Chez Scheme Version 9.4 Release Notes
May 2016

1. Overview

This document outlines the changes made to *Chez Scheme* for Version 9.4 since Version 8.4.

Version 9.4 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)
- Linux ARMv6 (32-bit), nonthreaded (arm32le)
- Linux PowerPC (32-bit), nonthreaded (ppc32le) and threaded (tppc32le)
- Windows x86, nonthreaded (i3nt) and threaded (ti3nt)
- Windows x86_64, nonthreaded (a6nt) and threaded (ta6nt) [experimental]

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.

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 9 User's Guide*. Online versions of both books can be found at http://www.scheme.com.

2. Functionality Changes

2.1. Extensions to whole-program, whole-library optimization (9.3.1, 9.3.4)

`compile-whole-program` now supports incomplete whole-program optimization, i.e., whole program optimization that incorporates only libraries for which wpo files are available while leaving separate libraries for which only object files are available. In addition, imported libraries can be left visible for run-time use by the `environment` procedure or for dynamically loaded object files that might require them. The new procedure `compile-whole-library` supports the combination of groups of libraries separate from programs and unconditionally leaves all imported libraries visible.

2.2. 24-, 40-, 48-, and 56-bit bit-field containers (9.3.3)

The total size of the fields within an ftype `bits` can now be 24, 40, 48, or 56 (as well as 8, 16, 32, and 64).

2.3. Object-counting for static-generation collections (9.3.3)

Object counting (see `object-counts` below) is now enabled for all collections targeting the static generation.

2.4. Support for off-line profile profile-dump processing (9.3.2)

Previously, the output of `profile-dump` was not specified. It is now specified to be a list of source-object, profile-count pairs. In addition, `profile-dump-html`, `profile-dump-list`, and `profile-dump-data` all now take an optional *dump* argument, which is a list of source-object, profile-count pairs in the form returned by `profile-dump` and defaults to the current value of `(profile-dump)`.

With these changes, it is now possible to obtain a dump from `profile-dump` in one process, and write it to a fasl file (using `fasl-write`) for subsequent off-line processing in another process, where it can be read from the fasl file (using `fasl-read`) and processed using `profile-dump-html`, `profile-dump-list`, `profile-dump-data` or some custom mechanism.

2.5. More support for controlling return of memory to the O/S (9.3.2)

A new parameter, `release-minimum-generation`, determines when the collector attempts to return unneeded virtual memory to the O/S. It defaults to the value of `collect-maximum-generation`, so the collector attempts to return memory to the O/S only when performing a maximum-generation collection. It can be set to a lower generation number to cause the collector to do so for younger generations we well.

2.6. `library-group` eliminated (9.3.1)

With the extensions to `compile-whole-program` and the addition of `compile-whole-library`, as described above, support for whole-program and whole-library optimization now subsumes the functionality of the experimental `library-group` form, and the form has been eliminated. This is an *incompatible change*.

2.7. Support for Version 7 interaction-environment semantics eliminated (9.3.1)

Prior to Version 8, the semantics of the interaction environment used by the read-eval-print loop (REPL), aka waiter, and by `load`, `compile`, and `interpret` without explicit environment arguments treated all variables in the environment as mutable, including those bound to primitives. This meant that top-level references to primitive names could not be optimized by the compiler because their values might change at run time, except that, at optimize-level 2 and above, the compiler did treat primitive names as always having their original values.

In Version 8 and subsequent versions, primitive bindings in the interaction environment are immutable, as if imported directly from the immutable Scheme environment. That is, they cannot be assigned, although they can be replaced with new bindings with a top-level definition.

To provide temporary backward compatibility, the `--revert-interaction-semantics` command-line option and `revert-interaction-semantics` parameter allowed programmers to revert the interaction environment to Version 7 semantics. This functionality has now been eliminated and along with it the special treatment of primitive bindings at optimize level 2 and above.

This is an *incompatible change*.

2.8. Explicit specification of profile source locations (9.3.1)

Version 9.3.1 augments existing support for explicit source-code annotations with additional features targeted at source profiling for externally generated programs, including programs generated by language front ends that target Scheme and use Chez Scheme as the back end. Included is a `profile` expression that explicitly associates a specified source object with a profile count (of times the expression is evaluated), `generate-profile-forms` parameter that controls whether the compiler (also) associates profile counts with source locations implicitly identified by annotated expressions in the input, and a finer-grained method for marking whether an individual annotation should be used for debugging, profiling, or both.

2.9. "Maybe" file (re)compilation (9.3.1)

When `compile-imported-libraries` is set to `#t`, libraries required indirectly by one of the file-compilation procedures, e.g., `compile-library`, `compile-program`, and `compile-file`, are automatically compiled if and only if the object file is not present, older than the source (main and include) files, or some library upon which they depend has been or needs to be recompiled.

Version 9.3.1 adds three new procedures: `maybe-recompile-library`, `maybe-recompile-program`, and `maybe-recompile-file`, that perform a similar analysis and compile the library, program, or file only under similar circumstances.

2.10. New primitives for querying memory utilization (9.3.1)

Three new primitives have been added to allow a Scheme process to track usage of virtual memory for its heap.

`current-memory-bytes` returns the total number of bytes of virtual memory used or reserved to represent the Scheme heap. This differs from `bytes-allocated`, which returns the number of bytes currently occupied by Scheme objects. `current-memory-bytes` additionally includes memory used for heap management as well as memory held in reserve to satisfy future allocation requests.

`maximum-memory-bytes` returns the maximum number of bytes of virtual memory occupied or reserved for the Scheme heap by the calling process since the last call to `reset-maximum-memory-bytes!` or, if `reset-maximum-memory-bytes!` has never been called, since system start-up.

`reset-maximum-memory-bytes!` resets the maximum memory bytes to the current memory bytes.

2.11. Unicode 7.0 support (9.3.1)

The character sets, character classes, and word-breaking algorithms for character, string, and Unicode-related bytevector operations have now been updated to Unicode 7.0.

2.12. Linux PowerPC (32-bit) support (9.3)

Support for running *Chez Scheme* on 32-bit PowerPC processors running Linux has been added, with machines type ppc32le (nonthreaded) and tppc32le (threaded). C code intended to be linked with these versions of the system should be compiled using the GNU C compiler's `-m32` option.

2.13. Printed representation of procedures (9.2.1)

The printed representation of a procedure now includes the source file and beginning file position when available.

2.14. I/O errors writing to the console error port (9.2.1)

The default exception handler now catches I/O exceptions that occur when it attempts to display a condition and, if an I/O exception does occur, resets as if by calling the `reset` procedure. The intent is to avoid an infinite regression (ultimately ending in exhaustion of memory) in which the process repeatedly recurs back to the default exception handler trying to write to a console-error port (typically stderr) that is no longer writable, e.g., due to the other end of a pipe or socket having been closed.

2.15. C locking macros (9.2.1)

The header file scheme.h distributed with Chez Scheme now includes several new lock-related macros: `INITLOCK` (corresponding to `ftype-init-lock!`), `SPINLOCK` (`ftype-spin-lock!`), `UNLOCK` (`ftype-unlock!`), `LOCKED_INCR` (`ftype-locked-incr!`), and `LOCKED_DECR` (`ftype-locked-decr!`). All take a pointer to an iptr or uptr. `LOCKED_INCR` and `LOCKED_DECR` also take an `lvalue` argument that is set to true (nonzero) if the result of the increment or decrement is zero, otherwise false (zero).

2.16. New `compile-to-file` procedure (9.2.1)

The new procedure `compile-to-file` is similar to `compile-to-port` with the output port replaced with an output pathname.

2.17. Whole-program optimization (9.2)

Version 9.2 includes support for whole-program optimization of a top-level program and the libraries upon which it depends at run time based on "wpo" (whole-program-optimization) files produced as a byproduct of compiling the program and libraries when the parameter `generate-wpo-files` is set to `#t`. The new procedure `compile-whole-program` takes as input a wpo file for a top-level program, combines it with the wpo files for any libraries the program requires at run time, and produces a single object file containing a self-contained program. In so doing, it discards unused code and optimizes across program and library boundaries, potentially reducing program load time, run time, and memory requirements.

`compile-file`, `compile-program`, `compile-library`, and `compile-script` produce wpo files as well as ordinary object files when the new `generate-wpo-files` parameter is set to `#t` (the default is `#f`). `compile-port` and `compile-to-port` do so when passed an optional *wpo output port*.

2.18. Type-specific symbol-hashtable operators (9.2)

A new set of primitives that operate on symbol hashtables has been added:

```
symbol-hashtable?
symbol-hashtable-ref
symbol-hashtable-set!
symbol-hashtable-contains?
symbol-hashtable-cell
symbol-hashtable-update!
symbol-hashtable-delete!
```

These are like their generic counterparts but operate only on symbol hashtables, i.e., hashtables created with `symbol-hash` as the hash function and `eq?`, `eqv?`, `equal?`, or `symbol=?` as the equivalence function.

These primitives are more efficient at optimize-level 3 than their generic counterparts when both are applied to symbol hashtables. The performance of symbol hashtables has been improved even when the new operators are not used (Section 4.9).

2.19. `strip-fasl-file` is now machine-independent (9.2)

`strip-fasl-file` can now strip fasl files created for a machine type other than the machine type of the calling process as long as the Chez Scheme version is the same.

2.20. `source-file-descriptor` and `locate-source` (9.2)

The new procedure `source-file-descriptor` can be used to construct a custom source-file descriptor or reconstruct a source-file descriptor from values previously extracted from another source-file descriptor. It takes two arguments: a string *path* and exact nonnegative integer *checksum* and returns a new source-file descriptor.

The new procedure `locate-source` can be used to determine a full path, line number, and character position from a source-file descriptor and file position. It accepts two arguments: a source-file descriptor *sfd* and an exact nonnegative integer file position *fp*. It returns zero values if the unmodified file is not found in the source directories and three values (string *path*, exact nonnegative integer *line*, and exact nonnegative integer *char*) if the file is found.

2.21. Compressed compiled scripts and partially compressed files (9.2)

Support for creating and handling files that begin with uncompressed data and end with compressed data has been added in the form of the new procedure `port-file-compressed!` that takes a port and if not already set up to read or write compressed data, sets it up to do so. The port must be a file port pointing to a regular file, i.e., a file on disk rather than a socket or pipe, and the port must not be an input/output port. The port can be a binary or textual port. If the port is an output port, subsequent output sent to the port will be compressed. If the port is an input port, subsequent input will be decompressed if and only if the port is currently pointing at compressed data.

When the parameter `compile-compressed` is set ot `#t`, the `compile-script` and `compile-program` procedures take advantage of this functionality to copy the `#!` prefix, if present in the source file, uncompressed in the object file while compressing the object code emitted for the program, thus reducing the size of the resulting file without preventing the `#!` line from being read and interpreted properly by the operating system.

2.22. Change in library import handling (9.2)

In previous releases, when an object file was found before the corresponding source file in the library directories, the object file was older, and the parameter `compile-imported-libraries` was not set, the object file was loaded rather than the source file. The (newer) source file is now loaded instead, just as it would be if the source file is found before the corresponding, older object file. This is an *incompatible change*.

2.23. Change in fasl-strip options (9.1)

`strip-fasl-file` now supports stripping of all compile-time information and no longer supports stripping of just library visit code. Stripping all compile-time information nearly always results in smaller object files than stripping just library visit code, with a corresponding reduction in the memory required when the resulting file is loaded.

To reflect this, the old fasl-strip option `library-visit-code` has been eliminated, and the new fasl-strip option `compile-time-information` has been added. This is an *incompatible change* in that code that previously used the fasl-strip option `library-visit-code` will have to be modified to omit the option or to replace it with `compile-time-information`.

2.24. Library loading (9.1)

Visiting (via `visit`) a library no longer loads the library's run-time information (invoke dependencies and invoke code), and revisiting (via `revisit`) a library no longer loads the library's compile-time information (import and visit dependencies and import and visit code).

When a library is invoked due to a run-time dependency of another library or a top-level program on the library, the library is now "revisited" (as if via `revisit`) rather than "loaded" (as if via `load`). As a result, the compile-time information is not loaded, which can result in substantial reductions in both library invocation time and memory footprint.

If a library is revisited, either explicitly or as the result of run-time dependency, a subsequent import of the library causes it to be "visited" (as if via `visit`) if the same object file can be found at the same path and the visit code has not been stripped. The compile-time code can alternatively be loaded explicitly from the same or a different file via a direct call to `visit`.

While this change is mostly transparent (ignoring the reduced invocation time and memory footprint), it is an *incompatible change* in the sense that the system potentially reads the file twice and can run code that is marked using `eval-when` as both visit and revisit code.

2.25. Finding objects in the heap (9.1)

Version 9.1 includes support for a new heap inspection tool that allows a programmer to look for objects in the heap according to arbitrary predicates. The new procedure `make-object-finder` takes a predicate *pred* and two optional arguments: a starting point x and a maximum generation g. The starting point defaults to the value of the procedure `oblist`, and the maximum generation defaults to the value of the parameter `collect-maximum-generation`. `make-object-finder` returns an object finder p that can be used to search for objects satisfying *pred* within the starting-point object x. Immediate objects and objects in generations older than g are treated as leaves. p is a procedure accepting no arguments. If an object y satisfying *pred* can be found starting with x, p returns a list whose first element is y and whose remaining elements represent the path of objects from x to y, listed in reverse order. p can be invoked multiple times to find additional objects satisfying the predicate, if any. p returns `#f` if no more objects matching the predicate can be found.

p maintains internal state recording where it has been so that it can restart at the point of the last found object and not return the same object twice. The state can be several times the size of the starting-point object x and all that is reachable from x.

The interactive inspector provides a convenient interface to the object finder in the form of `find` and `find-next` commands. The `find` command evaluates its first argument, which should evaluate to the desired predicate, and treats its second argument, if present, as the maximum generation, overriding the default. The starting point x is the object upon which the inspector is currently focused. If an object is found, the inspector's new focus is the found object, the parent focus (obtainable via the `up` command) is the first element in the (reversed) path, the parent's parent is the next element, and so on up to x. The `find-next` command repeats the last find, as if by an explicit invocation of the same object finder.

Relocation tables for static code objects are discarded by default, which prevents object finders from providing accurate results when static code objects are involved. That is, they will not find any objects pointed to directly from a code object that has been promoted to the static generation. If this is a problem, the command-line argument `--retain-static-relocation` can be used to prevent the relocation tables from being discarded.

2.26. Object counts (9.1)

The new procedure `object-counts` can be used to determine, for each type of object, the number and size in bytes of objects of that type in each generation. Its return value has the following structure:

`((`*type* `(`*generation count* `.` *bytes*`) ...) ...)`

type is either the name of a primitive type, represented as a symbol, e.g., `pair`, or a record-type descriptor (rtd). *generation* is a nonnegative fixnum between 0 and the value of `(collect-maximum-generation)`, inclusive, or the symbol `static` representing the static generation. *count* and *bytes* are nonnegative fixnums.

Object counts are accurate for a generation n immediately after a collection of generation n or higher if

enabled during that collection. Object counts are enabled by setting the parameter `enable-object-counts` to `#t`. The command-line option `--enable-object-counts` can be used to set this parameter to `#t` on startup. Object counts are not enabled by default since it adds overhead to garbage collection.

To make the information more useful in the presence of ftype pointers, the ftype descriptors produced by `define-ftype` for each defined ftype now carry the name of the ftype rather than a generic name like `ftd-struct`. (Ftype descriptors are subtypes of record-type descriptors and can appear as types in the `object-counts` return value.)

2.27. Native-eol style is now none (9.1)

To simplify interaction with tools that naively expose multiple-character end-of-line sequences such as CRLF as separate characters to the user, the native end-of-line style (`native-eol-style`) is now `none` on all machine types. This is an *incompatible change*.

2.28. Library-requirements options (9.1)

In previous releases, the `library-requirements` procedure returns a list of all libraries required by the specified library, whether they are needed when the specified library is imported, visited, or invoked. While this remains the default behavior, `library-requirements` now takes an optional "options" argument. This must be a library-requirements-options enumerations set, i.e., the value of a `library-requirements-options` form with some subset of the options `import`, `visit@visit`, `invoke@visit`, and `invoke`. `import` includes the libraries that must be imported when the specified library is imported; `visit@visit` includes the libraries that must be visited when the specified library is visited; `invoke@visit` includes the libraries that must be invoked when the specified library is visited; and `invoke` includes the libraries that must be invoked when the specified library is invoked. The default behavior is obtained by supplying a enumeration set containing all of these options.

2.29. Nested object size and composition (9.1)

Two new procedures, `compute-size` and `compute-composition`, can be used to determine the size and make-up of nested objects with the heap.

Both take an object and an optional generation. The generation must be a fixnum between 0 and the value of `(collect-maximum-generation)`, inclusive, or the symbol static. It defaults to the value of `(collect-maximum-generation)`.

`compute-size` returns the number of bytes occupied by the object and everything to which it points, ignoring objects in generations older than the specified generation.

`compute-composition` returns an association list giving the number and number of bytes of each type of object that the specified object is constructed from, ignoring objects in generations older than the specified generation. The association list maps type names (e.g., pair and flonum) or record-type descriptors to a pair of fixnums giving the count and bytes. Types with zero counts are not included in the list.

A surprising number of objects effectively point indirectly to a large percentage of all objects in the heap due to the attachment of top-level environment bindings to symbols, but the generation argument can be used in combination with explicit calls to collect (with automatic collections disabled) to measure precisely how much space is allocated to freshly allocated structures.

When used directly from the REPL with no other threads running, `(compute-size (oblist) 'static)` effectively gives the size of the entire heap, and `(compute-composition (oblist) 'static)` effectively gives the composition of the entire heap.

The inspector makes the aggregate size of an object similarly available through the `size` inspector-object message and the corresponding `size` interactive-inspector command, with the twist that it does not include

objects whose sizes were previously requested in the same session, making it possible to see the effectively smaller sizes of what the programmer perceives to be substructures in shared and cyclic structures.

These procedures potentially allocate a large amount of memory and so should be used only when the information returned by the procedure `object-counts` (see preceding entry) does not suffice.

Relocation tables for static code objects are discarded by default, which prevents these procedures from providing accurate results when static code objects are involved. That is, they will not find any objects pointed to directly from a code object that has been promoted to the static generation. If accurate sizes and compositions for static code objects are required, the command-line argument `--retain-static-relocation` can be used to prevent the relocation tables from being discarded.

2.30. Showing expander and optimizer output (9.1)

When the parameter `expand-output` is set to a textual output port, the output of the expander is printed to the port as a side effect of running `compile`, `interpret`, or any of the file compiling primitives, e.g., `compile-file` or `compile-library`. Similarly, when the parameter `expand/optimize-output` is set to a textual output port, the output of the source optimizer is printed.

2.31. Undefined-variable warnings (9.1)

When `undefined-variable-warnings` is set to `#t`, the compiler issues a warning message whenever it cannot determine that a variable bound by `letrec`, `letrec*`, or an internal definition will not be referenced before it is defined. The default value is `#f`.

Regardless of the setting of this parameter, the compiler inserts code to check for the error, except at optimize level 3. The check is fairly inexpensive and does not typically inhibit inlining or other optimizations. In code that must be carefully tuned, however, it is sometimes useful to reorder bindings or make other changes to eliminate the checks. Enabling this warning can facilitate this process.

The checks are also visible in the output of `expand/optimize`.

2.32. Detecting accidental use of generative record types (9.1)

When the new boolean parameter `require-nongenerative-clause` is set to `#t`, a `define-record-type` without a `nongenerative` clause is treated as a syntax error. This allows the programmer to detect accidental use of generative record types. Generative record types are rarely useful and are less efficient than nongenerative types, since generative record types require the construction of a record-type-descriptor each time a `define-record-type` form is evaluated rather than once, at compile time. To support the rare need for a generative record type while still allowing accidental generativity to be detected, `define-record-type` has been extended to allow a generative record type to be explicitly declared with a `nongenerative` clause with `#f` for the uid, i.e., `(nongenerative #f)`.

2.33. Improved support for cross compilation (9.1)

Cross-compilation support has been improved in two ways: (1) it is now possible to cross-compile a library and import it later in a separate process for cross-compilation of dependent libraries, and (2) the code produced for the target machine when cross compiling is no longer less efficient than code produced natively on the target machine.

2.34. Linux ARMv6 (32-bit) support (9.1)

Support for running *Chez Scheme* on ARMv6 processors running Linux has been added, with machine type arm32le (32-bit nonthreaded). C code intended to be linked with these versions of the system should be

compiled using the GNU C compiler's `-m32` option.

2.35. Source information in ftype ref/set! error messages (9.0)

When available at compile time, source information is now included in run-time error messages produced when `ftype-&ref`, `ftype-ref`, `ftype-set!`, and the locked ftype operations are handed invalid inputs, e.g., ftype pointers of some unexpected type, RHS values of some unexpected type, or improper indices.

2.36. `compile-to-port` top-level-program dependencies (9.0)

When passed a single `top-level-program` form, `compile-to-port` now returns a list of the libraries the top-level program requires at run time, as with `compile-program`. Otherwise, the return value is unspecified.

2.37. Better feedback for record-type mismatches (9.0)

When `make-record-type` or `make-record-type-descriptor` detect an incompatibility between two record types with the same UID, the resulting error messages provide more information to describe the mismatch, i.e., whether the parent, fields, flags, or mutability differ.

2.38. `enable-cross-library-optimization` parameter (9.0)

When a library is compiled, information is stored with the object code to enable propagation of constants and inlining of procedures defined in the library into dependent libraries. The new parameter `enable-cross-library-optimization`, whose value defaults to `#t`, can be set to `#f` to prevent this information from being stored and disable the corresponding optimizations. This might be done to reduce the size of the object files or to reduce the potential for exposure of near-source information via the object file.

2.39. Stripping object files (9.0)

The new procedure `strip-fasl-file` allows the removal of source information of various sorts from a compiled object (fasl) file produced by `compile-file` or one of the other file compiling procedures. It also allows removal of library visit code, i.e., the code required to compile (but not run) dependent libraries.

`strip-fasl-file` accepts three arguments: an input pathname, and output pathname, and a fasl-strip-options enumeration set, created by `fasl-strip-options` with zero or more of the following options.

`inspector-source`: Strip inspector source information.

`source-annotations`: Strip source annotations.

`profile-source`: Strip source file and character position information from profiled code objects.

`library-visit-code`: This strips library visit code from compiled libraries.

2.40. Ftype array bound of zero (9.0)

The bound of an ftype array can now be zero and, when zero, is treated as unbounded in the sense that no run-time upper-bound checks are performed for accesses to the array. This simplifies the creation of ftype arrays whose actual bounds are determined dynamically.

2.41. `compile-profile` no longer implies `generate-inspector-information` (9.0)

In previous releases, profile and inspector source information was gathered and stored together so that compiling with profiling enabled required that inspector information also be stored with each code object. This is no longer the case.

2.42. `case` now uses `member` (9.0)

`case` now uses `member` rather than `memv` for key comparisons, a generalization that allows `case` to be used for strings, lists, vectors, etc., rather than just atomic values. This adds no overhead when keys are comparable with `memv`, since the compiler converts calls to `member` into calls to `memv` (or `memq`, or even individual inline pointer comparisons) when it can determine the more expensive test is not required.

The `case` syntax exported by the `(rnrs)` and `(rnrs base)` libraries still uses `memv` for compatibility with the R6RS standard.

2.43. `write` and `display` and foreign addresses (9.0)

The `write` and `display` procedures now recognize foreign addresses that happen to look like Scheme objects and print them as `#<foreign>`; previously, `write` and `display` would attempt to treat the addresses as Scheme objects, typically leading to invalid memory references. Some foreign addresses are indistinguishable from fixnums and still print as fixnums.

2.44. Profile-directed optimization (9.0)

Compiled code can be instrumented to gather two kinds of execution counts, source-level and block-level, via different settings of the `compile-profile` parameter. When `compile-profile` is set to the symbol `source` at compile time, source execution counts are gathered by the generated code, and when `compile-profile` is set to `block`, block execution counts are gathered. Setting it to `#f` (the default) disables instrumentation.

Source counts are identical to the source counts gathered by generated code in previous releases when compiled with `compile-profile` set to `#t`, and `#t` can be still be used in place of `source` for backward compatibility. Source counts can be viewed by the programmer at the end of the run of the generated code via `profile-dump-list` and `profile-dump-html`.

Block counts are per *basic block*. Basic blocks are individual sequences of straight-line code and are the building blocks of the machine code generated by the compiler. Counting the number of times a block is executed is thus equivalent to counting the number of times the instructions within it are executed.

There is no mechanism for the programmer to view block counts, but both block counts and source counts can now be saved after a sample run of the generated code for use in guiding various optimizations during a subsequent compilation of the same code.

The source counts can be used by "profile-aware macros," i.e., macros whose expansion is guided by profiling information. A profile-aware macro can use profile information to optimize the code it produces. For example, a macro defining an abstract datatype might choose representations and algorithms based on the frequencies of its operations. Similarly, a macro, like `case`, that performs a set of disjoint tests might choose to order those tests based on which are most likely to succeed. Indeed, the built-in `case` now does just that. A new syntactic form, `exclusive-cond`, abstracts a common use case for profile-aware macros.

The block counts are used to guide certain low-level optimizations, such as block ordering and register allocation.

The procedure `profile-dump-data` writes to a specified file the profile data collected during the run of a program compiled with `compile-profile` set to either `source` or `block`. It is similar to `profile-dump-list` or `profile-dump-html` but stores the profile data in a machine readable form.

The procedure `profile-load-data` loads one or more files previously created by `profile-dump-data` into an internal database.

The database associates *weights* with source locations or blocks, where a weight is a flonum representing the ratio of the location's count versus the maximum count. When multiple profile data sets are loaded, the weights for each location are averaged across the data sets.

The procedure `profile-query-weight` accepts a source object and returns the weight associated with the location identified by the source object, or `#f` if no weight is associated with the location. This procedure is intended to be used by a profile-aware macro on pieces of its input to optimize code based on profile data previously stored by `profile-dump-data` and loaded by `profile-load-data`.

The procedure `profile-clear-data` clears the database.

The new `exclusive-cond` syntax is similar to `cond` except it assumes the tests performed by the clauses are disjoint and reorders them based on available profiling data. Because the tests might be reordered, the order in which side effects of the test expressions occur is undefined. The built-in `case` form is implemented in terms of `exclusive-cond`.

2.45. New `ssize_t` foreign type (9.0)

A new foreign type, `ssize_t`, is now supported. It is the signed analogue of `size_t`.

2.46. Guardian representatives (9.0)

When `make-guardian` is passed a second, *representative*, argument, the representative is returned from the guardian in place of the guarded object when the guarded object is no longer accessible.

2.47. Library reloading on dependency change (9.0)

A library initially imported from an object file is now reimported from source when a dependency (another library or include file) has changed since the library was compiled.

2.48. Expression-editor filename completion (8.9.5)

The expression editor now performs filename- rather than command-completion within string constants. It looks only at the current line to determine whether the cursor is within a string constant; this can lead to the wrong kind of command completion for strings that cross line boundaries.

2.49. New lock mechanisms and elimination of old lock mechanism (8.9.5)

The built in ftype `ftype-lock` has been eliminated along with the corresponding procedures, `acquire-lock`, `release-lock`, and `initialize-lock`. This is an incompatible change, although defining `ftype-lock` and the associated procedures is straightforward using the forms described below.

The functionality has been replaced and generalized by four new syntactic forms that operate on lock fields wherever they appear within a foreign type:

```
(ftype-init-lock! T (a ...) e)
(ftype-lock! T (a ...) e)
(ftype-spin-lock! T (a ...) e)
(ftype-unlock! T (a ...) e)
```

The access chain *a* `...` must specify a word-size integer represented using the native endianness, i.e., a `uptr` or `iptr`. It is a syntax violation when this is not the case.

For each of the forms, the expression *e* is evaluated first and must evaluate to a ftype pointer *p* of type *T*.

`ftype-init-lock!` initializes the specified field of the foreign object to which *p* points, puts the field into the unlocked state, and returns an unspecified value.

If the field is in the unlocked state, `ftype-lock!` puts it into the locked state and returns `#t`. If the field is already in the locked state, `ftype-lock!` returns `#f`.

`ftype-spin-lock!` loops until the lock is in the unlocked state, then puts it into the locked state and returns an unspecified value. *This operation will never return if no other thread or process unlocks the field, causing interrupts and requests for collection to be ignored.*

Finally, `ftype-unlock` puts the field into the unlocked state (regardless of the current state) and returns an unspecified value.

An additional pair of syntactic forms can be used when just an atomic increment or decrement is required:

`(ftype-locked-incr!` *T* `(`*a* `...)` *e*`)`
`(ftype-locked-decr!` *T* `(`*a* `...)` *e*`)`

As for the first set of forms, the access chain *a* `...` must specify a word-size integer represented using the native endianness.

2.50. `ftype-pointer-null?`, `ftype-pointer=?` (8.9.5)

The new procedure `ftype-pointer-null?` can be used to compare the address of its single argument, which must be an ftype pointer, against 0. It returns `#t` if the address is 0 and `#f` otherwise. Similarly, `ftype-pointer=?` can be used to compare the addresses of two ftype-pointer arguments. It returns `#t` if the address are the same and `#f` otherwise.

These are potentially more efficient than extracting ftype-pointer addresses first, which might result in bignum allocation for addresses outside the fixnum range, although the compiler also now tries to avoid allocation when the result of a call to `ftype-pointer-address` is directly compared with 0 or with the result of another call to `ftype-pointer-address`, as described in Section 4.16.

2.51. `gensym`'s new optional unique-name argument (8.9.5)

`gensym` now accepts a second optional argument, the unique name to use. It must be a string and should not be used by any other gensym intended to be distinct from the new gensym.

2.52. GC times now maintained with finer granularity (8.9.5)

In previous releases, collection times as reported by `statistics` or printed by `display-statistics` were gathered internally with millisecond granularity at each collection, possibly leading to significant inaccuracies over the course of many collections. They are now maintained using high-resolution timers with generally much better accuracy.

2.53. New time types for tracking collection times (8.9.5)

New time types `time-collector-cpu` and `time-collector-real` have been added. When `current-time` is passed one of these types, a time object of the specified type is returned and represents the time (cpu or real) spent during collection.

Previously, this information was available only via the `statistics` or `display-statistics` procedures, and then with lower precision.

2.54. New storage-management introspection procedures (8.9.5)

Three new storage-management introspection procedures have been added:

```
(collections)
(initial-bytes-allocated)
(bytes-deallocated)
```

`collections` returns the number of collections performed so far by the current Scheme process.

`initial-bytes-allocated` returns the number of bytes allocated after loading the boot files and before running any non-boot user code.

`bytes-deallocated` returns the total number of bytes deallocated by the collector.

Previously, this information was available only via the `statistics` or `display-statistics` procedures.

2.55. New time-object manipulation procedures (8.9.5)

Three new procedures for performing arithmetic on time objects have been added, per SRFI 19:

`(time-difference` *t1 t2*`)` ⇒ *t3*
`(add-duration` *t1 t2*`)` ⇒ *t3*
`(subtract-duration` *t1 t2*`)` ⇒ *t3*

`time-duration` takes two time objects *t1* and *t2*, which must have the same time type, and returns the result of subtracting *t2* from *t1*, represented as a new time object with type `duration`. `add-duration` adds time object *t2*, which must be of type `duration`, to time object *t1*, producing a new time object *t3* with the same type as *t1*. `subtract-duration` subtracts time object *t2* which must be of type `duration`, from time object *t1*, producing a new time object *t3* with the same type as *t1*.

SRFI 19 also names destructive versions of these operators:

`(time-difference!` *t1 t2*`)` ⇒ *t3*
`(add-duration!` *t1 t2*`)` ⇒ *t3*
`(subtract-duration!` *t1 t2*`)` ⇒ *t3*

These are available as well in *Chez Scheme* but are actually nondestructive, i.e., entirely equivalent to the nondestructive versions.

2.56. Better reporting of profile counts (8.9.4, 8.9.5)

The compiler now collects and reports profile counts for every source expression that is not determined to be dead either at compile time or by the time the profile information is obtained via `profile-dump-list` or `profile-dump-html`. Previously, the compiler suppressed profile counts for constants and variable references in contexts where the information was likely (though not guaranteed) to be redundant, and it dropped profile counts for some forms that were optimized away, such as inlined calls, folded calls, or useless code. Furthermore, profile counts now uniformly represent the number of times a source expression's evaluation was started, which was not always the case before.

A small related enhancement has been made in the HTML output produced by `profile-dump-html`. Hovering over a source expression now shows, in addition to the count, the starting position (line number and character) of the source expression to which the count belongs. This is useful for identifying when a

source expression does not have its own count but instead inherits the count (and color) from an enclosing expression.

2.57. Virtual registers (8.9.4)

A limited set of *virtual registers* is now supported by the compiler for use by programs that require high-speed, global, and mutable storage locations. Referencing or assigning a virtual register is potentially faster and never slower than accessing an assignable local or global variable, and the code sequences for doing so are generally smaller. Assignment is potentially significantly faster because there is no need to track pointers from the virtual registers to young objects, as there is for variable locations that might reside in older generations. On threaded versions of the system, virtual registers are "per thread" and thus serve as thread-local storage in a manner that is less expensive than thread parameters.

The interface consists of three procedures:

`(virtual-register-count)` returns the number of virtual registers. As of this writing, the count is set at 16. This number is fixed, i.e., cannot be changed except by recompiling *Chez Scheme* from source.

`(set-virtual-register!` k x`)` stores x in virtual register k. k must be a fixnum between 0 (inclusive) and the value of `(virtual-register-count)` (exclusive).

`(virtual-register` k`)` returns the value most recently stored in virtual register k (on the current thread, in threaded versions of the system).

To get the fastest possible speed out of the latter two procedures, k should be a constant embedded right in the call (or propagatable via optimization to the call). To avoid putting these constants in the source code, programmers should consider using identifier macros to give names to virtual registers, e.g.:

```
(define-syntax foo
  (identifier-syntax
    [id (virtual-register 0)]
    [(set! id e) (set-virtual-register! 0 e)]))
(set! foo 'hello)
foo  ⇒  hello
```

Virtual-registers must be treated as an application-level resource, i.e., libraries intended to be used by multiple applications should generally not use virtual registers to avoid conflicts with the applications use of the registers.

2.58. 24-, 40-, 48-, and 56-bit integer values (8.9.3)

Support for storing and extracting 24-, 40-, 48-, and 56-bit integers to and from records, bytevectors, and foreign types (ftypes) has been added. For records and ftypes, this is accomplished by declaring a field to be of type `integer-24`, `unsigned-24`, `integer-40`, `unsigned-40`, `integer-48`, `unsigned-48`, `integer-56`, or `unsigned-56`. For bytevectors, this is accomplished via the following new primitives:

```
bytevector-24-ref
bytevector-24-set!
bytevector-40-ref
bytevector-40-set!
bytevector-48-ref
bytevector-48-set!
bytevector-56-ref
bytevector-56-set!
```

Similarly, support has been added for sending and receiving 24-, 40-, 48-, and 56-bit integers to and from foreign code via `foreign-procedure` and `foreign-callable`. Arguments and return values of type `integer-24`

and `unsigned-24` are passed as 32-bit quantities, while those of type `integer-40`, `unsigned-40`, `integer-48`, `unsigned-48`, `integer-56`, and `unsigned-56` are passed as 64-bit quantities.

For unpacked ftypes, a 48-bit (6-byte) quantity is aligned on an even two-byte boundary, while a 24-bit (3-byte), 40-bit (5-byte), or 56-bit (7-byte) quantity is aligned on an arbitrary byte boundary.

2.59. New `pariah` expression (8.9.3)

A `pariah` expression:

`(pariah` *expr expr* `...)`

is syntactically similar and semantically equivalent to a begin expression but tells the compiler that the expressions within are relatively unlikely to be executed. This information is currently used by the compiler for prioritizing allocation of registers to variables and for putting pariah code out-of-line in an attempt to reduce instruction cache misses for the remaining code.

A `pariah` form is generally most usefully wrapped around the consequent or alternative of an `if` expression to identify which is the less likely path.

The compiler implicitly treats as pariah code any code that leads up to an unconditional call to `raise`, `error`, `errorf`, `assertion-violation`, etc., so it is not necessary to wrap a `pariah` around such a call.

At some point, there will likely be an option for gathering similar information automatically via profiling. In the meantime, we are interested in feedback about whether the mechanism is beneficial and whether the benefit of using the `pariah` form outweighs the programming overhead.

2.60. Improved automatic library recompilation (8.9.2)

Local imports within a library now trigger automatic recompilation of the library when the imported library has been recompiled or needs to be recompiled, in the same manner as imports listed directly in the importing library's `library` form. Changes in include files also trigger automatic recompilation.

(Automatic recompilation of a library is enabled when an import of the library, e.g., in another library or in a top-level program, is compiled and the parameter `compile-imported-libraries` is set to a true value.)

2.61. Redundant profile information (8.9.2)

Profiling information is no longer produced for constants and variable references where the information is likely to be redundant. It is still produced in contexts where the counts are likely to differ from those of the enclosing form, e.g., where a constant or variable reference occurs in the consequent or alternative of an `if` expression. This change brings the profiling information largely in sync with Version 8.4.1 and earlier, though Version 8.9.2 retains source information in a few cases where it is inappropriately discarded by Version 8.4.1's compiler, and Version 8.9.2 discards source information in a few cases where the code has been optimized away.

2.62. New `compile-to-port` procedure (8.9.2)

The procedure `compile-to-port` is like `compile-port` but, instead of taking an input port from which it reads expressions to be compiled, takes a list of expressions to be compiled. As with `compile-port`, the second argument must be a binary output port.

2.63. Debug levels (8.9.1)

Newly introduced debug levels control the amount of debugging support embedded in the code generated by the compiler. The current debug level is controlled by the parameter `debug-level` and must be set when the compiler is run to have any effect on the generated code. Valid debug levels are 0, 1, 2, and 3, and the default is 1. At present, the only difference between debug levels is whether calls to certain error-producing routines, like `error`, whether explicit or as the result of an implicit run-time check (such as the pair check in `car`), are treated as tail calls even when not in tail position. At debug levels 0 and 1, they are treated as tail calls, and at debug levels 2 and 3, they are treated as nontail calls. Treating them as tail calls is more efficient, but treating them as nontail calls leaves more information on the stack, which affects what can be shown by the inspector.

For example, assume `f` is defined as follows:

```
(define f
  (lambda (x)
    (unless (pair? x) (error #f "oops"))
    (car x)))
```

and is called with a non-pair argument, e.g.:

```
(f 3)
```

If the debug level is 2 or more at the time the definition is compiled, the call to `f` will still be on the stack when the exception is raised by `error` and will thus be visible to the inspector:

```
> (f 3)
Exception: oops
Type (debug) to enter the debugger.
> (debug)
debug> i
#<continuation in f>                                          : sf
  0: #<continuation in f>
  1: #<system continuation in new-cafe>
#<continuation in f>                                          : s
  continuation:          #<system continuation in new-cafe>
  procedure code:        (lambda (x) (if (...) ...) (car x))
  call code:             (error #f "oops")
  frame and free variables:
  0. x:                  3
```

On the other hand, if the debug level is 1 (the default) or 0 at the time the definition of `f` is compiled, the call to `f` will no longer be on the stack:

```
> (f 3)
Exception: oops
Type (debug) to enter the debugger.
> (debug)
debug> i
#<system continuation in new-cafe>                            : sf
  1: #<system continuation in new-cafe>
```

2.64. Cost centers (8.9.1)

Cost centers are used to track the bytes allocated, instructions executed, and/or cpu time elapsed while evaluating selected sections of code. Cost centers are created via the procedure `make-cost-center`, and costs are tracked via the procedure `with-cost-center`.

Allocation and instruction counts are tracked only for code instrumented for that purpose. This instrumentation is controlled by the `generate-allocation-counts` and `generate-instruction-counts` parameters. Instrumentation is disabled by default. Built in procedures are not instrumented, nor is interpreted code or non-Scheme code. Elapsed time is tracked only when the optional `timed?` argument to `with-cost-center` is provided and is not false.

The `with-cost-center` procedure accurately tracks costs, subject to the caveats above, even when reentered with the same cost center, used simultaneously in multiple threads, and exited or reentered one or more times via continuation invocation.

thread parameter: `generate-allocation-counts`

When this parameter has a true value, the compiler inserts a short sequence of instructions at each allocation point in generated code to track the amount of allocation that occurs. This parameter is initially false.

thread parameter: `generate-instruction-counts`

When this parameter has a true value, the compiler inserts a short sequence of instructions in each block of generated code to track the number of instructions executed by that block. This parameter is initially false.

procedure: `(make-cost-center)`

Creates a new `cost-center` object with all of its recorded costs set to zero.

procedure: `(cost-center?` *obj*`)`

Returns `#t` if *obj* is a `cost-center` object, otherwise returns `#f`.

procedure: `(with-cost-center` *cost-center thunk*`)`
procedure: `(with-cost-center` *timed? cost-center thunk*`)`

This procedure invokes *thunk* without arguments and returns its values. It also tracks, dynamically, the bytes allocated, instructions executed, and cpu time elapsed while evaluating the invocation of *thunk* and adds the tracked costs to the cost center's running record of these costs.

Allocation counts are tracked only for code compiled with the parameter `generate-allocation-counts` set to true, and instruction counts are tracked only for code compiled with `generate-instruction-counts` set to true. Cpu time is tracked only if *timed?* is provided and not false and includes cpu time spent in instrumented, uninstrumented, and non-Scheme code.

procedure: `(cost-center-instruction-count` *cost-center*`)`

This procedure returns instructions executed recorded by *cost-center*.

procedure: `(cost-center-allocation-count` *cost-center*`)`

This procedure returns the bytes allocated recorded by *cost-center*.

procedure: `(cost-center-time` *cost-center*`)`

This procedure returns the cpu time recorded by *cost-center*.

procedure: `(reset-cost-center!` *cost-center*`)`

This procedure resets the costs recorded by *cost-center* to zero.

2.65. Experimental access to hardware performance counters (8.9.1)

Two system primitives, `#%$read-time-stamp-counter` and `#%$read-performance-monitoring-counter`, provide access to the x86 and x86_64 hardware time-stamp counter register and to the model-specific performance monitoring registers.

These primitives rely on instructions that might be restricted to run only in kernel mode, depending on kernel configuration. The performance monitoring counters must also be configured to enable monitoring and to specify which event to monitor. This can be configured only by instructions executed in kernel mode.

procedure: `(#%$read-time-stamp-counter)`

This procedure returns the current value of the time-stamp counter for the CPU core executing this code. A general protection fault, which manifests as an invalid memory reference exception, results if this operation is not permitted by the operating system.

Since multiple processes might run on the same core between reads of the time-stamp counter, the counter does not necessarily reflect time spent only in the current process. Also, on machines with multiple cores, the executing process might be swapped to a different core with a different time-stamp counter.

procedure: (`#%$read-performance-monitoring-counter` *counter*)

This procedure returns the current value of the model-specific performance monitoring register specified by *counter*. *counter* must be a fixnum and should specify a valid performance monitoring register. Allowable values depend on the processor model. A general protection fault, which manifests as an invalid memory reference exception, results if this operation is not permitted by the operating system or if the specified counter does not exist.

In order to get meaningful results, the performance monitoring registers must be enabled, and the event to be monitored must by configured by the performance monitoring control register. This configuration can be done only by code run in kernel mode.

Since multiple processes might run on the same core between reads of a performance monitoring register, the register does not necessarily reflect only the activities of the current process. Also, on machines with multiple cores, the executing process might be swapped to a different core with its own set of performance monitoring registers and possibly a different configuration for those registers.

2.66. New inspector functionality (8.9.1)

Within the interactive inspector, closure and frame variables can now be set by name, and the forward (f) and back (b) commands can now be used to to move among the frames that comprise a continuation.

A new show-local (sl) command can be be used to look at just the local variables of a stack frame. This contrasts with the show (s) command, which shows the free variables of the frame's closure as well.

Errors occurring during inspection, such as attempts to assign immutable variables, are handled more smoothly than in previous versions.

2.67. Fasl support for records with non-ptr fields (8.4.1)

The fasl writer and reader now support records with non-ptr fields, e.g., integer-32, wchar, etc., allowing constant record instances with such fields to appear in source code (or be introduced as constants by macros) into code to be compiled via `compile-file`, `compile-library`, `compile-program`, `compile-script`, or `compile-port`. Ftype-pointer fields are not supported, since storing addresses in fasl files does not generally make sense.

3. Bug Fixes

3.1. `string->number` and reader numeric syntax issues (9.4)

`string->number` and the reader previously treated all complex numbers written in polar notation that Chez Scheme cannot represent exactly as inexact, even with an explicit `#e` prefix. For such numbers with the `#e` prefix, `string->number` now returns `#f` and the reader now raises an exception with condition type `&implementation-restriction`. Both still return an inexact representation for such numbers written without the `#e` prefix, even if R6RS requires an exact result, i.e., even if they have no decimal point, exponent, or mantissa width.

Ratios with an exponent, like `1/2e10`, are non-standard and now cause cause the procedure `string->number`

imported from `(rnrs)` to return `#f`. When the reader encounters a ratio followed by an exponent while in R6RS mode (i.e., when reading a library or top-level program and not following an `#!chezscheme`, or when following an explicit `#!r6rs`), it raises an exception.

Positive or negative zero followed by a large exponent now properly produces zero rather than an infinity, e.g., `0e3000` now produces `0` rather than `+inf.0`.

A rounding bug converting some small ratios into floating point numbers, when those numbers fall into the range of denormalized floats, has been fixed. This bug also affected the reading of and conversion of strings into denormalized floating-point numbers. [Some of these bugs dated back to Version 3.0.]

3.2. `date->time-utc` ignoring zone-offset field (9.4)

`date->time-utc` has been fixed to properly take into account the zone-offset field. [This bug dated back to Version 8.0.]

3.3. `dynamic-wind` mistakenly enabling interrupts (9.3.3)

A bug causing `dynamic-wind` to unconditionally enable interrupts upon a nonlocal exit from the body thunk has been fixed. Interrupts are now properly enabled only when the optional *critical?* argument is supplied and is not false. [This bug dated back to Version 6.9c.]

3.4. Incorrect optimization of various primitives (9.3.1)

Mistakes in our primitive database that caused the source optimizer to treat `append`, `append!`, `list*`, `cons*`, and `record-type-parent` as always returning true values have been fixed, along with mistakes that caused the source optimizer to treat `null-environment`, `source-object-bfp`, `source-object-efp`, and `source-object-sfd` as not requiring argument checks. [This bug dated back to Version 6.0.]

3.5. Increased allocation ceiling under 32-bit Windows (9.3.1)

We have worked around a limitation in the number of distinct allocation areas the Windows VirtualAlloc function permits to be allocated by allocating fewer, larger chunks of memory, effectively increasing the maximum size of the heap to the full amount permitted by the operating system.

3.6. Syntax errors for `let` and `let*` (9.2.1)

The expander now handles `let` and `let*` in such a way that certain syntax errors previously reported as syntax errors in `lambda` are now reported properly as syntax errors in `let` or `let*`. This includes duplicate identifier errors for `let` and errors involving internal definitions for both `let` and `let*`.

3.7. Dropped `profile-dump-html` calls (9.0)

A bug that caused effect-context calls to `profile-dump-html` to be dropped at optimize-level 3 has been fixed. [This bug dated back to Version 7.5.]

3.8. Proper treatment of imported meta bindings (8.9.3)

A deficiency in the handling of library dependencies that prevented meta definitions exported in one library from being used reliably by a macro defined in another library has been fixed. Handling imported meta bindings involves tracking visit-visit-requirements, which for a library `(A)` is the set of libraries that must

be visited (rather than invoked) when `(A)` is visited. An attempt to assign a meta variable imported from a library now results in a syntax error. [This bug dated back to Version 7.9.1.]

3.9. Reexport of identifiers with properties (8.9.3)

A bug that prevented an identifier given a property via `define-property` from being exported from a library `(A)`, imported into and reexported from a second library `(B)`, and imported from both `(A)` and `(B)` into and reexported from a third library `(C)` has been fixed. [This bug dated back to Version 8.1.]

3.10. Cyclic record-type descriptors (8.4.1)

The fasl (fast load) format used for compiled files now supports cyclic record-type descriptors (RTDs), which are produced for recursive ftype definitions. Previously, compiling a file containing a recursive ftype definition and subsequently loading the file resulted in corruption of the ftype descriptor used to typecheck ftype pointers, potentially leading to incorrect behavior or invalid memory references. [This bug dated back to Version 8.2.]

3.11. Invalid folding of record accesses (8.4.1)

A bug that caused the optimizer to fold calls to record accessors applied to a constant value of the wrong type, sometimes resulting in compile-time invalid memory references or other compile-time errors, has been fixed. [This bug dated back to Version 8.4.]

3.12. 4GB+ allocation for Windows x86_64 (8.4.1)

A bug that prevented objects larger than 4GB to be created under Windows x86_64 has been fixed. [This bug dated back to Version 8.4.]

4. Performance Enhancements

4.1. Improved oblist management (9.3.3)

As a result of improvements in the handing of the oblist (symbol table), the storage for a symbol is often reclaimed more quickly after it becomes inaccessible, less space is set aside for the oblist at start-up, oblist lookups are faster when the oblist contains a large number of symbols, and the minimum cost of a maximum-generation collection has been cut significantly, down from tens of microseconds to just a handful on contemporary hardware.

4.2. Reduced maximum-generation collection overhead (9.3.3)

Various changes in the storage manager have reduced the amount of extra memory required for managing heap storage and increased the likelihood that memory can be returned to the O/S as the heap shrinks. Returning memory to the O/S is now faster, so the minimum time for for a maximum-generation collection, or any other collection where release of memory to the O/S is enabled, has been cut.

4.3. Faster library load times (9.3.1)

Libraries now load faster at both compile and run time, with more pronounced improvements when dozens of libraries or more are being loaded.

4.4. Partially static record instances (9.3.1)

The source optimizer now maintains information about partially static record instances to eliminate field accesses and type checks when a binding site for a record instances is visible to the access or checking code. For example,

```
(let ()
  (import scheme)
  (define-record foo ([immutable ptr a] [immutable ptr b]))
  (define (inc r) (make-foo (foo-a r) (+ (foo-b r) 1)))
  (lambda (x)
    (let* ([r (make-foo 37 x)]
           [r (inc r)]
           [r (inc r)])
      r)))
```

is reduced by the source optimizer down to:

```
(lambda (x) ($record '#<record type foo> 37 (+ (+ x 1) 1)))
```

where `$record` is a low-level primitive for creating record instances. That is, the source optimizer eliminates the intermediate record structures, record references, and type checks, in addition to creating the record-type descriptor at compile time, eliminating the record-constructor descriptor, record constructor, and record accessors produced by expansion of the record definition.

4.5. More source-optimizer improvements (9.3.1)

The source optimizer now handles `apply` with a known-list final argument, e.g., a constant list or list constructed directly within the apply operation via `cons`, `list`, or `list*` (`cons*`) as if it were an ordinary call, i.e., without the `apply` and without the constant list wrapper or list constructor. For example:

```
(apply apply apply + (list 1 (cons 2 (list x (cons* 4 '(5 6))))))
```

folds down to `(+ 18 x)`. While not common at the source level, patterns like this can materialize as the result of other source optimizations, particularly inlining.

The source optimizer now also reduces applications of `car` and `cdr` to the list-building operators `cons` and `list`, e.g.:

```
(car (cons e1 e2))  →  (begin e2 e1)
(car (list e1 e2 e3))  →  (begin e2 e3 e1)
(cdr (list e1 e2 e3))  →  (begin e1 (list e2 e3))
```

discarding side-effect-free expressions in the `begin` forms where appropriate. It treats similarly calls of `vector-ref` on `vector`; `list-ref` on `list`, `list*`, and `cons*`; `string-ref` on `string`; and `fxvector-ref` on `fxvector`, taking care with `string-ref` and `fxvector-ref` not to optimize when doing so might mask an invalid type of argument to a safe constructor.

Finally, the source optimizer now removes certain unnecessary `let` bindings within the constraints of evaluation-order preservation. For example,

```
(let ([x e1] [y e2]) (list (cons x y) 7))
```

reduces to:

```
(list (cons e1 e2) 7)
```

Such bindings commonly arise from inlining. Eliminating them tends to make the output of `expand/optimize` more readable.

The impact on performance is minimal, but it can result in smaller expressions and thus enable more inlining within the same size limits.

4.6. Improved foreign-pointer address handling (9.3.1)

Various composed operation on ftypes now avoid allocating and dereferencing intermediate ftype pointers, i.e., `ftype-ref`, `ftype-set!`, `ftype-init-lock!`, `ftype-lock!`, `ftype-unlock!`, `ftype-spin-lock!`, `ftype-locked-incr!`, or `ftype-locked-decr!` applied directly to the result of `ftype-ref`, `ftype-&ref`, or `make-ftype-pointer`.

4.7. New source optimizations (9.2.1)

The source optimizer does a few new optimizations: it folds calls to `symbol->string`, `string->symbol`, and `gensym->unique-string` if the argument is known at compile time and has the right type; it folds zero-argument calls to `vector`, `string`, `bytevector`, and `fxvector`; and it discards subsumed case-lambda clauses, e.g., the second clause in `(case-lambda [(x . y)` e_1`] [(x y)` e_2`])`.

4.8. Reduced stack requirements after large apply (9.2)

A call to `apply` with a very long argument list can cause a large chunk of memory to be allocated for the topmost portion of the stack. This space is now reclaimed during the next collection.

4.9. Improved symbol-hashtables performance (9.2)

The performance of operations on symbol hashtables has been improved generally over previous releases by eliminating call overhead for the hash and equality functions. Further improvements are possible with the use of the new type-specific symbol-hashtable operators (Section 2.18).

4.10. Reduced library-invocation time, memory consumption (9.1)

The amount of time required to invoke a library and the amount of memory occupied by the library when the library is invoked as the result of a run-time dependency of another library or a top-level program have both been reduced by "revisiting" rather than "invoking" the library, effectively leaving the compile-time information on disk until if and when it is needed.

4.11. Discarding relocation tables for static code objects (9.1)

Unless the command-line parameter `--retain-static-relocation` is supplied, the collector now discards relocation tables for code objects when the code objects are promoted to the static generation, either at boot time via heap compaction or via a call to `collect` with the symbol `static` as the target generation. This results in a significant reduction in the memory occupied by the code object (around 20% in our tests).

4.12. Guardian registration (9.1)

The code to register an object with a guardian is now open-coded, at the cost of some additional work during the next collection. The result is a modest net improvement in registration overhead (around 15% in our tests). Of potentially greater importance when threaded, each registration no longer requires synchronization.

4.13. Generated code improvements (9.1)

The compiler generates better code in several small ways, resulting in small decreases in code size and corresponding small performance improvements in the range of 1–5% in our tests.

4.14. Reduced collector overhead for large heaps (9.0)

In previous releases, a factor in collector performance was the overall size of the heap (measured both in number of pages and the amount of virtual memory spanned by the heap). Through various changes to the data structures used to support the storage manager, this factor has been eliminated, which can significantly reduce the cost of collecting a younger generation with a small number of accessible objects relative to overall heap size. In our experiments, the minimum cost of collection on contemporary hardware exceeded 100 microseconds for heaps of 64MB or more and 5 milliseconds for heaps of 1GB or more. The minimum cost grew in proportion to the heap size from there. This is now fixed for all heap sizes at just a few microseconds.

4.15. Reduced mutation overhead (9.0)

Improvements in the compiler and storage manager have been made to reduce the cost of tracking possible pointers from older to younger generations when objects are mutated.

4.16. Improved foreign-pointer address handling (8.9.5)

Ftype pointers with constant addresses are now created at compile time, with ftype-pointer address checks optimized away as well.

Bignum allocation overhead is avoided for addresses outside the fixnum range when the results of two `ftype-pointer-address` calls are directly compared or the result of one `ftype-pointer-address` call is directly compared with 0. That is, comparisons like:

```
(= (ftype-pointer-address x) 0)
(= (ftype-pointer-address x) (ftype-pointer-address y))
```

are effectively optimized to:

```
(ftype-pointer-null? x)
(ftype-pointer=? x y)
```

This optimization is performed when the comparison procedure is `=`, `eqv?`, or `equal?` and the arguments are given in either order. The optimization is also performed when `zero?` is applied directly to the result of `ftype-pointer-address`.

Bignum allocation overhead is also avoided at optimize-level 3 when `ftype-pointer-address` is used in combination with `make-ftype-pointer` to effect a type cast, as in:

```
(make-ftype-pointer T (ftype-pointer-address x))
```

Both bignum and ftype-pointer allocation is avoided when the result of such a cast is used directly as the base pointer in an `ftype-ref`, `ftype-&ref`, `ftype-set!`, `ftype-locked-incr!`, `ftype-locked-decr!`, `ftype-init-lock!`, `ftype-lock!`, `ftype-spin-lock!`, or `ftype-unlock!` form, as in:

```
(ftype-ref T (fld) (make-ftype-pointer T (ftype-pointer-address x)))
```

These optimizations do not occur when the calls to `ftype-pointer-address` are not nested directly within the outer form, as when a `let` binding is used to name the result of the `ftype-pointer-address` call, e.g.:

```
(let ([addr (ftype-pointer-address x)]) (= addr 0))
```

In other places where `ftype-pointer-address` is used, the compiler now open-codes the extraction and (if necessary) bignum allocation, reducing overhead by the cost of a procedure call.

4.17. Improved performance when profiling (8.9.5)

In addition to improvements in the tracking of profile counts, the run-time overhead for gathering profile information has gone down by 5–10% in our tests and is now typically around 10% of the total unprofiled run time. (Unprofiled code is also slightly faster, but by less than 2% in our tests.)

4.18. New compiler back-end (8.9.1, 8.9.2, 8.9.5)

Versions starting with 8.9.1 employ a new compiler back end that is structured as a series of nanopassees and replaces the old linear-time register allocator with a graph-coloring register allocator. Compilation with the new back end is substantially slower (up to a factor of two) than with the old back end, while code generated with the new back end is faster (14–40% depending on architecture and optimization level) in our tests. These improvements are independent of improvements resulting from cross-library constant folding and inlining (Section 4.21). The code generated for a specific program might be faster or slower.

4.19. Open-coding of `make-guardian` (8.9.4)

Calls to `make-guardian` are now open-coded by the compiler to expose the implicit resulting `case-lambda` expression so that calls to the guardian can themselves be inlined, thus reducing the overhead for registering objects with a guardian and querying the guardian for resurrected objects.

4.20. Improved open-coding of `make-parameter` and `make-thread-parameter` (8.9.4)

`make-parameter` and `make-thread-parameter` are now open-coded in all cases to expose the implicit resulting `case-lambda` expression. (They were already open-coded when the second, *filter*, argument was a `lambda` expression or primitive name.)

4.21. Cross-library constant folding and inlining (8.9.2)

The compiler now propagates constants and inlines simple procedures across library boundaries. A simple procedure is one that, after optimization of the exporting library, is smaller than a given threshold, contains no free references to other bindings in the exporting library, and contains no constants that cannot be copied without breaking pointer identity. The size threshold is determined, as for inlining within a library or other compilation unit, by the parameter `cp0-score-limit`. In this case, the size threshold is determined based on the size *before* inlining rather than the size *after* inlining, which is often more conservative. Omitting larger procedures that might generate less code when inlined in a particular context reduces the amount of information that must be stored in the exporting library's object code to support cross-library inlining.

One particularly useful benefit of this optimization is that record predicates, accessors, mutators, and (depending on protocols) constructors created by a record definition in one library and exported by another are inlined in the importing library, just as if the record type were defined in the importing library.