Chez Scheme Version 9.9.9 Release Notes
September 2023

1. Overview

This document outlines the changes made to *Chez Scheme* for Version 9.9.9 since Version 8.4.

Version 9.9.9 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)
    - x86_64, nonthreaded (a6le) and threaded (ta6le)
    - AArch64 including Android, nonthreaded (arm64le) and threaded (tarm64le)
    - ARMv6 (32-bit) including Android, nonthreaded (arm32le) and threaded (tarm32le)
    - RV64G (64-bit RISC-V), nonthreaded (rv64le) and threaded (trv64le)
    - PowerPC (32-bit), nonthreaded (ppc32le) and threaded (tppc32le)
- Mac OS
    - x86, nonthreaded (i3osx) and threaded (ti3osx)
    - x86_64, nonthreaded (a6osx) and threaded (ta6osx)
    - AArch64 and iOS, nonthreaded (arm64osx) and threaded (tarm64osx)
    - PowerPC (32-bit) (ppc2osx) and threaded (tppc32osx)
- Windows
    - x86, nonthreaded (i3nt) and threaded (ti3nt)
    - Windows x86_64, nonthreaded (a6nt) and threaded (ta6nt)
    - AArch64, nonthreaded (arm64nt) and threaded (tarm64nt)
- OpenBSD
    - x86, nonthreaded (i3ob) and threaded (ti3ob)
    - x86_64, nonthreaded (a6ob) and threaded (ta6ob)
    - AArch64, nonthreaded (arm64ob) and threaded (tarm64ob)
    - ARMv6 (32-bit), nonthreaded (arm32ob) and threaded (tarm32ob)
    - PowerPC (32-bit), nonthreaded (ppc32ob) and threaded (tppc32ob)
- FreeBSD
    - x86, nonthreaded (i3fb) and threaded (ti3fb)
    - x86_64, nonthreaded (a6fb) and threaded (ta6fb)
    - AArch64, nonthreaded (arm64fb) and threaded (tarm64fb)
    - ARMv6 (32-bit), nonthreaded (arm32fb) and threaded (tarm32fb)
    - PowerPC (32-bit), nonthreaded (ppc32fb) and threaded (tppc32fb)
- NetBSD
    - x86, nonthreaded (i3nb) and threaded (ti3nb)
    - x86_64, nonthreaded (a6nb) and threaded (ta6nb)
    - AArch64, nonthreaded (arm64nb) and threaded (tarm64nb)
    - ARMv6 (32-bit), nonthreaded (arm32nb) and threaded (tarm32nb)
    - PowerPC (32-bit), nonthreaded (ppc32nb) and threaded (tppc32nb)
- GNU Hurd
    - x86, nonthreaded (i3gnu) and threaded (ti3gnu)
- OpenSolaris
    - x86, nonthreaded (i3s2) and threaded (ti3s2)
    - x86_64, nonthreaded (a6s2) and threaded (ta6s2)
- Other platforms
    - bytecode interpretation (pb, tpb, pb32l, tpb32l, pb32b, tpb32b, pb64l, tpb64l, pb64b, tpb64b)

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. New supported platforms and portable bytecode (9.9.9)

AArch64 (64-bit Arm) and RV64G (64-bit RISC-V) architectures are supported. Threads are supported on architectures with weak memory models while preserving the safety guarantee described in the documentation for variables and primitive datatypes.

The pb ("portable bytecode") machine type corresponds to a virtual machine that the Chez Scheme kernel can interpret on practically any desktop platform. This machine type is used to enable bootstrapping on all native-code platforms using a single set of boot files. The pb machine type is also used to support platforms for which Chez Scheme does not have a supported native-code generator. For the latter use, the pb32l, pb32b, pb64l, and pb64b machine types specialize the interpreter to a word size and endianness, which improves performance. For example, pb32l is a suitable machine type for compiling to WebAssembly via Emscripten. Each pb variant has a corresponding threaded variant: tpb, tpb32l, tpb32b, tpb64l, and tpb64b.

The `pbchunk-convert-file` procedure can convert sequences of static pb bytecode into C code. The generated C code can be compiled and linked with the Chez Scheme kernel to reduce the overhead of bytecode interpretation. In particular, since Chez Scheme is mostly written in Chez Scheme, a pbchunk conversion of its boot files improves performance of primitives and compilation.

2.2. Threaded default and build system changes (9.9.9)

Running `configure` assumes a threaded target machine type, unless `--nothreads` or a non-threaded machine type is specified.

The build system mostly uses Zuo instead of Make, but the `configure` script continues to generate wrapper makefiles, so that there's no difference for most users who build from source and do not modify Chez Scheme. The build process on Windows has changed so that Unix-style tools are not required, but using Unix-style tools to compile with MinGW is supported.

The Chez Scheme sources no longer include boot files for multiple platforms. Instead, a single set of pb boot files is normally used to build on any platform. This change makes an initial build of Chez Scheme slower, but it makes the build process consistent across all supported platforms and enables a much smaller source size. Alternatively, Chez Scheme can be bootstrapped from an older version of Chez Scheme (9.5.4 or later) using the `re.boot` makefile target.

The new `scheme-build-number` function complements `scheme-version-number`, adding an extra level of versioning during development. The `scheme-build-number` procedure always returns zero for a released version of Chez Scheme.

2.3. Compiler improvements (9.9.9)

The compiler generates code that locally unboxes floating-point operations, so that compositions of floating-point operations can be much faster.

The compiler infers information about the type of value produced by an expression, and it uses that information to eliminate run-time checks or conditionals. For example, within (`if` (`flonum? x`) *E1 E2*), assuming that the variable `x` is never mutated, then it holds a flonum value within *E1*, so arithmetic can

be specialized to flonum operations. Type inference requires an additional pass for compilation, but when compiling in safe mode, the pass tends to more than pay for itself; reduced checks and error handling create less work for later compilation passes. Even when many checks are removed, run-time performance may improve only modestly on modern hardware, since the removed branches are perfectly predictable. Inference can be disabled by setting `enable-type-recovery` to `#f`.

The compiler can infer more than previously about procedures that always return a value or that never return (because they raise a noncontinuable exception). The compiler will no longer move an expression from a non-tail position to a tail position if that movement could be detected with continuation marks (see Section 2.5), and it will not change a program in a way that discards an error due to returning multiple values to a single-value context. The `assert-unreachable` procedure always reports an error in safe mode, but it in unsafe mode, it must never be called; the claim that some code is unreachable propagates backward to remove other code that must also be unreachable, and even drop conditionals that must always go the other way to avoid unreachable code. A macro expansion might use `assert-unreachable` to indicate that unsafe mode can assume that a check succeeded and eliminate the check, for example.

The compiler consistently "lifts" procedures: When a procedure has only known call sites, refers to only immutable variables bound outside the procedure, and has only call site within the scope of those variables, then variables used by the procedure are converted to arguments, and call sites are converted to supply the variables as arguments. As a result, nested procedure definitions can be used more freely without a potential cost from closure allocation. There are programs where forming a closure would be faster, but consistent lifting is almost always faster in practice.

Cross-library inlining can apply to a procedure that is exported from a library (explicitly or implicitly) and whose body refers to another procedure or other variable that is also exported (explicitly or implicitly).

The compiler reduces code size by using a more compact representation for most call-return sites and by using a more compact dispatch to interrupt checking in most cases.

The internal representation of record types has changed so that a record-type predicate is a constant-time operation, instead of proportional to the subtype depth.

2.4. Garbage-collector improvements (9.9.9)

Garbage collection takes advantage of machine parallelism. Parallel collection is automatically enabled when a program has multiple Scheme threads active at the point that a garbage collection starts, but parallel collection can benefit from a hint about which threads are most relevant as allocators (see Section 2.18).

The garbage collector uses a hybrid mark–copy algorithm, which can improve performance for large objects and long-lived objects compared to a pure copying collector. The `in-place-minimum-generation` parameter tunes the hybrid by selecting the youngest generation at which marking is used.

The `collect-maximum-generation-threshold-factor` parameter is used by `collect` when called with no arguments. In the case that the maximum generation would be collected based on the gc-trip counter, it is collected only when the total memory use is at least k times the memory use just after the most recent collection of the maximum generation, where the parameter's value is k. The default value of 2 helps the collector adapt to the scenario that a program has much more long-lived data than short-lived data, and where the program does not explicitly promote its long-lived data to the static generation. Setting the parameter to 0 makes `collect` behave as in previous versions of Chez Scheme.

See also Sections 2.19 and 2.20 for related new functionality.

2.5. Continuation marks (9.9.9)

Continuation marks add key–value information to continuations. The `with-continuation-mark` form associates a key–value pair to the current continuation, replacing any same-keyed mark already associated to the immediate continuation, and adding to marks associated with continuations that the current continuation extends. Inspect marks with these functions:

```
current-continuation-marks
continuation-next-marks
continuation-marks-first
continuation-marks->list
continuation-marks->iterator
call-with-immediate-continuation-mark
continuation-marks?
```

The `call-in-continuation` function can invoke a continuation similar to calling a procedure as a function, but instead of a value to deliver to the continuation, it takes a procedure of no arguments to call within the continuation.

2.6. New fixnum operations (9.9.9)

New operations support common bit-related operations on fixnums:

```
fx+/wraparound
fx-/wraparound
fx*/wraparound
fxsll/wraparound
fxpopcount
fxpopcount32
fxpopcount16
```

Immutable fixnum vectors are no longer supported, so the following functions are removed:

```
fxvector->immutable-fxvector    ; removed
mutable-fxvector?               ; removed
immutable-fxvector?             ; removed
```

2.7. New flonum operations (9.9.9)

The `eq?` operator works on flonums in the sense that two flonums values that are `eq?` at some point (such as creation time) will continue to be `eq?` in the future.

Mutable flonum vectors cooperate with local unboxing (see Section 2.3) and are supported through the following functions:

```
flvector?
make-flvector
flvector-length
flvector-ref
flvector-set!
flvector-fill!
flvector->list
list->flvector
flvector-copy
```

The `flsingle` operation drops precision from a flonum so that the result is representable as a single-precision (32-bit) flonum. Wrapping each operation of flonum arithmetic with `flsingle` produces the same result as single-precision arithmetic, and local unboxing plus type inference makes the combination reasonably efficient.

Flonum printing can be controlled by new parameters:

```
print-subnormal-precision
print-positive-exponent-sign
print-select-flonum-exponential-format
```

The `fl=`, `fl<`, , `fl>`, `fl<=`, and `fl>=` no longer produce `#f` for a single argument that's `+nan.0`, which makes them consistent with `=`, `<`, `>`, `<=`, and `>=`.

2.8. Improved bignum arithmetic (9.9.9)

Multiplication of very large numbers uses Karatsuba, Toom-3, or Toom-4, while division (including GCD) uses Burnikel-Ziegler. These changes make arithmetic and number-to-string conversions faster for large values, and they avoid unbounded delays between interrupt polling.

2.9. Exact zeros, transcendentals, and exponentials (9.9.9)

Division of an inexact number by an exact `0` now raises exception instead of producing `+inf.0` or `-inf.0`. This makes division more consistent with with the existing behavior of multiplication, where multiplication of an inexact number by an exact `0` produces an exact `0`.

Small changes to transcendental functions help improve consistency for derived compositions. The `expt` function with two `+inf.0` or `-inf.0` arguments now produces `+nan.0+nan.0i` instead of a complex number with infinite components. The `sqrt` and `log` functions change so that (`sqrt -0.0`) and (`log -0.0`) produce real numbers, instead of complex numbers, on the grounds that (`negative? -0.0`) produces `#f`.

The `expt` function recognizes an exact `1/2` as its second argument, and in that case it behaves like `sqrt`, which may produce an exact result.

2.10. New character, string, and Unicode functions (9.9.9)

New functions support decoding grapheme clusters within strings or character streams and expose the underlying new character classifiers:

```
string-grapheme-span
string-grapheme-count
char-grapheme-step
char-grapheme-break-property
char-extended-pictographic?
```

The `string-append-immutable` function enables the creation of an immutable string as the result of appending strings without creating an intermediate mutable string.

The `path-build` procedure combines two path strings to form a path, adding a directory separator between them if necessary.

2.11. Hashtable changes (9.9.9)

Operations returning hashtable cells (`hashtable-cell`, `eq-hashtable-cell`, `symbol-hashtable-cell`, and `hashtable-cells`) now raise an exception when given an immutable hashtable. This is an *incompatible change*, as they previously accepted immutable hashtables (and allowed changing them).

The `make-weak-hashtable` and `make-ephemeron-hashtable` functions support weak and ephemeron hash tables with arbitrary equality and hashing functions.

New operations for hashtable cells resemble procedures like `hashtable-keys` or `hashtable-ref`:

```
hashtable-cells
hashtable-ref-cell
eq-hashtable-ref-cell
symbol-hashtable-ref-cell
```

The `eq-hashtable-try-atomic-cell` procedure supports lock-free use of an `eq?`-based hash table, but with constraints on concurrent operations and resizing. Clean-up resizing can be performed within a collect-request handler, since only one thread can run at that time.

A weak or ephemeron `eqv?` hash table now retains non-fixnum numbers only weakly, instead of strongly. This change is related to changes to `eq?` for flonum values and the availability of weak and ephemeron hash tables with arbitrary equality and hashing functions.

2.12. Stencil vectors (9.9.9)

Stencil vectors provide support in the compiler and runtime for implementing data structures such as persistent maps:

```
stencil-vector
stencil-vector?
stencil-vector-mask
stencil-vector-length
stencil-vector-ref
stencil-vector-set!
stencil-vector-update
stencil-vector-truncate!
stencil-vector-mask-width
```

2.13. New symbol functions (9.9.9)

While gensyms support most symbol-generation needs, uninterned symbols are useful for purposes where properties must not prevent a symbol from being reclaimed by the storage manager:

```
string->uninterned-symbol
uninterned-symbol?
```

The string returned by `symbol->string` is always immutable.

2.14. Record anonymous fields (9.9.9)

Records can have anonymous fields, which can save memory in the representation when many record types need to be created (as opposed to many record instances of a few types), each with many fields. For a given record type and its ancestors, either all fields are named or all fields are anonymous.

```
make-record-type-descriptor*
record-type-named-fields?
record-type-field-indices
```

The new `record-instance?` predicate is a specialization of `record?`, where the first argument is required to be a record. An unsafe `record-instance?` test can be faster than an unsafe `record?` test.

2.15. Lists assuming immutability (9.9.9)

In a context where pairs are not mutated, the new `list-assuming-immutable?` predicate is useful as a variant of the `list?` predicate. The new predicate implements an efficient, amortized constant-time decision on whether a value represents a list, but its behavior is unspecified if the `cdr` or any pair relevant to the result is mutated.

2.16. New random number generation (9.9.9)

A new random-number API implements the MRG32K3A algorithm:

```
make-pseudo-random-generator
pseudo-random-generator?
pseudo-random-generator-next!
pseudo-random-generator-seed!
pseudo-random-generator->vector
vector->pseudo-random-generator
```

2.17. Wrapper procedures (9.9.9)

A *wrapper procedure* provides an inexpensive way to an adjust a procedure's name, constrain its arity, or associate extra data to the procedure. A wrapper procedure is implemented as closure with an optimized jump to the wrapped procedure after arity checking, propagating the original arguments faster than `apply`.

```
make-wrapper-procedure
make-arity-wrapper-procedure
wrapper-procedure?
wrapper-procedure-procedure
wrapper-procedure-data
set-wrapper-procedure-data!
set-wrapper-procedure!
```

2.18. New thread functions (9.9.9)

The new `thread-join` operator can be used to wait for a thread to terminate. Waiting for termination in this sense can be useful to ensure that a single thread is running and calling `collect` is legal, for example.

The new `get-initial-thread` procedure returns a descriptor for the initial thread (like a descriptor produced by `fork-thread` for a subsequently created thread).

The new `thread-preserve-ownership!` procedure provides a hint to the storage manager that parallel garbage collection is likely to benefit by keeping track of which objects were allocated by a given thread.

Useful on architectures with a weak consistency model, `memory-order-acquire` and `memory-order-release` implement fencing operators suitable for abstractions that acquire (load-load and load-store fence) or release (store-store and store-load fence) shared resources.

2.19. Garbage collection introspection (9.9.9)

Similar to `enable-object-counts`, the `enable-object-backreferences` parameter enables recording of information about reachability. After a collection with backreferences enabled, `object-backreferences` reports an association for each object to the a referencing object—one that caused the storage manager to consider the former object reachable.

A new `bytes-finalized` procedure reports the number of bytes that have been finalized via guardians, which is useful for deciding when to perform an extra major collection as a follow-up to an old-generation garbage collection.

A *phantom bytevector* is a small object that stands for a large externally allocated object, especially one that can be finalized. Registering external allocations with the storage manager helps it trigger collection requests at times that take into account the rate and scale of external allocation.

```
make-phantom-bytevector
phantom-bytevector?
phantom-bytevector-length
set-phantom-bytevector-length!
```

The new `compute-size-increments` procedure is similar to `compute-size`. Instead of a single object, it takes a list of objects, and it reports a list of sizes where any memory use attributed to objects earlier in the list is not counted for objects later in the list, while objects later the list are not considered reachable by objects earlier in the list. For example, given a list of threads, `compute-size-increments` effectively treats each thread as an accounting domain, where memory is charged to an earlier thread rather than a later thread when objects are reachable from both. Since this computation involves the same sort of traversal as a garbage collection, the `collect` function takes a list as an optional last argument to fuse a garbage collection with size accounting.

2.20. Storage management and foreign interfaces (9.9.9)

The `make-guardian` function accepts an option to create an *ordered guardian*. An ordered guardian treats each of its objects as accessible in the case that object is reachable from a guardian-registered object, whether or not the latter object is considered accessible. Ordered finalization is error-prone and cannot handle reference cycles, but it can be necessary to implement certain storage-management interfaces and abstractions.

An *immobile object* is one that the storage manager will not relocate as a long as it is referenced, in the same way that a locked object is never relocated. Unlike a locked object, an immobile object can be reclaimed by the storage manager.

```
box-immobile
make-immobile-vector
make-immobile-bytevector
```

For foreign interfaces that involve arrays of references to allocated objects, the storage manager supports *reference bytevectors*. A reference bytevector is a bytevector, but each word-aligned group of bytes is treated as a potential reference to an object managed by the collector; only a word whose value corresponds a region of managed memory is treated as an object reference, and it can be updated by the garbage collector if the referenced object is relocated. Using the reference-bytevector interface requires care, but it can greatly simplify certain foreign-library interactions. An object reference in a bytevector takes the form of a *reference address* for a Scheme object. The reference address of a bytevector (including a reference bytevector) or flvector is the address of the first byte of its content.

```
make-reference-bytevector
make-immobile-reference-bytevector
reference-bytevector?
bytevector-reference-set!
bytevector-reference-ref
bytevector-reference*-ref
object->reference-address
reference-address->object
reference*-address->object
```

The `keep-live` procedure accepts any value and returns `(void)`, but the argument to `keep-live` is defined to be reachable until the call to `keep-live` returns.

The maximum non-static generation for collection has been reduced from 254 to 6. This change is related to parallel garbage collection and internal changes that make allocation thread-local, which in turn makes the size of a thread's representation proportional to the maximum number of generations.

2.21. Foreign interface extensions (9.9.9)

Some ABIs treat functions with varargs (i.e., specified with `...` in the C prototype) differently than functions without varargs, and some ABIs treat specific arguments differently depending on whether the argument is a vararg (i.e., before the `...` or not). A procedure type described with `foreign-procedure` or a `function` type can include a `__varargs` or (`__varargs_after` n) convention to indicate which arguments of a function are varargs.

The new `foreign-alignof` procedure reflects alignment information for a primitive type.

To better support the pb machine type, where endianness is not known statically, endianness in a foreign type can not be `native` or `swapped` in addition to `big` or `little`.

Chez Scheme's kernel also expose record-type and field access to C, mainly useful for for records that contain only pointer-sized values:

```
Srecord_type
Srecord_type_parent
Srecord_type_size
Srecord_type_uniformp
Srecord_uniform_ref
```

The kernel also provides a new function `Sregister_boot_file_fd_segment` for loading a boot file from a file descriptor and offset. This form of boot-file registration is useful for loading boot files that are embedded with an executable segment.

2.22. Fasl and vfasl (9.9.9)

Reading fasl data has been made safe no matter how deeply nested the structure of the data.

For reading and writing fasl data, the `fasl-write` procedure accepts a predicate to detect "external" values for which only a placeholder is saved, and `fasl-read` accepts a table of external values to substitute for place-holders. These "external" values can then have their own serialization and deserialization. The `fasl-write` function also accepts an option to save a record type as only its unique ID, which can save space and time in contexts where the relevant record types will always be available already (and where savings are worth skipping a consistency check). The `compile-to-port` procedure accepts new arguments like `fasl-write`.

A fasl stream can be converted a *vfasl* (very fasl) format, which is close in structure to an image of data in memory. Using vfasl for boot files to load directly into a static generation can make startup much faster; otherwise, the time–space tradeoff rarely pays off. Convert from fasl to vfasl using `vfasl-convert-file`.

2.23. New compiler options (9.9.9)

Two new parameters skip safety checks in specific situations: `enable-unsafe-application` assumes that the target of a procedure application is a procedure, and `enable-unsafe-variable-reference` assumes that a variable has a value when it is referenced. These checks would be skipped at `optimize-level` 3, but the parameters cause them to be skipped independent of the value of `optimize-level`.

A true value for the `enable-arithmetic-left-associative` parameter ensures that arithmetic operations are performed as left-associative, which amounts to a constraint on optimizations.

A new set of parameters control the way that libraries are compiled and invoked so that consistency checks, invocations checks, and recompilations can be skipped (to save time and space) in a context that ensures that they are unnecessary:

```
library-timestamp-mode
expand-omit-library-invocations
compile-omit-concatenate-support
```

Along those lines, the `current-generate-id` parameter provides control over names that are generated during macro expansion. While the default parameter uses `gensym`, name generation can potentially be made deterministic, which in turn may reduce the need to recompile clients.

The new `procedure-known-single-valued?` predicate is always allowed to return `#f`, but it may return `#t` for a procedure that the compiler was able to prove always returns a single value. This information may, in turn, be useful for run-time specialization in a library.

Procedure objects recognize the `realm` inspector message alongside `name`. The result for `realm` is normally `#f`, but if `compile-procedure-realm` is set to a symbol when the procedure is compiled, then the procedure answers `realm` with that symbol. The intent of a realm is that it describes where a procedure name came from, so the name can be converted as appropriate for a run-time context that uses different naming conventions.

The `enable-error-source-expression` parameter determines whether error messages that become embedded in code can refer to the original source file's path.

2.24. New conversions from Scheme to C signed and unsigned integers (9.6.4)

The following new functions allow foreign code to try converting Scheme values to signed or unsigned integer values without triggering a `longjmp` in the Scheme runtime as can happen when calling the corresponding functions `Sinteger_value`, etc:

```
Stry_integer_value
Stry_integer32_value
Stry_integer64_value
Stry_unsigned_value
Stry_unsigned32_value
Stry_unsigned64_value
```

2.25. New types for code that uses C exports (9.6.4)

The header file scheme.h distributed with Chez Scheme now defines `Sint32_t`, `Suint32_t`, `Sint64_t`, and `Suint64_t` types for 32-bit and 64-bit signed and unsigned integers that are compatible with the types for exports such as `Sinteger64`.

2.26. New transcoded port buffer-size parameters (9.6.0)

The new parameter `transcoded-port-buffer-size` specifies the size of the string buffer that is allocated when creating a new transcoded port. If the underlying binary port implements `port-position`, a transcoded input port allocates an internal fxvector the same size as its string buffer for use by `port-position`. The new parameter `make-codec-buffer` can be used to supply an appropriately sized internal bytevector buffer for the codec used by a new transcoded port. The size of these string and bytevector buffers was previously hardcoded at 1024.

2.27. Unicode 15.0 support (9.6.0)

The character sets, character classes, and word-breaking algorithms for character, string, and Unicode-related bytevector operations have now been updated to Unicode 15.0.

2.28. Basic ftypes can be referenced, even if shadowed by syntactic binding (9.5.8)

Previously, it was possible to interfere with the definition of ftypes by creating a syntactic binding for one of the built-in types, such as `integer-32`, `float`, etc. As of 9.5.8, syntactic bindings that do not bind an

ftype descriptor are no longer considered when defining ftypes.

This change also allows a base ftype to be bound using `define-ftype`, though this fixes the endianness of the type. For instance:

```
(define-ftype integer-32 integer-32)
```

This binds the ftype `integer-32` to the native-endian `integer-32`. It is possible to bind both endiannesses by using explicit names:

```
(define-ftype integer-32-be (endian big integer-32))
(define-ftype integer-32-le (endian little integer-32))
(define-ftype integer-32 integer-32) ;; fixed to native endianness
```

2.29. Improved error messages (9.5.6)

When the reader reports an invalid bytevector element, the error message now includes the token value only if the token type is atomic.

When the expander reports that an ellipsis is missing in a syntax form, it now includes the name of an identifier that is missing an ellipsis within that form.

2.30. Additional reader syntax for booleans (9.5.6)

The reader now case-insensitively accepts `#true` and `#false` as alternative spellings of the booleans `#t` and `#f`, respectively.

2.31. Self-evaluating vector literals (9.5.6)

The new parameter `self-evaluating-vectors` can be used to treat unquoted vector literals as self-evaluating instead of syntax errors. This parameter is turned off by default.

2.32. Incremental promotion of collected objects (9.5.4)

In previous versions of *Chez Scheme*, the collector always promoted surviving objects from every collected generation into a single target generation. For example, when the target generation was 3, it promoted not only surviving objects from generation 2 to generation 3 but also surviving objects from generations 0 and 1 directly to generation 3. This caused some prematurely promoted objects to be subjected to collection less frequently than their ages justified, potentially resulting in substantial inappropriate storage retention. This is particularly problematic when side effects result in pointers from the inappropriately retained objects to younger objects, as can happen with nonfunctional queues and lazy streams.

Unless directed to do otherwise, the collector now promotes objects up only one generation at a time. That is, generation 0 objects that survive collection are promoted to generation 1, generation 1 objects are promoted to generation 2, and so on. (Objects that survive a maximum nonstatic collection are promoted back into the maximum nonstatic collection.) Most applications should exhibit lower peak memory usage and possibly lower run times with this change. Applications that are adversely affected, if any, might benefit from a custom collect-request handler or custom values for the collection parameters that affect the behavior of the default handler.

2.33. Unicode Basic Multilingual Plane console I/O in Windows (9.5.4)

Console I/O now supports characters from the Unicode Basic Multilingual Plane in Windows. Windows consoles do not yet support the supplementary planes.

2.34. Incompatible fasl-format and compiled-file compression changes (9.5.4)

The fasl (fast-load) format now supports per-object compression. Whether the fasl writer actually performs compression is determined by the new `fasl-compressed` parameter, whose value defaults to `#t`. The compression format and level are determined by the `compress-format` and `compress-level` parameters.

The `compile-compressed` parameter has been eliminated. Since compiled files are written in fasl format, the `fasl-compressed` parameter also now controls whether compiled files are compressed.

Because individual portions of a fasl file are already compressed by default, attempting to compress a fasl file as a whole is often ineffective as well as inefficient both when writing and reading fasl objects. Thus, in particular, the *output-port* and `wpo-port` supplied to `compile-port` and `compile-to-port` should not be opened for compression. Similarly, external tools should not expect compiled files to be compressed as a whole, nor should they compress compiled files.

Because compression of fasl files was previously encouraged and is now discouraged, the first attempt to write fasl data to or read fasl data from a compressed port will cause a warning to be issued, i.e., an exception with condition type `&warning` to be raised.

The rationale for this change is to allow the fasl reader to seek past, without reading, portions of an object file that contain compile-time code at run time and run-time code at compile time.

2.35. Bytevector compression and compression level (9.5.4)

The procedure `bytevector-compress` now selects the level of compression based on the `compress-level` parameter. Prior to this it always used a default setting for compression.

The `compress-level` parameter can now take on the new value `minimum` in addition to `low`, `medium`, `high`, and `maximum`.

2.36. Combining object files (9.5.4)

In previous versions of Chez Scheme, multiple object files could be combined by concatenating them into a single file. To support faster object file loading and loadability verification (described later in this document), recompile information and information about libraries and top-level programs within an object file is now placed at the top of the file. The new `concatenate-object-files` procedure can be used to combine multiple object files while moving this information to the top of the combined file.

2.37. Explicitly invoking libraries (9.5.4)

The new procedure `invoke-library` can be used to force the evaluation of a library's body expressions (variable definition right-hand sides and initialization expressions) before they might otherwise be needed. It is generally useful only for libraries whose body expressions have side effects.

2.38. Verifying loadability of libraries and programs (9.5.4)

The new procedure `verify-loadability` can be used to determine, without actually loading any object code or defining any libraries, whether a set of object files and the object files satisfying their library dependencies, direct or indirect, are present, readable, and mutually compatible.

To support loadability verification, information about libraries and top-level programs within an object file is now placed at the top of the file, just after recompile information. This change can be detected by unusual setups, e.g., a source file that interleaves library definitions and top-level forms that call library-list, but is backward compatible for standard use cases in which each file contains one or more libraries possibly followed by a top-level program.

2.39. Unregistering objects from guardians (9.5.4)

The set of as-yet unresurrected objects registered with a guardian can be unregistered and retrieved by means of the new primitive `unregister-guardian`. Consult the user's guide for usage and caveats. Guardians can now be distinguished from other procedures (and other objects) via the new primitive `guardian?`.

2.40. Coverage support and source tables (9.5.4)

When the new parameter `generate-covin-files` is set to `#t` rather than the default `#f`, file compilation routines such as `compile-file` and `compile-library` produce coverage information (`.covin`) files that can be used in conjunction with profile information to measure coverage of a source-code base. Coverage information is also written out when the optional *covop* argument is supplied to `compile-port` and `compile-to-port`.

A covin file contains a printed representation of a *source table* mapping each profiled source object in the code base to a count of zero. Source tables generally associate source objects with arbitrary values and are allocated and manipulated with hashtable-like operations specific to source tables.

Profile information can be tracked even through releasing and clearing of profile counters via the new procedure `with-profile-tracker`, which produces a source table.

Coverage of a source-code base can thus be achieved by comparing the set of source objects in the covin-file source tables for one or more source files with the set of source objects in the source tables produced by one or more runs of tests run with profile information tracked by `with-profile-tracker`.

2.41. Importing a library from an object file now visits the file (9.5.4)

As described in Section 4.4, importing a library from an object file now causes the object file to be visited rather than fully loaded. If the run-time information is needed, i.e., if the library is invoked, the file will be revisited. This is typically transparent to the program, but problems can arise if the program changes its current directory (via `current-directory`) prior to invoking a library, and the object file cannot be found.

2.42. Recompile information (9.5.4)

As described in Section 4.4, all recompile information is now placed at the front of each object file where it can be read without the need to scan through the remainder of the file. Because the library manager expects to find recompile information at the front of an object file, it will not find all recompile information if object files are concatenated together via some mechanism other than then new `concatenate-object-files` procedure.

Also, the compiler has to hold in memory the object code for all expressions in a file so that it can emit the unified recompile information, rather than writing to the object file incrementally, which can significantly increase the memory required to compile a large file full of individual top-level forms. This does not affect top-level programs, which were already handled as a whole, or a typical library file that contains just a single library form.

2.43. Optional new `fasl-read` situation argument (9.5.4)

It is now possible to direct `fasl-read` to read only visit (compile-time) or revisit (run-time) objects via the optional new situation argument. Situation `visit` causes the fasl reader to skip over revisit (run-time-only) objects, while `revisit` causes the fasl reader to skip over visit (compile-time-only) objects. Situation `load` doesn't skip over any objects.

2.44. Optional `read-token` *sfd* and *bfp* arguments (9.5.4)

In addition to the optional input-port argument, `read-token` now takes optional *sfd* (source-file-descriptor) and *bfp* (beginning-file-position) arguments. If either is provided, both must be provided. Specifying *sfd* and *bfp* improves the quality of error messages, guarantees the `read-token` *start* and *end* return values can be determined, and eliminates the overhead of asking for a file position on each call to `read-token`. *bfp* is normally 0 for the first call to `read-token` at the start of a file, and the *end* return value of the preceding call for each subsequent call.

2.45. Compression format and level (9.5.4)

Support for LZ4 compression has been added. LZ4 is now the default format when compressing files (including object files produced by the compiler) and bytevectors, while `gzip` is still supported and can be enabled by setting the new `compress-format` parameter to the symbol `gzip` instead of the default `lz4`. Reading in compressed mode infers the format, so reading `gzip`-compressed files will still work without changing `compress-format`. Reading LZ4-format files tends to be much faster than reading `gzip`-format files, while `gzip`-compressed files tend to be smaller. In particular, object files created by the compiler now tend to be larger but load more quickly.

The new `compress-level` parameter can be used to control the amount of time spent on file and bytevector compression. It can be set to one of the symbols `minimum`, `low`, `medium`, `high`, and `maximum`, which are listed in order from shortest to longest compression time and least to greatest effectiveness. The default value is `medium`.

2.46. Mutexes and condition variables can have names (9.5.4)

The procedures `make-mutex` and `make-condition` now accept an optional argument `name`, which must be a symbol that identifies the object or `#f` for no name. The name is printed every time the mutex or condition object is printed, which is useful for debugging.

2.47. Improved packaging support (9.5.1)

The Chez Scheme `Makefile` has been enhanced with new targets for creating binary packages for Unix-like operating systems. The `create-tarball` target generates a binary tarball package for distribution, the `create-rpm` target generates a Linux RPM package, and the `create-pkg` target generates a macOS package file.

2.48. Library search handler (9.5.1)

The new `library-search-handler` parameter controls how library source or object code is located when `import`, `compile-whole-program`, or `compile-whole-library` are used to load a library. The value of the `library-search-handler` parameter must be a procedure expecting four arguments: the *who* argument is a symbol that provides context in `import-notify` messages, the *library* argument is the name of the desired library, the *directories* is a list of source and object directory pairs in the form returned by `library-directories`, and the *extensions* parameter is a list of source and object extension pairs in the form returned by `library-extensions`. The default value of the `library-search-handler` is the newly exposed `default-library-search-handler` procedure.

2.49. Ftype guardians (9.5.1)

Applications that manage memory outside the Scheme heap can leverage new support for ftype guardians to help perform reference counting. An ftype guardian is like an ordinary guardian except that it does not

necessarily save from collection each ftype pointer registered with it but instead decrements (atomically) a reference count at the head of the object to which the ftype pointer points. If the reference count becomes zero as a result of the decrement, it preserves the object so that it can be retrieved from the guardian and freed; otherwise it allows it to be collected.

2.50. Recompile information and whole-program optimization (9.5.1)

`compile-whole-program` and `compile-whole-library` now propagate recompile information from the named `wpo` file to the object file to support `maybe-compile-program` and `maybe-compile-library` in the case where the new object file overwrites the original object file.

2.51. Directly accessing the value of compile-time values (9.5.1)

The value of a compile-time value created by `make-compile-time-value` can be retrieved via the new procedure `compile-time-value-value`. The new predicate `compile-time-value?` can be used to determine if an object is a compile-time value.

2.52. Extracting a subset of hashtable entries (9.5.1)

The new `hashtable-cells` function is similar to `hashtable-entries`, but it returns a vector of cells instead of two vectors. An optional argument to `hashtable-keys`, `hashtable-values`, `hashtable-entries`, or `hashtable-cells` limits the size of the result vector.

2.53. Profile data retained for reclaimed code (9.5.1)

Profile data is now retained indefinitely even for code objects that have been reclaimed by the garbage collector. Previously, the counters holding the data were reclaimed by the collector along with the code objects. This makes profile output more complete and accurate, but it does represent a potential space leak in programs that create or load and release code dynamically. Such programs can avoid the potential space leak by releasing the counters explicitly via the new procedure `profile-release-counters`.

2.54. Procedure source location without inspector information (9.5.1)

When `generate-inspector-information` is set to `#f` and `generate-procedure-source-information` is set to `#t`, source location information is preserved for a procedure, even though other inspector information is not preserved.

2.55. Atomic compare-and-set (9.5.1)

The new procedures `box-cas!` and `vector-cas!` atomically update a box or vector with a given new value when the current content is `eq?` to a given old value. Atomicity is guaranteed even if multiple threads attempt to update the same box or vector.

2.56. Foreign-procedure thread activation (9.5.1)

A new `__collect_safe` foreign-procedure convention, which can be combined with other conventions, causes a foreign-procedure call to deactivate the current thread during the call so that other threads can perform a garbage collection. Similarly, the `__collect_safe` convention modifier for callables causes the current thread to be activated on entry to the callable, and the activation state is reverted on exit from the callable; this activation makes callables work from threads that are otherwise unknown to the Scheme system.

2.57. Garbage collection and threads (9.5.1)

A new `collect-rendezvous` function performs a garbage collection in the same way as when the system determines that a collection should occur. For many purposes, `collect-rendezvous` is a variant of `collect` that works when multiple threads are active. More precisely, the `collect-rendezvous` function invokes the collect-request handler (in an unspecified thread) after synchronizing all active threads and temporarily deactivating all but the one used to call the collect-request handler.

2.58. Foreign-procedure struct arguments and results (9.5.1)

A new (`&` *ftype*) form allows a struct or union to be passed between Scheme and a foreign procedure. The Scheme-side representation of a (`&` *ftype*) argument is the same as a (`*` *ftype*) argument, but where (`&` *ftype*) passes an address between the Scheme and C worlds, (`&` *ftype*) passes a copy of the data at the address. When (`&` *ftype*) is used as a result type, an extra (`*` *ftype*) argument must be provided to receive the copied result, and the directly returned result is unspecified.

2.59. Record equality and hashing (9.5, 9.5.1)

Several new procedures and parameters allow a program to control what `equal?` and `equal-hash` do when applied to structures containing record instances. The procedures `record-type-equal-procedure` and `record-type-hash-procedure` can be used to customize the handling of records of specific types by `equal?` and `hash`, and the procedures `record-equal-procedure` and `record-hash-procedure` can be used to look up the applicable (possibly inherited) equality and hashing procedures for specific record instances. The parameters `default-record-equal-procedure` and `default-record-hash-procedure` can be used to control the default behavior when comparing or hashing records without type-specific equality and hashing procedures.

2.60. Immutable vectors, fxvectors, bytevectors, strings, and boxes (9.5)

Support for immutable vectors, fxvectors, bytevectors, strings, and boxes has been added. Immutable vectors are created via `vector->immutable-vector`, and immutable fxvectors, bytevectors, and strings are created by similarly named procedures. Immutable boxes are created via `box-immutable`. Any attempt to modify an immutable object causes an exception to be raised.

2.61. Ephemeron pairs and hashtables (9.5)

Support for ephemeron pairs has been added, along with eq and eqv hashtables that use ephemeron pairs to combine keys and values. An ephemeron pair avoids the "key in value" problem of weak pairs, where a weakly held key is paired to a value that refers back to the key, in which case the key remains reachable as long as the pair is reachable. In an ephemeron pair, the cdr of the pair is not considered reachable by the garbage collector until both the pair and the car of the pair have been found reachable. An ephemeron hashtable implements a weak mapping where referencing a key in a value does not prevent the mapping from being removed from the table.

2.62. Optional timeout for `condition-wait` (9.5)

The `condition-wait` procedure now takes an optional *timeout* argument and returns a boolean indicating whether the thread was awakened by the condition before the timeout. The *timeout* can be a time record of type `time-duration` or `time-utc`, or it can be `#f` for no timeout (the default).

2.63. `date-dst?` and `date-zone-name` (9.5)

The new primitive procedures `date-dst?` and `date-zone-name` access time-zone information for a `date` record that is created without an explicit zone offset. The zone-offset argument to `make-date` is now optional.

2.64. `procedure-arity-mask` (9.5)

The new primitive procedure `procedure-arity-mask` takes a procedure p and returns a two's complement bitmask representing the argument counts accepted by p. For example, the arity mask for a two-argument procedure such as *cons* is 4 (only bit two set), while the arity mask for a procedure that accepts one or more arguments, such as *list**, is -2 (all but bit 0 set).

2.65. Bytevector compression (9.5)

The new primitive procedures `bytevector-compress` and `bytevector-decompress` exposes for bytevectors the kind of compression functionality that is used for files with the `compressed` option.

2.66. Line caching and source objects (9.5)

The `locate-source` function accepts an optional argument that enables the use of a cache for line information, so that a source file does not have to be consulted each time to compute line information. To further avoid file and caching issues, a source object has optional beginning-line and beginning-column components. Source objects with line and column components take more space, but they allow reporting of line and column information even if a source file is later modified or becomes unavailable. The value of the `current-make-source-object` parameter is used by the reader to construct source objects for programs, and the parameter can be modified to collect line and column information eagerly. The value of the `current-locate-source-object-source` parameter is used for error reporting, instead of calling `locate-source` or `locate-source-object-source` directly, so that just-in-time source-location lookup can be adjusted, too.

2.67. High-precision clock time in Windows 8 and up (9.5)

When running on Windows 8 and up, Chez Scheme uses the high-precision clock time function for the current date and time.

2.68. Printing of non-standard (extended) identifiers (9.5)

Chez Scheme extends the syntax of identifiers as described in the introduction to the Chez Scheme User's Guide, except within forms prefixed by `#!r6rs`, which is implied in a library or top-level program. Prior to Version 9.5, the printer always printed such identifiers using hex scalar value escapes as necessary to render them with valid R6RS identifier syntax. When the new parameter `print-extended-identifiers` is set to `#t`, these identifiers are printed without escapes, e.g., `1+` prints as `1+` rather than as `\x31;+`. The default value of this parameter is `#f`.

2.69. Expression-editor Unicode support (9.5)

The expression editor now supports Unicode characters under Linux and MacOS X except that combining characters are not treated correctly for line-wrapping.

2.70. 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.

2.71. 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).

2.72. 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.

2.73. Support for off-line 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.

2.74. 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.

2.75. sstats changes (9.3.1)

The vector-based sstats structure has been replaced with a record type. The time fields are all time objects, and the bytes and count fields are now exact integers. `time-difference` no longer coerces negative results to zero.

2.76. `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*.

2.77. 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*.

2.78. 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.

2.79. "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.

2.80. 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.

2.81. 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.

2.82. 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.

2.83. Printed representation of procedures (9.2.1)

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

2.84. 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.

2.85. 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).

2.86. 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.

2.87. 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*.

2.88. 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 4.19).

2.89. `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.

2.90. `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.

2.91. 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.

2.92. 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*.

2.93. 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`.

2.94. 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.

2.95. 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.

2.96. 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.)

2.97. 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*.

2.98. 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.

2.99. 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.

2.100. 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.

2.101. 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`.

2.102. 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)`.

2.103. 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.

2.104. 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.

2.105. 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.

2.106. `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.

2.107. 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.

2.108. `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.

2.109. 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.

2.110. 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.

2.111. `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.

2.112. `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.

2.113. `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.

2.114. 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`.

2.115. New `ssize_t` foreign type (9.0)

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

2.116. 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.

2.117. 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.

2.118. 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.

2.119. 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.

2.120. `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 4.26.

2.121. 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.

2.122. 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.

2.123. 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.

2.124. 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.

2.125. 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-difference` 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 `time-duration`. `add-duration` adds time object *t2*, which must be of type `time-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 `time-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.

2.126. 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.

2.127. 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.

2.128. 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.

2.129. 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.

2.130. 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.)

2.131. 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.

2.132. 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.

2.133. 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>
```

2.134. 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.

2.135. 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 processor 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.

2.136. 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.

2.137. 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. Incorrect code for `record?` at optimize-level 3 (9.9.9)

At optimize-level 3, the `record?` predicate could short circuit without evaluating the *rtd* expression.

3.2. Incorrect result from `Sinteger64` on 32-bit platforms (9.6.4)

On 32-bit platforms, calling `Sinteger64` or `Sunsigned64` with `0x8000000000000000` could return the wrong value.

3.3. `Sinteger32` and `Sinteger64` return unexpected bignum (9.6.4)

When called on a C value equal to `most-negative-fixnum`, `Sinteger32` and `Sinteger64` could return a bignum where a fixnum is expected. The values have the same printed representation, yet comparing the resulting bignum with `most-negative-fixnum` via Scheme's = returned false.

3.4. Library-reference import syntax (9.6.4)

A bug where `import` did not recognize a *library-spec* of the form (`library` *library-reference*) has been fixed.

3.5. Garbage collector incorrectly handles emphemerons (9.6.0)

A bug where the garbage collector sometimes incorrectly handles epehemeron pairs has been fixed.

3.6. Garbage collector incorrectly handles mutated weak pairs (9.6.0)

A bug where the garbage collector sometimes incorrectly handles mutated weak pairs has been fixed.

3.7. Division by an infinite complex number sometimes incorrectly returns +nan.0 (9.6.0)

A bug that caused /, `cfl/`, `atan`, and `atanh` to return an incorrect real or imaginary +nan.0 component has been fixed. The inexact complex number division routine now uses Robert L. Smith's 1962 algorithm.

3.8. Invalid live-pointer mask for some inline primitive calls (9.6.0)

Fixed a crash at optimize-level 2 and higher caused by the compiler generating an invalid live-pointer mask when inline-expanding certain primitive calls that store raw data on the stack and whose operands contain calls.

3.9. Optimization bug in `remove`, `member` and `assoc` (9.6.0)

When `remove`, `member` or `assoc` were used in optimize-level 3 in a position where the result was discarded, a bug in the source code optimizer could drop the call even though the first argument may be a record with a custom equality predicate that has side effects.

3.10. Foreign-callable floating-point argument allocation for x86 (9.6.0)

When a foreign callable receives a `double` or `float` argument, the allocation of space to box the number would try to save the floating-point return register in the (rare) case that allocation requires a new page of memory. Saving the return register is harmless on most platforms, but on x86, a save and restore involves popping then pushing the x87 register stack, which is invalid if nothing was there at the start.

3.11. Code generation for a specific branch displacement on ppc32 (9.6.0)

Branch generation would go wrong if the displacement was exactly 32,764 bytes.

3.12. `char-` returns negative results (9.6.0)

The character difference operator returned a large positive integer in situations where the first argument is represented by a lower number than the second argument. For example: `(char- #\a #\b)` on 64-bit macOS returns `720575940379279351`. The fix corrects this, so that it instead returns `-1`.

3.13. Certain mixed exact/inexact arithmetic comparisons (9.5.8)

The arithmetic comparison functions (`<`, `<=`, `=`, `>=`, and `>`) are required to be transitive by the R6RS specification, but this property was not maintained for `<=`, `=`, and `>=` comparisons between exact and inexact numbers in the range where fixnum precision is greater than flonum precision. For example, the flonum representation of `9007199254740992.0` and `9007199254740993.0` is identical, but obviously `9007199254740992` and `9007199254740993` (which are fixnums on 64 bit systems) are not. The arithmetic comparators now no longer convert comparisons of a fixnum and flonum to comparisons of two flonums when the fixnum cannot be converted without loss of precision.

3.14. `rational-valued?` and exceptional flonums (9.5.8)

The `rational-value?` function returned incorrect results when called on a value with an inexact zero imaginary part and real part that is an exceptional floating point value (i.e., an infinity or NaN). For example, `(rational-valued? +inf.0+0.0i)` incorrectly returned `#t`, but now returns `#f`.

3.15. Calls to foreign-callable procedures may cause the process to terminate with error 0xC0000409 STATUS_STACK_BUFFER_OVERRUN on 64-bit Windows (9.5.8)

A interaction bug between Microsoft's longjmp on 64-bit Windows and foreign-callable stack frames has been fixed.

3.16. Calls to `printf` may cause an invalid memory reference at compile time (9.5.8)

A bug in the compiler that causes an invalid memory reference with particular `printf` control strings and argument counts has been fixed. One example is (`printf "~a~:*"`).

3.17. Certain foreign calls with signed 8- and 16-bit integers on x86_64 (9.5.6)

The x86_64 code generator now properly sign-extends foreign-call arguments passed in registers via (`& integer-8`) and (`& integer-16`).

3.18. Bitwise right shift of negative bignum (9.5.6)

When a negative bignum is shifted right by a multiple of the big-digit bit size (32), a shifted-off bit is non-zero, and the result would be a sequence of big digits with all one bits before rounding to deal with the dropped bits, then a carry that should have been delivered to a new high digit was dropped, producing 0 instead of a negative number.

For example, (`ash (- 1 (ash 1 64)) -32`) no longer returns 0.

3.19. `sleep` with negative duration (9.5.6)

Prior to this release, `sleep` of a negative duration would result in an infinite pause in Windows. Now `sleep` returns immediately on all platforms when given a negative duration.

3.20. Flonum `remainder` and `modulo` (9.5.6)

The `remainder` and `modulo` functions could produce imprecise or wrong answers for large integer flonums. Most of the repair was to use the C library's `fmod`.

3.21. Buffering signals (9.5.4)

Prior to this release, only one unhandled signal was buffered for any signal for which a handler has been registered via `register-signal-handler`, so two signals delivered in quick succession could be seen as only one. The system now buffers a much larger number (63 in this release) of signals, and the fact that signals can be dropped has now been documented.

3.22. Clear-output bug (9.5.4)

A bug has been fixed in which a call to `clear-output-port` on a port could lead to unexpected behavior involving the port, including loss of buffering or suppression of future output to the port.

3.23. Various argument type-error issues (9.5.4)

A variety of primitive argument type-checking issues have been fixed, including missing checks, misleading error messages, and checks made later than appropriate, i.e., after the primitive has already had side effects.

3.24. `__collect_safe`, x86 64, and floating-point arguments or results (9.5.4)

The `__collect_safe` mode for a foreign call or callable now correctly preserves floating-point registers used for arguments or results while activating or deactivating a thread on x86 64.

3.25. `putenv` memory leak (9.5.4)

`putenv` now calls the host system's `setenv` instead of `putenv` on non-Windows hosts and avoids allocating memory that is never freed, although `setenv` might do so.

3.26. String ports from immutable strings (9.5.4)

A bug that miscalculated the buffer size for `open-string-input-port` given an immutable string has been fixed.

3.27. Multiplying -2^{30} with itself on 64-bit platforms (9.5.4)

A bug that produced the wrong sign when multiplying -2^{30} with itself on 64-bit platforms has been fixed.

3.28. Compiler dropping affects from record-accessor calls (9.5.4)

A bug that could cause the source optimizer to drop effects within the argument of a record-accessor call has been fixed.

3.29. Welcome text in macOS package file (9.5.2)

The welcome text and copyright year in the macOS package file was corrected.

3.30. Fasl representation change for recursive ftypes (9.5.2)

A bug in the reading of mutually recursive ftype definitions from compiled files has been fixed. The bug was triggered by recursive ftype definitions in which one of the mutually recursive ftypes is a subtype of another, as in:

```
(define-ftype
  [A (* B)]
  [B (struct [h A])]))
```

It manifested in the fasl reader raising bogus "incompatible record type" exceptions when two or more references to one of the ftypes occur in in separate compiled files or in separate top-level forms of a file compiled via `compile-file`. The bug could also have affected other record-type descriptors with cycles involving parent rtds and "extra" fields as well as fasl output created via `fasl-write`.

3.31. Unbound object resulting from libraries combined with `compile-whole-library` (9.5.1)

A bug in `compile-whole-library` that allowed the invoke code for a library included in the combined library body to be executed without first invoking its binary library dependencies has been fixed. This bug could arise when a member of a combined library was invoked without invoking the requirements of the other libraries it was combined with. For instance, consider the case where libraries `(A)` and `(B)` are combined

and (B) has dependencies on library (A) and binary library (C). One possible sort order of this graph is (C), (A), (B), where the invoke code for (A) and (B) are combined into a single block of invoke code. If library (A) is invoked first, it will implicitly cause the invoke code for (B) to be invoked without invoking the code for (C). We address this by adding explicit dependencies between (A) and all the binary libraries that precede it and all of the other libraries clustered with (A) and (A), such that no matter which library clustered with (A) is invoked first, (A) will be invoked, causing all binary libraries that precede (A) to be invoked. It is also possible for a similar problem to exist between clusters, where invoking a later cluster may invoke an earlier cluster without invoking the binary dependencies for the earlier cluster. We address this issue by adding an invoke requirement between each cluster and the first library in the cluster that precedes it. These extended invoke requirements are also added to the import requirements for each library, and the dependency graph is enhanced with import requirement links to ensure these are taken into account during the topological sort.

3.32. Automatic recompilation and missing include files (9.5.1)

A bug in automatic recompilation involving missing include files has been fixed. The bug caused automatic recompilation to fail, often with an exception in `file-modification-time`, when a file specified by an absolute pathname or pathname starting with "./" or "../" was included via `include` during a previous compilation run and is no longer present.

3.33. Invalid memory reference instantiating `foreign-callable` code object (9.5.1)

A bug that caused evaluation of a `foreign-callable` expression in code that has been collected into the static generation (e.g., when the `foreign-callable` form appears in code compiled to a boot file) to result in an invalid memory reference has been fixed.

3.34. Invalid constant-folding of some calls to `apply` (9.5.1)

A bug in the source optimizer (cp0) allowed constant-folding of some calls to `apply` where the last argument is not known to be a list. For example, cp0 incorrectly reduced `(apply zero? 0)` to `#t` and reduced `(lambda (x) (apply box? x) x)` to `(lambda (x) x)`, but now preserves these calls to `apply` so that they may raise an exception.

3.35. Disk-relative filenames in Windows (9.5.1)

In Windows, filenames that start with a disk designator but no directory separator are now treated as relative paths. For example, `(path-absolute? "C:")` now returns `#f`, and `(directory-list "C:")` now lists the files in the current directory on disk C instead of the files in the root directory of disk C.

In addition, `file-access-time`, `file-change-time`, `file-directory?`, `file-exists?`, `file-modification-time`, and `get-mode` no longer remove trailing directory separators on Windows.

3.36. Globally unique names on non-Windows systems no longer contain the IP address (9.5.1)

The globally unique names of gensyms no longer contain the IP address on non-Windows systems. Windows systems already used a universally unique identifier.

3.37. Invalid memory reference from `fxvector` calls (9.5)

A compiler bug that could result in an invalid memory reference or some other unpleasant behavior for calls to `fxvector` in which the nested subexpression to compute the new value to be stored is nontrivial has been fixed. This bug could also affect calls to `vector-set-fixnum!` and possibly other primitive operations.

3.38. Incorrect return code when `exit` is called with multiple arguments (9.5)

A bug in the implementation of the default exit handler with multiple values has been fixed.

3.39. Boot files containing compiled library code fail to load (9.5)

Compiled library code may now appear within fasl objects loaded during the boot process, provided that they are appended to the end of the base boot file or appear within a later boot file.

3.40. Misleading cyclic dependency error (9.5)

The library system no longer reports a cyclic dependency error during the second and subsequent attempts to visit or invoke a library after the first attempt fails for some reason other than an actual cyclic dependency. The fix also allows a library to be visited or invoked successfully on the second or subsequent attempt if the visit or invoke failed for a transient reason, such as a missing or incorrect version in an imported library.

3.41. Incomplete handling of import specs within standalone export forms (9.5)

A bug that limited the (`import` *import-spec* ...) form within a standalone `export` form to (`import` *import-spec*) has been fixed.

3.42. Permission denied after deleting files or directories in Windows (9.5)

In Windows, deleting a file or directory briefly leaves the file or directory in a state where a subsequent create operation fails with permission denied. This race condition is now mitigated. [This bug applies to all versions up to 9.5 on Windows 7 and later.]

3.43. Incorrect handling of offset in `date->time-utc` on Windows (9.5)

A bug when `date->time-utc` is called on Windows with a date-zone-offset smaller than the system's time-zone offset has been fixed. [This bug dated back to Version 9.5.]

3.44. Compiler mishandling of fx /carry operations (9.5)

A bug in the source optimizer that caused an internal compiler error when folding certain calls to `fx+/carry`, `fx-/carry`, and `fx*/carry` has been fixed. [This bug dated back to Version 9.1.]

3.45. Compiler mishandling of nested `call-with-values` calls (9.5)

A bug in that caused an internal compiler error when optimizing certain nested calls to `call-with-values` has been fixed. [This bug dated back to Version 8.9.1.]

3.46. Incorrect expansion of `define-values` of no values (9.5)

A bug in the expansion of `define-values` that caused it to produce a non-definition form when used to define no values has been fixed. [This bug dated back to at least Version 8.4.]

3.47. Optimizer dropping `pariah` forms (9.5)

A bug in the source optimizer that caused pariah forms to be ignored has been fixed. [This bug dated back to at least Version 9.3.1.]

3.48. Invalid memory references involving complex numbers (9.5)

A bug on 64-bit platforms that occasionally caused invalid memory references when operating on inexact complex numbers or the imaginary parts of inexact complex numbers has been fixed. [This bug dated back to Version 8.9.1.]

3.49. Overflow detection for left-shift operations on fixnums (9.5)

A bug that caused `fxsll`, `fxarithmetic-shift-left`, and `fxarithmetic-shift` to fail to detect overflow in certain cases has been fixed. [This bug dated back to Version 4.0.]

3.50. Missing `enum-set-indexer` argument check (9.5)

A missing argument check that resulted in the procedure returned by `enum-set-indexer` causing an invalid memory reference when passed a non-symbol argument has been fixed. [This bug dated back to Version 7.5.]

3.51. Storage for inaccessible mutexes and conditions is reclaimed (9.5)

The C heap storage for inaccessible mutexes and conditions is now reclaimed. [This bug dated back to Version 6.5.]

3.52. Missing guardian entries when a thread exits (9.5)

A bug that caused guardian entries for a thread to be lost when a thread exits has been fixed. [This bug dated back to Version 6.5.]

3.53. Incorrect code for certain nested `if` patterns (9.5)

A bug in the source optimizer that produced incorrect code for certain nested `if` patterns has been fixed. For example, the code generated for the following expression:

```
(if (if (if (if (zero? (a)) #f #t) (begin (b) #t) #f)
        (c)
        #f)
    (x)
    (y))
```

inappropriately evaluated the subexpression `(b)` when the subexpression `(a)` evaluates to 0 and not when `(a)` evaluates to 1. [This bug dated back to Version 9.0.]

3.54. Leaked or unexpected `cpvalid-defer` form (9.5)

A bug in the pass of the compiler that inserts valid checks for `letrec` and `letrec*` bindings has been fixed. The bug resulted in an internal compiler exception with a condition message regarding a leaked or unexpected `cpvalid-defer` form. [This bug dated back to Version 6.9c.]

3.55. `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 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.]

3.56. `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.]

3.57. `wchar` and `wchar_t` record field types fail to inline in Windows (9.4)

On Windows, the source optimizer has been fixed to handle `wchar` and `wchar_t` record field types.

3.58. path-related procedures cause invalid memory reference with non-string arguments in Windows (9.4)

On Windows, the path-related procedures now raise an appropriate exception when the path argument is not a string.

3.59. Mutex acquisition bug (9.4)

A bug in the handling of mutexes has been fixed. The bug typically presented as a spurious "recursively locked" exception.

3.60. `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.]

3.61. 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.]

3.62. 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.

3.63. 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*`.

3.64. 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.]

3.65. 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.]

3.66. 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.]

3.67. 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.]

3.68. 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.]

3.69. 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. Reduced allocation and copying (9.6.0)

When given a bytevector whose length is less than `file-buffer-size`, `bytevector->string` now allocates the minimum size string buffer, internal ioffsets fxvector, and internal codec buffer. The size of each was formerly hardcoded at 1024. The `bytevector->string`, `get-bytevector-all`, `get-bytevector-n`, `get-string-all`, and `get-string-n` procedures may avoid extra allocation and copying when the result is not more than `file-buffer-size` and no intermediate reads need to be stitched together to form the result.

4.2. Special-cased basic arithmetic operations (9.5.4)

The basic arithmetic operations (addition, subtraction, multiplication, division) are now much faster when presented with certain special cases, e.g., multiplication of a large integer by 1 or -1 or addition of large integer and 0.

4.3. Faster right-shift of large integers (9.5.4)

Right shifting a large integer is now much faster in most cases where the shift count is a significant fraction of the number of bits in the large integer.

4.4. Faster object-file loading (9.5.4)

Visiting an object file (to obtain only compile-time information and code) and revisiting an object file (to obtain only run-time information and code) is now faster, because revisions to the fasl format, fasl writer, and fasl reader allow run-time code to be seeked past when visiting and compile-time code to be seeked past when revisiting. For compressed object files (the default), seeking still requires reading all of the data, but the cost of parsing the fasl format and building objects in the skipped portions is avoided, as are certain side effects, such as associating record type descriptors with their uids.

Similarly, recompile information is now placed at the front of each object file where it can be loaded separately from the remainder of an object file without even seeking past the other portions of the file. Recompile information is used by `import` (when `compile-imported-libraries` is `#t`) and by maybe-compile routines such as `maybe-compile-program` to help determine whether recompilation is necessary.

Importing a library from an object file now causes the object file to be visited rather than fully loaded. (Libraries were already just revisited when required for their run-time code, e.g., when used from a top-level program.)

Together these changes can significantly reduce compile-time and run-time overhead, particularly in applications that make use of a large number of libraries.

4.5. Faster `profile-release-counters` (9.5.4)

`profile-release-counters` is now generation-friendly, meaning it does not incur any overhead for code objects in generations that have not been collected since the last call to `profile-release-counters`. Also, it no longer allocates memory when counters are released.

4.6. Reduced cost for obtaining profile counts (9.5.4)

The cost of obtaining profile counts via `profile-dump` and other mechanisms has been reduced significantly.

4.7. Better code for `bytevector` (9.5.1)

The compiler now generates better inline code for the `bytevector` procedure. Instead of one byte memory write for each argument, it writes up to four (32-bit machines) or eight (64-bit machines) bytes at a time, which almost always results in fewer instructions and fewer writes.

4.8. `vector-for-each` and `string-for-each` improvement (9.5.1)

The last call to the procedure passed to `vector-for-each` or `string-for-each` is now reliably implemented as tail call, as was already the case for `for-each`.

4.9. Lambda commonization (9.5.1)

After running the main source optimization pass (cp0), the compiler optionally runs a *commonization* pass, which commonizes code for similar lambda expressions. The parameter `commonization-level` controls whether the commonization pass is run and, if so, how aggressive it is. The parameter's value must be a nonnegative exact integer ranging from 0 through 9. When the parameter is set to 0, the default, commonization is not run. Otherwise, higher values result in more commonization.

4.10. Improved compile times (9.5.1)

Compile times are now lower, sometimes by an order of magnitude or more, for procedures with thousands of parameters, local variables, and compiler-introduced temporaries. For such procedures, the register/frame allocator proactively spills variables with large live ranges, cutting down on the size and cost of building the conflict graph used to represent pairs of variables that are live at the same time and therefore cannot share a location.

4.11. 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.

4.12. 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 a maximum-generation collection, or any other collection where release of memory to the O/S is enabled, has been cut.

4.13. 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.

4.14. 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 instance 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.

4.15. 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 e_1] [y e_2]) (list (cons x y) 7))

reduces to:

(list (cons e_1 e_2) 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.

4.16. 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`.

4.17. 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]).

4.18. 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.

4.19. 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 2.88).

4.20. 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.

4.21. 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).

4.22. 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.

4.23. 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.

4.24. 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.

4.25. 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.

4.26. 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.

4.27. 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.)

4.28. 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 nanopasses 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 4.31). The code generated for a specific program might be faster or slower.

4.29. 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.

4.30. 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.)

4.31. 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.