Chez Scheme Version 6.1 Release Notes

May 1999

Overview

This document outlines the changes made to *Chez Scheme* for Version 6.1 since Version 6.0.

Version 6.1 is available for the following platforms:

- DEC Alpha AXP Digital Unix 3.X, 4.X
- HP PA-RISC HPUX 10.x
- Intel 80x86 Linux 2.0.X
- Intel 80x86 Windows 95/NT
- Motorola PowerPC AIX 4.X
- Sun Sparc Solaris 2.X
- Silicon Graphics IRIX 6.X

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 prerelease to support the change.

1. Major Functionality Changes Since Version 6.0

1.1. Compile-time checking of format strings (6.1)

Compile-time control-string and argument-count checking is now performed at optimize levels 2 and 3 for `format`, `printf`, `fprintf`, `error`, and `warning`.

1.2. Quasiquote extension (6.1)

`unquote` and `unquote-splicing` have been extended in an upwardly compatible way to allow multiple subforms in splicing (list or vector) contexts. `(unquote-splicing x ...)` now splices into the surrounding context the result of appending the values of `x ...`, and `(unquote x ...)` now inserts into the surrounding context the values of `x ...`. These changes support certain useful nested `quasiquote` idioms, such as `,@,@`, which has the effect of a doubly indirect splicing when used within a doubly nested and doubly evaluated `quasiquote` expression. For example,

```
(define x '(m n))
(define m '(b c))
(define n '(d e))
(eval ``(a ,@,@x f) (interaction-environment))
```

evaluates to `(a b c d e f)`.

1.3. Tracing syntax transformers (6.1)

Support for `trace-define-syntax` has been added. This form is like `define-syntax`, except that the transformer bound to the defined keyword is wrapped in tracing code that displays the input and output to the transformer in the same manner as `trace-lambda`. See the *Chez Scheme User's Guide* for details.

1.4. Making ports from file descriptors (6.1)

`make-output-port`, `make-input-port`, and `make-input/output-port` now allow the first, *handler*, argument to be a nonnegative fixnum representing a low-level file descriptor, obtained via some operating-system dependent means. When *handler* is a file descriptor, the newly created port is given a handler that reads from and/or writes to the file descriptor. See the *Chez Scheme User's Guide* for details.

1.5. Inspection of record-type descriptors (6.1)

The interactive and object inspectors now allow a record's record-type descriptor to be accessed and inspected. The command `rtd` is used while inspecting a record in the interactive inspector to inspect the record's record-type descriptor. The message `rtd` is used similarly in the object inspector.

1.6. Block Comments (6.0a)

The reader now allows a block of characters to be commented out within properly nested `#|`, `|#` pairs, without regard to line boundaries. For example, the expression

```
(+ #| This is a comment |# 3 4)
```

evaluates to `7`, as does

```
(+ #| This is a comment
      with another #| nested
      inside, crossing several
      line |# boundaries |# 3 4)
```

1.7. Support for Windows "stdcall" Convention (6.0a)

Chez Scheme now supports the Windows `__stdcall` procedure calling convention on Intel 80x86 systems for foreign procedures created with `foreign-procedure`. The default is still the `__cdecl` convention, as with Microsoft's C compiler. `__stdcall` is specified by placing the keyword `__stdcall` before the foreign procedure name, e.g.:

```
(foreign-procedure __stdcall "GetVolumeInformationA"
  (string string unsigned-32 string string string string unsigned-32)
  boolean)
```

`__stdcall` is useful for calling Windows API functions, except for those with "varargs" interfaces, which use the default `__cdecl` convention.

2. Bugs Fixed Since Version 6.0

2.1. Module names introduced by macros (6.1)

The expander now properly renames module names introduced into the expansion of a macro at top-level, as it already did for other defined identifiers.

2.2. Interpreter `case-lambda` bug (6.1)

A bug in the interpreter's treatment of `case-lambda`, which caused argument-count errors to be produced for certain correct applications of procedures created by `case-lambda`, has been fixed.

2.3. IRIX memory limitation (6.1)

A limitation on the maximum size of a heap under IRIX caused by inappropriate linker assumptions about the operating environment has been lifted. In theory, the size of a heap is now limited only by available (32-bit) virtual memory, and in practice we have been able to test heaps of up to about two gigabytes in size.

2.4. Storage-Management and Records (6.0a)

A bug in the garbage collector's handling of records containing both mutable pointer (`scheme-object`) and nonpointer (`integer-32`, `unsigned-32`, or `double-float`) fields has been fixed. The bug sometimes caused unrecoverable invalid memory references.

2.5. Source-Tracking Problems (6.0a)

A couple of problems with source tracking in the compiler have been fixed. These problems sometimes resulted in memory faults within the inspector or in misleading information being printed as part of syntax error message. An unrelated bug with finding source files with pathnames that start with a drive letter under Windows has also been fixed.

2.6. Overeager Argument-Count Checking (6.0a)

The compiler now signals a warning rather than an error when it detects a call to a locally bound procedure with an incorrect number of arguments. This prevents the compiler from rejecting programs such as the following, which might otherwise execute without errors:

```
(let ((f (lambda (x) x)))
    (if (read)
        (f 3)
        (f 3 4))))
```

It also prevents the compiler from rejecting programs with less obvious "errors," uncovered during procedure inlining, such as the following:

```
(let ((f (lambda (g n)
           (if (= n 1)
               (g (lambda (x) x))
               (g (lambda (x y) x))))))
  (f (lambda (h) (h 3)) 1))
```

3. Performance Enhancements Since Version 6.0

3.1. Vector handling in `quasiquote` (6.1)

Handling of certain vectors in `quasiquote` has been improved. For example, `` `#(a 1 ,(+ 3 4) x y) `` now expands into the following code.

```
(#2%vector 'a 1 (+ 3 4) 'x 'y)
```

Previously, the same expression expanded into code that generated a list and converted this list into a vector via `list->vector`.