systems under NeXT Mach 2.0. Although it is probable that the mc68040 executable images will also run on NeXT mc68030-based systems running NeXT Mach 2.0, this has not been tested.

Dynamic loading of foreign object code is not supported by the NeXT version; however, foreign code can be included in a *Chez Scheme* image by rebuilding the system in the "custom" directory.

No interfaces to Objective C or the Application Kit are provided or envisioned at this time.

2. Performance Enhancements Since Version 3

2.1. Compiler optimizations

Various compiler optimizations have been added for Version 4, including recognition of more built-in primitives, open coding of allocation operations, reordering of application expressions to avoid use of temporary locations, reduction in the average size of a closure, and improved instruction selection. We have completely redesigned the low-level representations for objects and pointers, and many of the optimizations we have added are directly related to this redesign.

2.2. Type checking

The speed of most type checking operations is now considerably faster. Many operations require nothing more than a register mask, compare, and branch; other operations require a single memory reference. Previously, nearly all type checks required computing an index into a table and a memory reference to pull the type out of the table.

2.3. Allocation

The speed of allocation operations, even for those that are not open coded (see "Compiler optimizations") has been improved by dedicating two machine registers to the current allocation pointer and current end of allocation pointer. Previously, allocation required several memory references just to obtain heap space for an allocated object.

2.4. Garbage collection

The garbage collector has been converted from a relatively simple stop-and-copy algorithm to a significantly more complex generational algorithm. The generational algorithm allows objects that survive more than a few garbage collections to migrate into an area of memory that is infrequently considered for collection. As a result, the average time it takes to perform a garbage collection has been reduced, especially for programs that create and retain a large number of heap allocated structures.

2.5. Floating Point Arithmetic

Floating point intensive programs coded with generic operators run approximately two times faster in Version 4. Even greater improvements are possible when using the new flonum- or complexnum-specific operators. In addition, the allocation overhead for floating point-intensive computations has been cut in half.

2.6. Register allocation for procedure arguments

In previous releases, the return address and arguments for all procedures were passed on the stack. In Version 4, the return address and some of the arguments are passed in registers, along with the closure pointer, which was always passed in a register.

2.7. Literal references

References to quoted objects are now placed directly into the code stream. In previous versions, a reference to a quoted object (such as a list) required a register-offset indirect load from the current closure. One side benefit of this change is that closures for procedures with no free variables are never allocated at run time, even if they contain references to quoted objects.

2.8. Optimize-level 2 fixnum operations

Inline code for various fixnum operations, *e.g.*, `fx=` and `fx+`, is now produced at optimize level 2. Previously, fixnum operations were coded inline only at optimize level 3.

2.9. 68020 fixnum operations

On 68020-based systems, fixnum multiply and divide operations (`fx*`, `fx/`, and `fxquotient`), which were previously coded as calls to library routines, are now open coded using the 68020's multiply and divide instructions.

3. Bugs Fixed Since Version 3

3.1. `with` and `quasiquote`

The release notes for Version 3 incorrectly claim that `with` and `quasiquote` can be used together; now this is actually true.

3.2. `with` expansion bug

A bug that caused the expansion of `with` expressions to require time proportional to the square of the number of clauses has been fixed.

3.3. `set-timer` bug

A bug in `set-timer` that could sometimes result in interrupts being disabled permanently has been fixed.

3.4. `foreign-procedure` bug

A bug in `foreign-procedure` that could cause interrupts to be ignored when the return value is a double float has been fixed.

3.5. Sun-4 return address bug

A Sun-4 code generation bug that caused strange behavior when very large `lambda` expressions were compiled has been fixed.

3.6. Console output bug

A bug that resulted in the last line of output to the console before the system exits being dropped if it was not followed by a newline or explicitly flushed has been fixed.

3.7. `exact->inexact` bug

A bug in `exact->inexact`, which manifested itself in the reading of certain floating point numbers, has been fixed. The bug involved converting exact ratios into inexact (floating point) numbers.

3.8. `process` and nonexistent commands

A bug that caused the system to exit after writing to a nonexistent process through a process output port has been fixed.

3.9. `dynamic-wind` error messages

The error message resulting from passing `dynamic-wind` something other than a procedure for its second or third argument has been fixed to report the correct offending object.

3.10. `dynamic-wind` *out* continuation bug

In Version 3, signaling an error or invoking a continuation from within the *out* argument to `dynamic-wind` would result in repeated illegal instruction traps or memory faults. This bug has been corrected.

3.11. First-class environment bug

Occurrences of `(the-environment)` (available after invoking `support-first-class-environments`) within the "else" part of an `if` expression sometimes caused invalid unbound variable errors. This bug has been fixed.

3.12. `explode` (SICP) bug

The procedure `explode` (available after invoking `support-sicp`) always signaled an error; this bug has been fixed.

3.13. Multiple entries and `provide-foreign-entries`

When given a list of more than one entry to provide, `provide-foreign-entries` would fail with a "ld: - : bad flag" error; this bug has been fixed.

3.14. Dynamic loading of foreign code under SunOS 4.1

Two bugs with dynamic loading of foreign code that arose due to changes in "ld" under SunOS 4.1 have been fixed. Under SunOS 4.1, *Chez Scheme* Version 3 sometimes aborts with a "heap request refused" message when there are linker errors during a call to `load-foreign`, and the detected errors go unreported. Also under SunOS 4.1, *Chez Scheme* Version 3 `provide-foreign-entries` fails with a "no such file or directory" error whenever the entry being provided is not already present in the *Chez Scheme* image.

3.15. `cond =>` bug

The `=>` syntax for a `cond` clause erroneously evaluated the conditional expression (the expression to the left of `=>`) twice when this expression yields a true value. For example,

```
(cond [(begin (write 'hello) #t) => (lambda (x) x)])
```

would generate `hellohello` on the current output port, rather than simply `hello`. This bug has been fixed.

3.16. Bug with `letrec` and `call/cc`

A subtle bug with the interaction between `call/cc` and `letrec` has been fixed. The bug caused:

```
(letrec ((v 0) (k (call/cc (lambda (x) x))))
   (set! v (+ v 1))
   (k (lambda (x) v)))
```

to return `2` instead of `1`.

3.17. `call-with-input-file` and `call-with-output-file`

The procedures `call-with-input-file` and `call-with-output-file` now close the ports they create immediately upon normal exit. In previous versions, the ports would be closed eventually by the garbage collector; however, this may not be soon enough if the port must be closed and Scheme's internal buffers flushed prior to some subsequent operation.

3.18. Zombie processes

Zombie processes (also called "defunct" or "exiting" processes), created when processes created by calls to `process` and `load-foreign` die, are now reaped automatically by Chez Scheme, via a Unix SIGCHLD signal handler.

4. Compatibility Issues between Version 3 and Version 4

Several changes have been made for Version 4 to bring it in line with recent changes to the Revised Report and IEEE Standard for Scheme, and to help users catch portability problems. Unfortunately, some programs that worked correctly in Version 3 will no longer work correctly in Version 4.

The change to distinguish the empty list from the false object is likely to cause the most difficulty. Bugs caused by this change are likely to come from either (a) specifying one of the constants `#f` or `()` where the other is actually desired, or (b) expecting the empty list to count as false in boolean expressions. To find problems caused by (a), look for occurrences of `#f` and `()`, and make sure that the appropriate constant is being used. To find problems caused by (b), look for the use of variables or applications of non-predicate procedures (especially `cdr`) in the test position of `if`, `cond`, `when`, `unless`, or `do` clauses to make sure the variable or procedure uses `#f` to denote false. Also, be sure to check `and` and `or` expressions as well.

The switch from top-level variables to *parameters* may cause some problems, although they will generally be easy to spot and easy to fix. Look for references to, assignments to, or fluid bindings for top-level variables whose names begin and end with asterisks, *e.g.*, `*print-length*`. Unless the variable is used for user-defined purposes, convert the reference, assignment, or fluid binding into a call or parameterization of the corresponding parameter.

The return values of various assignment operators, *e.g.*, `set!` and `set-car!`, have always been unspecified. However, in previous versions of *Chez Scheme*, these operators typically return the new value. For example, in Version 3, `(set! x 3)` returns `3`. In Version 4, most assignment operators return a special "void" object. As a result of this change, programs that actually use the value returned by an assignment operator will no longer function as expected. Look for uses of these operators, and make sure their value is not used.

The order in which the subexpressions of an application are evaluated is not specified by Scheme or *Chez Scheme*. Version 4 often chooses a different order of evaluation for a given application from that chosen by previous versions. As a result, programs that appear to be correct in Version 4 may not function as expected in other versions, and *vice-versa*. Look for applications and `let` expressions where a given order of evaluation is assumed, and introduce `let` expressions to enforce the desired order of evaluation. Also look for calls to map where the procedure is expected to be applied to the elements of the list or lists in a certain order.