Chez Scheme Version 8.2 Release Notes

February 2011

1. Overview

This document outlines the changes made to *Chez Scheme* for Version 8.2 since Version 8.0, most of which are focused on converting *Chez Scheme* into an implementation of the new Scheme standard described in the Revised[6] Report on Scheme.

Version 8.2 is available for the following platforms. The Chez Scheme machine type (returned by the `machine-type` procedure) is given in parentheses.

- Linux x86, nonthreaded (i3le) and threaded (ti3le)
- Linux x86_64, nonthreaded (a6le) and threaded (ta6le)
- MacOS X x86, nonthreaded (i3osx) and threaded (ti3osx)
- MacOS X x86_64, nonthreaded (a6osx) and threaded (ta6osx)
- Windows x86, nonthreaded (i3nt) and threaded (ti3nt)
- OpenBSD x86, nonthreaded (i3ob) and threaded (ti3ob)
- OpenBSD x86_64, nonthreaded (a6ob) and threaded (ta6ob)
- FreeBSD x86, nonthreaded (i3fb) and threaded (ti3fb)
- FreeBSD x86_64, nonthreaded (a6fb) and threaded (ta6fb)
- NetBSD x86, nonthreaded (i3nb) and threaded (ti3nb)
- NetBSD x86_64, nonthreaded (a6nb) and threaded (ta6nb)
- OpenSolaris x86, nonthreaded (i3s2) and threaded (ti3s2)
- OpenSolaris x86_64, nonthreaded (a6s2) and threaded (ta6s2)

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

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

2. Functionality Changes

2.1. Foreign datatype (ftype) support (8.2)

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

Ftypes can also be used as `foreign-procedure` and `foreign-callable` argument and return types, so there is often no need to deal directly with the addresses of foreign data.

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

2.2. Locks (8.2)

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

2.3. New export forms (8.1)

Three new export forms have been added:

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

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

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

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

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

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

The `implicit-exports` form determines whether identifiers that are not directly exported from a module or

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

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

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

2.4. Change in `import` `import-only` (8.1)

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

In Version 8.2, all of the module names referenced by any of the subforms are scoped where the `import` or `import-only` form appears. Furthermore, for `import-only`, all (and only) the imports specified by the entire set of subforms are visible where the `import-only` form appears. This simplifies the task of creating a limited local scope from multiple libraries and modules.

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

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

2.5. Improved support for `define-property` (8.1)

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

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

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

2.6. Improved `library-group` support (8.1)

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

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

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

2.8. Fewer import dependencies (8.1)

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

2.9. No longer using the SIGCHLD signal (8.1)

In previous versions of *Chez Scheme*, processes created under Unix-based operating systems by the `process` and `open-process-ports` procedures were reaped via a SIGCHLD interrupt handler. In Version 8.2, such processes are instead reaped by the garbage collector, and the system no longer sets up a SIGCHLD handler. This allows programs to set up their own handlers without interfering with the reaping of those created by `process` and `open-process-ports`. It also eliminates a potential race condition involving the `system` procedure or foreign procedures that create and wait for subprocesses after first disabling the SIGCHLD interrupt.

2.10. `top-level-syntax` generalization (8.1)

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

2.11. NetBSD x86/x86_64 support (8.1)

Support for running *Chez Scheme* with 32-bit pointers on the i386 architecture or 64-bit pointers on the x86_64 architecture under NetBSD has been added, with machine types i3nb (32-bit), a6nb (64-bit), ti3nb (32-bit threaded), and ta6nb (64-bit threaded). C code intended to be linked with these versions of the system should be compiled using the Gnu C compiler's `-m32` or `-m64` options as appropriate.

2.12. Now using msvcr100.dll (8.1)

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

3. Bug Fixes

3.1. Expression-editor paste problems (8.2)

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

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

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

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

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

3.4. Bug in `record-constructor` (8.1)

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

3.5. Bug in `string-titlecase` (8.1)

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

3.6. Bug in `exp` (8.1)

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

3.7. Missing check in `(rnrs) case` (8.1)

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

3.8. Missing check in `fxvector-set!` (8.1)

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

3.9. Engine space leak (8.1)

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

4. Performance Enhancements

4.1. Better arithmetic support (8.1)

The `fx*` procedure is now inlined even outside of optimize-level 3 on x86 and x86_64 processors, and a slight improvement has been made in the efficiency of generic arithmetic operators when passed non-fixnum arguments.

4.2. Slightly faster type checks (8.1)

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