

# Chez Scheme Version 8.4 Release Notes

## Copyright © 2011 Cadence Research Systems

### All Rights Reserved

### October 2011

## 1. Overview

This document outlines the changes made to *Chez Scheme* for Version 8.4 since Version 8.0, most of which are focused on converting *Chez Scheme* into an implementation of the new Scheme standard described in the Revised<sup>6</sup> Report on Scheme.

Version 8.4 is supported for the following platforms. The Chez Scheme machine type (returned by the `machine-type` procedure) is given in parentheses.

- Linux x86, nonthreaded (i3le) and threaded (ti3le)
- Linux x86\_64, nonthreaded (a6le) and threaded (ta6le)
- MacOS X x86, nonthreaded (i3osx) and threaded (ti3osx)
- MacOS X x86\_64, nonthreaded (a6osx) and threaded (ta6osx)
- Windows x86, nonthreaded (i3nt) and threaded (ti3nt)
- Windows x86\_64, nonthreaded (i3nt) and threaded (ti3nt) [experimental]
- OpenBSD x86, nonthreaded (i3ob) and threaded (ti3ob)
- OpenBSD x86\_64, nonthreaded (a6ob) and threaded (ta6ob)
- FreeBSD x86, nonthreaded (i3fb) and threaded (ti3fb)
- FreeBSD x86\_64, nonthreaded (a6fb) and threaded (ta6fb)
- NetBSD x86, nonthreaded (i3nb) and threaded (ti3nb)
- NetBSD x86\_64, nonthreaded (a6nb) and threaded (ta6nb)
- OpenSolaris x86, nonthreaded (i3s2) and threaded (ti3s2)
- OpenSolaris x86\_64, nonthreaded (a6s2) and threaded (ta6s2)

This document contains three sections describing significant (1) functionality changes, (2) bugs fixed, and (3) performance enhancements. A version number listed in parentheses in the header for a change indicates the first minor release or internal prerelease to support the change. Many other changes, too numerous to list, have been made to improve reliability, performance, and the quality of error messages and source information produced by the system.

More information on *Chez Scheme* and *Petite Chez Scheme* can be found at <http://www.scheme.com>, and extensive documentation is available in *The Scheme Programming Language, 4th edition* (available directly from MIT Press or from online and local retailers) and the *Chez Scheme Version 8 User's Guide*. Online versions of both books can be found at <http://www.scheme.com>.

## 2. Functionality Changes

### 2.1. `sleep` now deactivates calling thread (8.4)

The `sleep` procedure now deactivates the calling thread to allow collections to occur while it is asleep.

### 2.2. Experimental support for Windows x86\_64 (8.4)

Experimental support for running *Chez Scheme* with 64-bit pointers on the x86.64 architecture under Windows has been added, with machine types `a6nb` (nonthreaded) and `ta6nb` (threaded).

### 2.3. Annotations (8.4)

When source code is read from a file by `load`, `compile-file`, or variants of these, such as `load-library`, the reader attaches *annotations* to each object read from the file. These annotations identify the file and the position of the object within the file. Annotations are tracked through the compilation process and associated with compiled code at run time. The expander and compiler use the annotations to produce syntax errors and compiler warnings that identify the location of the offending form, and the inspector uses them to identify the locations of calls and procedure definitions.

While these annotations are usually maintained “behind the scenes,” Version 8.4 allows the programmer direct access to them via a set of routines for creating and accessing annotations.

The procedures that create, check for, and access annotations, source objects, and source-file descriptors are summarized below.

```
(make-annotation obj source-object obj) → annotation
(annotation? obj) → boolean
(annotation-expression annotation) → obj
(annotation-source annotation) → source-object
(annotation-stripped annotation) → obj

(make-source-object sfd uint uint) → source-object
(source-object? obj) → boolean
(source-object-bfp source-object) → uint
(source-object-efp source-object) → uint
(source-object-sfd source-object) → sfd

(make-source-file-descriptor string binary-input-port) → sfd
(make-source-file-descriptor string binary-input-port reset?) → sfd
(source-file-descriptor? obj) → boolean
(source-file-descriptor-checksum sfd) → obj
(source-file-descriptor-path sfd) → obj
```

Additional annotation-related procedures include the following

```
(get-datum/annotations textual-input-port sfd uint) → obj, uint
(open-source-file sfd) → #f or port
(syntax->annotation obj) → #f or annotation
```

All of procedures are described in detail in Section 11.11 of the User’s Guide.

Once read, an annotation can be passed to the expander, interpreter, or compiler. The procedures `sc-expand`, `interpret`, and `compile` all accept annotated or unannotated input.

The value of `current-expand`, which defaults to `sc-expand` and is used by `expand`, `compile`, `interpret`, `compile-file`, `compile-library`, `compile-program`, `compile-script`, and `compile-port` to expand input forms, must accept either annotated or unannotated input. So must the value of `current-eval`,

which defaults to `compile` (*Chez Scheme*) or `interpret` (*Petite Chez Scheme*) and is used by `eval`, `load`, `load-library`, and `load-program` to evaluate input forms.

This is an *incompatible change*; in previous releases, only unannotated input was passed to the values of `current-expand` and `current-eval` when they did not hold their default values. For example:

```
(load "myfile.ss" pretty-print)
```

prints annotations rather than unannotated forms. To print unannotated forms, the annotations must be stripped:

```
(load "myfile.ss"
  (lambda (x)
    (pretty-print
     (if (annotation? x)
         (annotation-stripped x)
         x))))
```

In this case, the “evaluator” passed to `load` can actually assume that its input is an annotation. This is not true of evaluators in general, which might be called on s-expressions constructed at run time.

Finally, the procedure `datum->syntax` now accepts either an annotated or unannotated input datum.

## 2.4. More foreign datatype (ftype) support (8.4)

Support for function ftypes has been added to the ftype mechanism. A function ftype takes one of the following two forms:

```
(function (arg-type ...) result-type)
(function conv (arg-type ...) result-type)
```

The *conv*, *arg-type*, and *result-type* specifiers are identical to those accepted by `foreign-procedure`.

Function ftypes are valid only at the top level of an ftype, e.g.:

```
(define-ftype bvcopy_t (function (u8* u8* size_t) void))
```

or as the immediate sub-type of a pointer ftype, as in the following definitions, which are equivalent assuming the definition of `bvcopy_t` above.

```
(define-ftype A
  (struct
   [x int]
   [f (* (function (u8* u8* size_t) void))]))

(define-ftype A
  (struct
   [x int]
   [f (* bvcopy_t)]))
```

That is, a function cannot be embedded within a struct, union, or array, but a pointer to a function can be so embedded.

A function ftype can be used with `make-ftype-pointer` to create an ftype-pointer to a C function, either by providing the name of the C function or its address, possibly obtained via the `foreign-entry` procedure, as illustrated by the following equivalent definitions:

```
(define bvcopy-fptr (make-ftype-pointer bvcopy_t "memcpy"))
(define bvcopy-fptr (make-ftype-pointer bvcopy_t (foreign-entry "memcpy")))
```

A library that defines *memcpy* must be loaded first via `load-shared-object`, or *memcpy* must be registered from C via one of the methods described in Section 4.8 of the User's Guide.

An `ftype` pointer can also be obtained as a return value from a C function declared to return a function type. However it is obtained, an function `ftype` pointer can be converted into a Scheme-callable procedure via `fctype-ref`:

```
(define bvcopy (fctype-ref bvcopy_t () bvcopy-fptr))
(define bv1 (make-bytevector 8 0))
(define bv2 (make-bytevector 8 57))
bv1 => #vu8(0 0 0 0 0 0 0 0)
bv2 => #vu8(57 57 57 57 57 57 57 57)
(bvcopy bv1 bv2 5)
bv1 => #vu8(57 57 57 57 57 0 0 0)
```

Thus, function `fctype`s can be used as alternative to `foreign-procedure` for creating Scheme-callable wrappers for C functions. Similarly, `fctype`s can be used as an alternative to `foreign-callable` for creating a C-callable wrapper for a Scheme procedure, simply by passing `make-fctype-pointer` a Scheme procedure:

```
(define fact
  (lambda (n)
    (if (= n 0) 1 (fact (- n 1)))))
(define-fctype fact_t (function (int) int))
(define fact-fptr (make-fctype-pointer fact_t fact))
```

The resulting function `fctype` pointer can be passed to a C routine, which can invoke it as it would any other function pointer. When a C-callable wrapper is created for a Scheme procedure in this manner, the address embedded within the `fctype` pointer is a pointer into a Scheme code object. Since all Scheme objects, including code objects, can be relocated or even reclaimed by the garbage collector the code object is automatically locked, as if via `lock-object`, before it is embedded in the `fctype` pointer. The code object should be unlocked after its last use from C, since locked objects take up space, cause fragmentation, and increase the cost of collection. Since the system cannot determine automatically when the last use from C occurs, the program must explicitly unlock the code object, which it can do by extracting the address from the `fctype-pointer` converting the address (back) into a code object, and passing it to `unlock-object`:

```
(unlock-object
  (foreign-callable-code-object
   (fctype-pointer-address fact-fptr)))
```

Once unlocked, the `fctype` pointer should not be used again, unless it is relocked, e.g., via:

```
(lock-object
  (foreign-callable-code-object
   (fctype-pointer-address fact-fptr)))
```

A program can determine whether an object is already locked via the new `locked-object?` predicate.

In addition to function `fctype`s, the `fctype` syntax supports three new base types: `wchar_t`, which is equivalent to the existing `wchar`, `size_t`, and `ptrdiff_t`. This set corresponds to the set required by ANSI C to be defined in the `stddef.h` header file, and thus completes the set of base types built into or required to be available by ANSI C.

One other change has been made to the `fctype` mechanism: `fctype-pointer->sexpr` has been modified to produce more readable output. In particular, null pointers are detected and represented by the symbol `null`, and Scheme strings are used to represent the contents of character arrays and character pointers, i.e., those declared to be of type `char` or `wchar`. The length of the string produced for a character array is the lesser of the length of the array and the location of the first null character, which, if present in the character array, is not included in the string. The length of the string produced for a character pointer is determined solely

by the position of the first null character found. This can lead to problems if the character array or pointer is not null-terminated; in such cases, the data should be extracted via `ftype-ref` instead.

## 2.5. Source information for primitive argument-count errors (8.4)

In previous releases, source information for argument-count errors was produced by the compiler for a subset of primitives; this has now been extended to all primitive calls. The same information is now also produced for interpreted code, i.e., the evaluator used by Petite Chez Scheme.

## 2.6. Improved recovery from invalid-memory references (8.4)

In order to handle invalid memory references caused by improper use of the `ftype` mechanism more reliably, two changes have been made. First, the Windows versions of *Chez Scheme* now detect and attempt to recover from invalid memory references, while in previous releases an invalid memory reference caused termination. Second, the recovery process has been made more robust, particularly with respect to resulting invalid pointers that might have caused problems during collection. Invalid memory references should still be avoided, whenever possible, since complete recovery cannot always be guaranteed.

## 2.7. Additional foreign datatype (`ftype`) support (8.3)

The `ftype-&ref`, `ftype-ref`, and `ftype-set!` operators added in Version 8.2 have been extended to take an optional index. For example, while `(ftype-ref double () fptr)` references the double pointed to by the `ftype` pointer `fptr`, `(ftype-ref double () fptr 1)` references the double following the double pointed to by `fptr`, and `(ftype-ref double () fptr -1)` references the double preceding the double pointed to by `fptr`.

Also, for pointer `ftypes` created with the `*` syntax, an index is now allowed in place of a `*` in the list of accessors, where it serves the same purpose as the optional index described above.

## 2.8. Eq hashtable references no longer destructive (8.3)

In previous versions, apparently nondestructive operations on hashtables, including `hashtable-ref`, are potentially destructive if passed an `eq` or `eqv` hashtable, since they may need to rehash part of the table after a garbage collection occurs. This rehashing is now performed directly by the collector so that these operations are no longer destructive. The impact of this change is that concurrent hashtable references are now thread-safe. Assignments of new values to existing keys, e.g., via `hashtable-set!`, are also thread-safe. Synchronization is still required for all hashtable operations whenever hashtable entries might be concurrently added or removed, e.g., by `hashtable-set!` or `hashtable-delete!`.

## 2.9. New `hashtable-values` procedure (8.3)

The new procedure `hashtable-values` can be used to obtain a vector containing the values stored in a hashtable. Previously, the values could be obtained only with the keys via `hashtable-entries`, which requires two vectors, one for keys and one for values, to be created, rather than just one.

## 2.10. Fasling of eq hashtables (8.3)

Eq hashtables can now be fasled out as part of a compiled file or directly via `fasl-write`.

## 2.11. Inspector and profiler source information for macro calls (8.3)

The expander now transfers source information from a macro call to the output of the macro call, if the output does not already have source information. This makes more source information available to the inspector and profiler (via `profile-dump-html`). The compiler also does a better job of propagating source information through various optimizations. On the other hand, `datum->syntax` no longer transfers source information from the template identifier to the datum, since doing so typically resulted in misleading source information.

## 2.12. Line numbers in `profile-dump-html` output (8.3)

By default, the html rendering of a source file by `profile-dump-html` now includes line numbers. The new parameter `profile-line-number-color` can be used to determine their color (which defaults to a slightly dark shade of gray). When this parameter is set to false, no line numbers are produced.

## 2.13. Change in `implicit-exports` semantics (8.3)

If `(implicit-exports #t)` appears among the definitions of a module, `(implicit-exports #t)` is no longer implied for an enclosing module or library. This allows the programmer to exercise finer control over implicit exports with no loss of expressivity, since `(implicit-exports #t)` can always be added to the enclosing module or library if desired.

## 2.14. New `compile-file-message` parameter (8.3)

The “compiling *input-file* with output to *output-file*” message printed by `compile-file`, `compile-library`, `compile-program`, and `compile-script` can be suppressed by setting the parameter `compile-file-message` to `#f`. The parameter defaults to `#t`.

## 2.15. New `compile-library-handler` parameter (8.3)

A new parameter `compile-library-handler` has been added to allow programmers to control how libraries are compiled by the expander when `compile-imported-libraries` is `#t` and the expander determines that an imported library needs to be compiled. The parameter must be set to a procedure, and the procedure is called by the expander with two string arguments identifying the input and output paths. The procedure should invoke `compile-library` and pass it the two arguments. The default value of the parameter does exactly that. The procedure can perform other operations as well, such as parameterizing compilation parameters, establishing guards, or gathering statistics.

## 2.16. `library-directories` and `library-extensions` now thread parameters (8.3)

In previous versions, these were global parameters for no good reason.

## 2.17. Parameterization of `cp0` compilation parameters (8.3)

`run-cp0`, `cp0-effort-limit`, `cp0-score-limit`, and `cp0-outer-unroll-limit` are now parameterized by `compile-file`, `compile-library`, `compile-program`, and `compile-script`, like most other compilation parameters. Thus, if one is set within a source file at compile time (via an `eval-when` form), the setting will affect the compilation of that source file only.

## 2.18. Foreign datatype (ftype) support (8.2)

A new, high-level, and efficient syntactic interface for manipulating foreign data has been added. Through the `define-ftype` form, the interface supports the declaration of foreign types (ftypes), including structures, unions, arrays, and bit fields, and it allows control over the endianness and packing of the fields of the type. It also supports creation (`make-foreign-pointer`), access (`foreign-&ref` and `foreign-ref`), and assignment forms (`foreign-set!`) with automatic type checking to ensure that all accesses are consistent with the declared type of a foreign pointer. As with most other type checks, the checks are disabled at `optimize-level 3`.

Ftypes can also be used as `foreign-procedure` and `foreign-callable` argument and return types, so there is often no need to deal directly with the addresses of foreign data. Structs, unions, and arrays can be passed only by reference and must be explicitly wrapped in the syntax of the `foreign-procedure` or `foreign-callable` form with a pointer ftype specifier, i.e., `(* ftype)`. Base types, including user-defined aliases for base types, can be passed by value and should not be so wrapped except when the argument or return value should be a pointer to a base type rather than the base type itself.

Foreign objects can be inspected, like other objects, via the inspector. It is also possible to convert a foreign object (including one with cycles) into an s-expression suitable for pretty-printing, making it relatively easy to view the contents of a foreign object.

## 2.19. Locks (8.2)

A new low-level lock mechanism for synchronization of parallel programs has been added. Locks are intended to be allocated outside of the Scheme heap and, if allocated in memory shared by multiple processors, can be used for synchronization among separate O/S processes. Locks can also be used in place of mutexes for synchronization among the threads of a single process in threaded versions of *Chez Scheme*. Locks lack some of the functionality of mutexes but have lower overhead than mutexes.

## 2.20. New export forms (8.1)

Three new export forms have been added:

```
(export <export-spec> ...)  
(implicit-exports <boolean>)  
(indirect-export id indirect-id ...)
```

Each of these forms is a definition. The first two can appear only within the definitions of a module or library; the third can appear anywhere other definitions can appear.

The `export` form causes the identifiers specified by each *export-spec* to be exports of the enclosing module or library. An *export-spec* is one of:

```
export-spec  →  identifier  
                |  (rename (old-identifier new-identifier) ...)  
                |  (import import-spec ...)
```

The first two are syntactically identical to the R6RS library export form *export-specs*, while the third is syntactically identical to a *Chez Scheme* `import` form, which is an extension of the R6RS library `import` subform. If the *export-spec* is an identifier, that identifier becomes an export of the enclosing library or module. If it is a `rename` form, the bindings of the old identifiers become exports under the new identifier names. If it is an `import` form, the imported identifiers become exports, with aliasing, renaming, prefixing, etc., as specified by the *import-specs*.

The module or library whose bindings are exported by an `import` form appearing within an `export` form can be defined within or outside the exporting module or library and need not be imported elsewhere within the exporting module or library.

The `implicit-exports` form determines whether identifiers that are not directly exported from a module or library are automatically indirectly exported to the top level if any meta-binding (keyword, meta definition, or property definition) is directly exported from a library or module to top level. The default for libraries is `#t`, to match the behavior required by the R6RS, while the default for modules is `#f`, to match the behavior of previous versions of *Chez Scheme*. The `implicit-exports` form is meaningful only within a library, top-level module, or module enclosed within a library or top-level module. It is allowed but ignored in a module enclosed within a `lambda`, `let`, or similar body, because none of that module's bindings can be exported to top level.

The advantage of (`implicit-exports #t`) is that indirect exports need not be listed explicitly, which is convenient. One disadvantage is that it might result in more bindings than necessary being elevated to top level where they cannot be discarded as useless by the optimizer. For modules, another, often significant, disadvantage is such bindings cannot be proven immutable, which inhibits important optimizations such as procedure inlining.

Finally, the `indirect-export` form declares that the named *indirect-ids* are indirectly exported to top level if *id* is exported to top level. It is meaningful only within a top-level library, top-level module, or module enclosed within a library or top-level module. It is allowed anywhere else definitions can appear, however, so macros that expand into indirect export forms can be used in any definition context.

## 2.21. Change in `import import-only` (8.1)

In previous versions, an `import` or `import-only` form with multiple subforms is treated the same as a sequence of `import` or `import-only` forms. For `import`, the consequences of this are that a module name visible in the scope of the `import` form and reference by one of its subforms can be shadowed by the imports of an earlier subform. For `import-only` the consequences are (1) a module name referenced in the second or a subsequent subform must be imported via the preceding subform, and (b) only the bindings imported by the last subform are visible in the rest of the body in which the `import-only` form appears.

In Version 8.4, all of the module names referenced by any of the subforms are scoped where the `import` or `import-only` form appears. Furthermore, for `import-only`, all (and only) the imports specified by the entire set of subforms are visible where the `import-only` form appears. This simplifies the task of creating a limited local scope from multiple libraries and modules.

If the old behavior is desired, a macro that expands into a sequence of import forms can be used, e.g.:

```
(define-syntax import*
  (syntax-rules ()
    [(_ subform ...) (begin (import subform) ...)]))
```

## 2.22. Improved support for `define-property` (8.1)

When properties attached to the same identifier with different keys are imported from different modules or libraries, all of the properties are now visible. For example, if library (L1) attaches a `car` property to `cons` and reexports `cons`, while library (L2) attaches a `cdr` property to `cons` and reexports `cons`, and both (L1) and (L2) are imported, the `car` and `cdr` properties are both available to a macro within the scope of the imports.

Also, if indirect exports are declared for an identifier imported into and reexported from a library or top-level module, and the identifier is given a property via `define-property` within that library or top-level module, the indirect exports are now indirectly exported to top level when the identifier is exported to top level, even though indirect exports are normally ignored for reexported bindings.

Finally, an identifier that is made the alias of another identifier via `import` or `alias` forms now has the same properties as the aliased identifier at the point where the alias occurs, although properties subsequently defined for either identifier do not affect the other.

### 2.23. Improved library-group support (8.1)

The `library-group` now handles “synthetic cycles” involving static dependencies. For example, if library (C) depends on library (B), and library (B) depends on library (A), putting libraries (A) and (C) but not (B) into a library group creates a “synthetic cycle” involving the library group and the separate library (B). Previously, such cycles resulted in cyclic dependency errors.

### 2.24. Improved top-level `begin` library support (8.1)

When multiple libraries and top-level programs appear in a top-level `begin` form, the expander now delays the invocation of each library until just before it is actually needed. Previously, all libraries not defined in the same top-level `begin` form were invoked before any of the forms in the `begin` were evaluated, which sometimes led to undefined library or cyclic dependency errors.

### 2.25. Fewer import dependencies (8.1)

Libraries imported only locally in code compiled to a file no longer show up as import dependencies for the compiled code, reducing load-time overhead for the compiled code and eliminating the need to distribute some libraries with the compiled code. If a library’s exported identifiers need to be visible when the compiled code is subsequently loaded, the library should be imported explicitly at top level. This change affects only code defined outside of a `library` or RNRS top-level program.

### 2.26. No longer using the SIGCHLD signal (8.1)

In previous versions of *Chez Scheme*, processes created under Unix-based operating systems by the `process` and `open-process-ports` procedures were reaped via a SIGCHLD interrupt handler. In Version 8.4, such processes are instead reaped by the garbage collector, and the system no longer sets up a SIGCHLD handler. This allows programs to set up their own handlers without interfering with the reaping of those created by `process` and `open-process-ports`. It also eliminates a potential race condition involving the `system` procedure or foreign procedures that create and wait for subprocesses after first disabling the SIGCHLD interrupt.

### 2.27. `top-level-syntax` generalization (8.1)

The `top-level-syntax` procedure now returns a binding for any bound identifier, even if it is defined as a variable, and `top-level-syntax?` returns true for all bound identifiers. Thus, it is now possible to transfer the compile-time binding of any identifier to another, regardless of the type of binding.

### 2.28. NetBSD x86/x86\_64 support (8.1)

Support for running *Chez Scheme* with 32-bit pointers on the i386 architecture or 64-bit pointers on the x86\_64 architecture under NetBSD has been added, with machine types `i3nb` (32-bit), `a6nb` (64-bit), `ti3nb` (32-bit threaded), and `ta6nb` (64-bit threaded). C code intended to be linked with these versions of the system should be compiled using the Gnu C compiler’s `-m32` or `-m64` options as appropriate.

### 2.29. Now using `msvcr100.dll` (8.1)

Windows builds of *Chez Scheme* now link against `msvcr100.dll`. Static libraries built using the corresponding `libcmnt` are also available. With the current Microsoft C compiler tools, manifests are no longer required or recommended and so are no longer recorded with the DLLs and executables.

## 3. Bug Fixes

### 3.1. `assert` return value (8.4)

The fix below to `assert` was made incorrectly and caused `assert` not to return the value of its subexpression when that value is true. This new bug has been fixed. [This bug dated back to Version 8.3.]

### 3.2. `let-values` and `assert` `fasl` errors (8.3)

A bug that resulted in a `fasl` error while compiling files containing certain `let-values`, `let*-values`, `define-values`, and `assert` forms has been fixed. To trigger the bug, the source code for one of these forms had to include a non-`faslable` value, such as a procedure or generic hashtable; or one of the identifiers occurring in the form had to have an `expand-time` property whose value was non-`faslable`. The problem resulted from the inclusion of the source syntax object in the generated code for use in producing more useful error messages; the solution is to create the useful error message ahead of time and include just the message in the generated code. [This bug dated back to Version 7.5 for `assert` and 7.9.2 for `let-values`.]

### 3.3. Bignum hash values (8.3)

A bug in the handling of bignum hash values by the generic hashtable routines has been fixed. [This bug dated back to Version 7.5.]

### 3.4. Keyboard interrupt segmentation violation (8.3)

A bug that caused segmentation violations after a keyboard interrupts on 64-bit machines has been fixed. Also, when such interrupts occur while reading from the console, they are no longer noncontinuable interrupts. [This bug dated back to Version 7.9.2.]

### 3.5. Expression-editor paste problems (8.2)

A bug that resulted in an invalid memory reference if `^V` (control-V) was entered into the expression editor when no text is available to paste on 64-bit versions of Chez Scheme has been fixed. The code that locates and inserts pasted text under X Windows has also been made more robust to reduce the likelihood that an ill-behaved application will deny access to its selected text. [This bug dated back to Version 7.9.1.]

### 3.6. `define-record-type` and undefined `rcd`/`rtd` set (8.1)

A bug that sometimes resulted in an unexported-identifier exception for `rtd` or `rcd` when a record type defined via `define-record-type` and exported by a top-level module or module defined at the top level of a library was used as the parent for another record type defined outside of the module or library has been fixed. [This bug dated back to Version 7.5.]

### 3.7. Incorrect error messages with `source-directories` set (8.1)

A bug that caused the incorrect (and useless) message “*filename* not found in source directories” when a syntax error occurred in a file loaded via `load`, `load-library`, or `load-program` while `source-directories` was set to something other than `(".")` or `("")` has been fixed. [This bug dated back to Version 7.5.]

### 3.8. Bug in record-constructor (8.1)

A bug that caused certain record constructors for records with parents who themselves have parents and user-defined protocols to produce an “incorrect number of arguments” error has been fixed. [This bug dated back to Version 7.5.]

### 3.9. Bug in string-titlecase (8.1)

A bug that resulted in `string-titlecase` causing an invalid memory reference when passed a string containing a sequence of two or more digits has been fixed. [This bug dated back to Version 7.5.]

### 3.10. Bug in exp (8.1)

A bug in `exp` that caused it to return incorrect results for large inputs, including `+inf.0`, has been fixed. The same bug also caused some problems with `expt` and possibly with certain other mathematical operations that use `exp` internally. [This bug dated back to Version 4.0.]

### 3.11. Missing check in (rnrs) case (8.1)

Non-pair unparenthesized keys are now properly rejected by the version of `case` exported by the `(rnrs base)` and `(rnrs)` libraries. [This bug dated back to Version 7.5.]

### 3.12. Missing check in fxvector-set! (8.1)

`fxvector-set!` now properly checks to make sure its third (new value) argument is a fixnum. [This bug dated back to Version 7.3.]

### 3.13. Engine space leak (8.1)

A bug that caused the engine system to retain inaccessible continuations has been fixed.

## 4. Performance Enhancements

### 4.1. Compile-time record instance creation (8.4)

The source optimizer already produced inline machine code for record constructors, predicates, accessors, etc. It now goes a step further and folds (performs at compile time) construction of nongenerative rtds, and immutable record instances. It also now folds record predicate calls and field references on constant immutable records.

### 4.2. Native threads under Windows (8.4)

Thread support under Windows is no longer based on the `pthread-win32` library but instead uses Windows API threading directly, cutting down on some overhead.

### 4.3. Faster macro expansion (8.3)

The expander’s handling of libraries with numerous implicit indirect exports took a performance hit in Version 8.1 when the new library export forms were added. This performance hit has been eliminated.

Furthermore, the macro expander now handles any program with large numbers of identifier bindings more efficiently even than Version 8.0. The improvement in overall compile time is typically around 10-15% versus Version 8.0 and can be an order of magnitude or more versus Version 8.1.

#### **4.4. Faster dumping of profile information (8.3)**

The procedure `profile-dump-list` is now considerably faster when processing large amounts of profiled code. The procedure `profile-dump-html` is somewhat faster as well, although its cost is largely determined by the time required to read source files and write html output.

#### **4.5. Better arithmetic support (8.1)**

The `fx*` procedure is now inlined even outside of optimize-level 3 on x86 and x86\_64 processors, and a slight improvement has been made in the efficiency of generic arithmetic operators when passed non-fixnum arguments.

#### **4.6. Slightly faster type checks (8.1)**

Reduced by one instruction the cost of `vector?` and certain other type checks. This is unlikely to have measurable impact on most programs.