R7RS Considered Unifier of Previous Standards *

William D Clinger
Northeastern University
will@ccs.neu.edu

Abstract

The R7RS (small) language standard can be implemented while preserving near-perfect backward compatibility with the R6RS standard and substantial compatibility with the R5RS and IEEE/ANSI standards for the Scheme programming language. When this is done, as in Larceny, R6RS Scheme becomes a proper subset of R7RS Scheme.

1. Introduction

Portability and interoperability are two different things, and often come into conflict with each other. When programming languages are specified, portability can be increased (and interoperability diminished) by inventing some specific semantics for all corner cases while forbidding extensions to standard syntax and procedures. Interoperability can be increased (and portability diminished) by leaving unimportant corner cases unspecified while allowing implementations to generalize syntax and semantics in ways that simplify interoperation with other standards and components.

Consider, for example, the `port-position` procedure of R6RS Scheme [23]. Its first draft specification, based on a Posix feature that assumes every character is represented by a single byte, would have precluded efficient interoperation with UTF-8, UTF-16, and Windows [20]. Subsequent drafts weakened that specification, increasing interoperability at the expense of allowing port positions to behave somewhat differently on different machines and in different implementations of the standard.

The R6RS tended to resolve those conflicts in favor of portability, however, while the R7RS tended to favor interoperability [17, 21, 23]. It is therefore fairly easy for an R7RS-conforming implementation to preserve backward compatibility with R6RS and SRFI libraries and to provide convenient access to libraries and components written in other languages, but it is considerably more difficult for R6RS-conforming implementations to avoid creating barriers to interoperation with R7RS and SRFI libraries. That asymmetry between the R7RS and R6RS standards often goes unremarked and its importance under-appreciated when incompatibilities between those two standards are discussed.

Throughout this paper, R7RS (without parenthetical disambiguation) means the R7RS (small) language standard, which was endorsed in 2013 after its ninth draft had been approved by 88.9% of the votes cast [17]. Six of the seven votes cast against the draft included comments expressing concern about incompatibilities with the previous R6RS or R5RS standards [12, 21]. These comments accounted for 17 of the 61 comments that required a response from Working Group 1 before the R7RS language standard was endorsed [5]. Another 7 of the 61 comments expressed similar concerns but accompanied votes cast in favor of the draft.

Although the R7RS language standard does not mandate compatibility with previous standards, it turns out that the R7RS can be implemented while preserving near-perfect backward compatibility with the R6RS standard and substantial compatibility with the R5RS and IEEE/ANSI standards. When this is done, as in Larceny v0.98, R6RS Scheme becomes a proper subset of R7RS Scheme.

A year ago, the implementor of Sagittarius explained how he implemented the R7RS language on top of an R6RS system, allowing R7RS/R6RS libraries and programs to interoperate [11].

This paper builds upon that Sagittarius experience by describing design decisions and compromises that improve interoperability between R7RS/R6RS libraries and programs. This paper also describes several open-source components that may be of interest to other implementors of R7RS/R6RS systems, including a portable implementation of Unicode 7.0 and other libraries along with test suites and benchmarks used here to appraise current support for the R7RS and R6RS standards.

2. Larceny

Larceny v0.98, released in March 2015 [1], supports the R7RS, R6RS, R5RS, and IEEE/ANSI standards for Scheme.

Throughout this paper, "Larceny" refers to two implementations of Scheme that share source libraries, runtime system, and compiler front end, but their different approaches to generating machine code justify classifying them as separate implementations.

Native Larceny JIT-compiles all Scheme code to native machine code for ARMv7 and IA32 (x86, x86-64) proces-

sors running Linux, OS X, or Windows. For faster loading, files can also be compiled to machine code ahead of time.

Petit Larceny is a highly portable implementation that uses an interpreter for read/eval/print loops but can compile files to C code. Scheme code compiled by Petit Larceny runs about half as fast as in native Larceny.

Larceny was the second implementation of the R6RS, and was first to implement almost all of the R6RS standard. Larceny was not so quick to implement the R7RS; at least ten other systems were already supporting the R7RS (in whole or in part) when Larceny v0.98 was released.

If `pgm` is an R7RS or R6RS program, then

```
larceny --r7r6 --program pgm
```

will run the program. Omitting `--program pgm` will enter an R7RS-compatible read/eval/print loop in which all libraries described by the R7RS and R6RS standards have been pre-imported. Larceny's `--r7rs` option is equivalent to the `--r7r6` option when a program is specified, but imports only the `(scheme base)` library when the `--program` option is omitted.

Larceny's `--r6rs` option provides a legacy mode whose primary purpose is to test whether R6RS programs limit themselves to R6RS syntax and semantics. This `--r6rs` mode enforces several "absolute requirements" of the R6RS that prohibit extensions to R6RS syntax and procedures and forbid interactive read/eval/print loops. As explained in the next section, these prohibitions interfere with interoperability.

3. More Honored in the Breach

Citing RFC 2119 [10], R6RS Chapter 2 says it uses the words "must" and "must not" when stating an "absolute requirement" or "absolute prohibition" of the specification.

The R6RS contains many such absolute requirements. R6RS "mustard" absolutely forbids most extensions to lexical syntax, library syntax, and semantics of procedures exported by standard libraries.

The R6RS also contains absolute requirements that have the effect of forbidding interactive read/eval/print loops. According to chapter 8 of the R6RS rationale, the R6RS editors adopted those requirements because they wanted to leave interactive environments "completely in the hands of the implementors" rather than run the risk of restricting "Scheme implementations in undesirable ways" [18]. Their rationale tells us the editors themselves believed the R6RS mustard forbidding interactive read/eval/print loops would be "more honored in the breach than the observance" [16].

That created precedent for honoring other absolute requirements in the breach. Most implementations of the R6RS default to a deliberately non-conformant mode that offers at least some forbidden features such as a read/eval/print loop or lexical extensions favored by the implementation. Users who wish to run their programs in a less permissive mode must disable extensions prohibited by the R6RS by manipulating various flags and switches.

Some members of the Scheme community still find it hard to believe the R6RS absolutely forbids many extensions often regarded as desirable. It is therefore necessary to examine a few examples in detail.

3.1 Example: lexical syntax

R6RS Chapter 4 says

> An implementation must not extend the lexical or datum syntax in any way, with one exception: it need not treat the syntax `#!<identifier>`, for any `<identifier>` (see section 4.2.4) that is not `r6rs`, as a syntax violation, and it may use specific `#!`-prefixed identifiers as flags indicating that subsequent input contains extensions to the standard lexical or datum syntax.

That absolute requirement would allow R6RS programs to read data produced by R7RS programs when the data are preceded by a flag such as `#!r7rs`, but it forbids extension of the R6RS `read` procedure to accept unflagged R7RS syntax for symbols, strings, characters, bytevectors, and circular data structures.

Kent Dybvig suggested the `#!r6rs` flag in May 2006. When I formally proposed addition of Dybvig's suggestion, I anticipated a future R7RS lexical syntax in which the `#!r6rs` flag would mark data and source code that still used R6RS syntax [2]:

> I propose we add `#!r6rs` as a new external representation that every R6RS-conforming implementation must support. Its purpose is to flag code that is written in the lexical syntax of R6RS, to ease the eventual transition from R6RS to R7RS lexical syntax.

Less than six weeks later, in the R6RS editors' status report, the `#!r6rs` flag had come to mean R6RS lexical syntax must be rigidly enforced, with all lexical extensions forbidden unless preceded by a `#!` flag other than `#!r6rs` [7]:

- The new syntax `#!r6rs` is treated as a declaration that a source library or script contains only r6rs-compatible lexical constructs. It is otherwise treated as a comment by the reader.
- An implementation may or may not signal an error when it sees `#!`*symbol*, for any symbol *symbol* that is not `r6rs`. Implementations are encouraged to use specific `#!`-prefixed symbols as flags that subsequent input contains extensions to the standard lexical syntax.
- All other lexical errors must be signaled, effectively ruling out any implementation-dependent extensions unless identified by a `#!`-prefixed symbol.

That is the semantics demanded by the R6RS standard ratified in 2007.

Consider, for example, the R7RS datum label syntax that allows reading and writing of circular data. This lexical syntax is not described by the R6RS standard, so the standard `read` and `get-datum` procedures provided by implementations of the R6RS can support the syntax only as an implementation-dependent extension that's absolutely forbidden by R6RS mustard unless it is preceded by an implementation-specific `#!`-prefixed flag.

- Racket does not as yet offer an R7RS mode, and its R6RS mode does not appear to allow the R7RS datum label syntax under any circumstances.
- Sagittarius, Larceny, and Petit Larceny allow the R7RS datum label syntax in R7RS modes but do not allow it in R6RS mode unless it is preceded by an implementation-specific flag such as `#!r7rs`.
- Petite Chez Scheme allows R7RS datum label syntax by default but enforces strict R6RS lexical syntax when data or code follow an `#!r6rs` flag.
- Vicare allows the R7RS datum label syntax even in data or expressions that follow an `#!r6rs` flag.

As explained in section 9.1, those are the leading implementations of the R6RS currently available for Linux machines. Four of them (Racket, Sagittarius, Larceny, and Petit Larceny) enforce R6RS mustard with respect to this particular lexical extension. The other two (Petite Chez and Vicare) honor that absolute requirement in the breach.

3.2 Example: argument checking

R6RS Section 5.4 says implementations *must* check restrictions on the number of arguments passed to procedures specified by the standard, *must* check other restrictions "to the extent that it is reasonable, possible, and necessary" to do so, and *must* raise an exception with condition type `&assertion` whenever it detects a violation of an argument restriction. These and other absolute requirements forbid extension of R6RS procedures such as `map`, `member`, and `string->utf8` to accomodate the more general semantics of those procedures as specified by the R7RS.

3.3 Example: syntax violations

R6RS Section 5.5 says implementations *must* detect syntax violations, and *must* respond to syntax violations by raising a `&syntax` exception before execution of the top-level program is allowed to begin. These are the absolute requirements that forbid interactive read/eval/print loops. They also forbid extension of the `define-record-type` syntax to accept R7RS, SRFI 9, or SRFI 99 syntax, and forbid extension of the `syntax-rules` and `case` syntaxes to accept new features added by the R7RS.

3.4 Possible responses

The examples offered above show how R6RS mustard interferes with interoperability between R6RS and R7RS code.

One possible response to these absolute requirements is to regard the R6RS as a dead end, worthy of support only in legacy modes.

Another possible response is to take R6RS absolute requirements seriously, even when they interfere with interoperability. Programs that import R7RS and R6RS libraries would have to rename all syntaxes and procedures whose R6RS and R7RS specifications differ in even the smallest of ways, just as R6RS programs have had to rename the `map`, `for-each`, `member`, `assoc`, and `fold-right` procedures when importing `(rnrs base)`, `(rnrs lists)`, and `(srfi :1 lists)`.

For Larceny we chose a third way, regarding many of the R6RS's absolute requirements as quaint customs that would be more honored in the breach. When interoperability between R7RS/R6RS/R5RS code would be improved by ignoring an R6RS requirement, Larceny ignores the requirement.

With many technical standards, implementations that ignore any of the standard's absolute requirements would be severely crippled or unusable. With the R6RS, however, implementations that ignore the standard's absolute requirements become more usable than implementations that take those requirements seriously.

4. Compromises and Workarounds

As explained by Larceny's user manual [13]:

> Larceny is R6RS-compatible but not R6RS-conforming. When Larceny is said to support a feature of the R6RS, that means the feature is present and will behave as specified by the R6RS so long as no exception is raised or expected. Larceny does not always raise the specific conditions specified by the R6RS, and does not perform all of the checking for portability problems that is mandated by the R6RS. These deviations do not affect the execution of production code, and do not compromise Larceny's traditional safety.

This distinction between R6RS-compatible and R6RS-conforming foreshadowed compromises that would be needed for convenient interoperation between R7RS, R6RS, and R5RS libraries and programs.

Most incompatibilities between the R7RS and R6RS can be made to disappear by adopting the more general semantics specified by the R7RS while ignoring absolute requirements of the R6RS that forbid such extensions.

The R7RS explicitly allows extensions to its lexical syntax and procedures, so implementations of the R7RS are free to extend the `read` procedure to accept R6RS lexical syntax as well as R7RS syntax. Larceny's implementation of `read` is described in a separate section below.

Although R6RS `define-record-type` has little in common with `define-record-type` as specified by the R7RS,

SRFI 9, and SRFI 99, that syntactic incongruity made it easy for Larceny's `define-record-type` to accept code written according to any of those standards.

The R6RS `error` procedure treats its first argument as a description of the procedure reporting the error, and allows that argument to be a string, a symbol, or `#f`; there must also be a second argument, which must be a string. The R7RS and SRFI 23 standards specify an `error` procedure that requires its first argument to be a string, and treats it as an error message. In Larceny, a single `error` procedure implements both the R7RS and R6RS semantics by using the execution mode and its arguments to guess whether it should behave as specified by the R6RS or as specified by the R7RS and SRFI 23. The `error` procedure enforces R6RS semantics under either of these circumstances:

- Larceny is running in `--r6rs` mode
- its first argument is not a string, and its second argument is a string

Larceny's default exception handler reports errors in a laconic format that should make sense even when the `error` procedure guesses wrong.

One incompatibility between the R6RS and R7RS standards could not be resolved by generalizing a syntax or procedure. Their specifications of `bytevector-copy!` are dangerously incompatible because they disagree concerning whether the first and third arguments are destination or source of the copy. The `(larceny r7r6)` library that's imported by Larceny's `--r7r6` option renames the R6RS procedure to `r6rs:bytevector-copy!`. Libraries and programs that import `(rnrs bytevectors)` directly get the original spelling, of course, and must rename something themselves if they also import `(scheme base)`.

The R7RS specification of `real?` says "`(real? z)` is true if and only if `(zero? (imag-part z))` is true" but gives an example saying `(real? -2.5+0.0i)` evaluates to false. I believe the prose specification should have said this:

> In implementations that do not provide the optional `(scheme complex)` library, `(real? z)` is always true. In implementations that do provide the library, `(real? z)` is false if `(zero? (imag-part z))` is false, true if both `(zero? (imag-part z))` and `(exact? (imag-part z))` are true, and may be true whenever `(zero? (imag-part z))` is true.

Without that repair, the R7RS prose is consistent with R5RS but not with R6RS, while the R7RS examples are consistent with R6RS but not with R5RS.

The R6RS semantics of `real?` was a carefully reasoned improvement over the R5RS semantics, and experience with the R6RS has shown that programmers doing numerical work appreciate the improvement, while casual programmers seldom notice it. Larceny is consistent with the R7RS examples, with the R6RS, and with the correction I suggested above. A survey of other implementations, detailed in the appendix, supports that correction.

5. Library Syntax

Larceny implemented the R6RS using Andre van Tonder's implementation of R6RS libraries and macros [27]. For Larceny v0.98, we upgraded that component to process R7RS libraries and programs as well as R6RS libraries and programs. It now expands `define-library` and `library` syntax into a common intermediate form, so there is no way for client code to tell which syntax was used to define libraries it imports. Hence R7RS and R6RS libraries and programs are fully interoperable.

Any incompatibilities between `define-library` and `library` must therefore be rooted in their own syntax and semantics. Their syntaxes are obviously disjoint, so there is no direct conflict between R7RS and R6RS library syntax.

On the other hand, the R7RS `define-library` syntax allows unsigned integers to appear within library names such as `(srfi 1)` and `(srfi 1 lists)`. R6RS `library` syntax does not allow those names.

Larceny enforces the R6RS prohibition of unsigned integers within the names of libraries *defined* by R6RS `library` syntax, but ignores the R6RS absolute requirement that forbids *importation* of libraries with such names into an R6RS `library` or program. This partial flouting of R6RS absolute requirements may seem arbitrary, but it

- improves portability (by discouraging creation of R6RS source libraries whose names would be rejected by other implementations of the R6RS) and also
- improves interoperability (by allowing unrestricted importation of R7RS and SRFI libraries that may not even exist in other implementations of the R6RS).

SRFI 97 specifies a convention in which the numeric part of a SRFI library name is preceded by a colon, as in `(srfi :1 lists)` [9]. The R7RS standard rendered that SRFI 97 convention obsolete outside of R6RS libraries and programs.

Larceny now uses the R7RS convention, as in `(srfi 1)` and `(srfi 1 lists)`, to name the SRFI libraries it supports. For backward compatibility, Larceny continues to supply duplicate libraries that use the SRFI 97 naming convention, as in `(srfi :1)` and `(srfi :1 lists)`. For the newer SRFI libraries (numbered above 101), Larceny supports only the R7RS naming convention. That decision can be reconsidered if enough programmers tell us they are still using the R6RS `library` syntax when writing new code.

The R7RS `define-library` syntax offers several advantages over the R6RS `library` syntax. R7RS `include` and `cond-expand` facilities have already shown their worth, and liberalized placement of `import` declarations works well with `cond-expand` and read/eval/print loops.

The R6RS `library` syntax supports optional versioning, but that feature never really caught on, partly because the R6RS did not even suggest a file naming convention that could accomodate its hierarchical versions. R6RS Section 7.1 implies the mapping from library names to library code is implementation-dependent, and this implication becomes more emphatic in R6RS Non-normative Appendixes E and G [22]. That sanctions implementations such as Larceny in which an R6RS library's version is ignored.

The `library` syntax's most apparent advantage over `define-library` is explicit phasing of procedural macros. The R6RS community now appears to favor implicit phasing, which is allowed by the R6RS, so this advantage may not be real [8]. Larceny v0.98 requires explicit phasing, but that is likely to change in a future release.

6. Lexical Syntax

In most modes, Larceny normally recognizes R7RS lexical syntax together with most of the lexical syntax specified by the older R6RS, R5RS, and IEEE/ANSI standards. In `--r6rs` mode, which tries to enforce most absolute requirements of the R6RS, Larceny normally recognizes only R6RS lexical syntax.

The lexical syntax allowed on a textual input port can be altered by reading a `#!r7rs`, `#!r6rs`, `#!r5rs`, `#!larceny`, `#!fold-case`, or `#!no-fold-case` flag from the port. The `#!fold-case` and `#!no-fold-case` flags behave as specified by the R7RS. The other flags affect a set of port-specific flag bits that determine whether the port allows R7RS, R6RS, and Larceny weirdness (which is Larceny-specific jargon for extensions to R5RS/IEEE/ANSI lexical syntax). As required by the R6RS, the `#!r6rs` flag enables R6RS weirdness while disabling R7RS and Larceny weirdness. The `#!r7rs` flag enables R7RS weirdness without disabling other weirdness, and also enables case-sensitivity. The `#!larceny` flag enables R7RS, R6RS, and Larceny weirdness without disabling other weirdness; it too enables case-sensitivity. The R5RS allows extensions to its lexical syntax, so Larceny's `#!r5rs` flag is equivalent to this sequence of flags:

```
#!r7rs #!larceny #!fold-case
```

The lexical syntax allowed on newly opened textual ports is determined by a set of parameters that have been given names such as `read-r7rs-weirdness?` even though they affect output ports as well as input ports.

Bytevectors are written using R7RS syntax unless the output port disallows R7RS weirdness and allows R6RS weirdness, in which case R6RS syntax is used. Larceny's `read` procedure accepts both R7RS and R6RS bytevector syntax unless the input port disallows both R7RS and Larceny weirdness, in which case only R6RS bytevector syntax is accepted.

Symbols, strings, and characters are written using R7RS syntax unless the output port disallows R7RS weirdness, in which case R6RS syntax is used unless the port also disallows R6RS weirdness, in which case characters that would not be portable in context under R5RS rules are written using inline hex escapes.

The R6RS does not allow its `write` and `display` procedures to produce a finite representation of cyclic data structures that can be read reliably by the R6RS `read` procedure, but does allow those procedures to go into an infinite loop when asked to print cyclic data. The R7RS requires `write` to use datum labels when printing cyclic data, as in `#1=(0 . #1#)`, but forbids datum labels when there are no cycles. Larceny's `write` and `display` procedures therefore produce datum labels only when their R6RS behavior is essentially unspecified, which is a rare example of interoperability made possible by underspecification in the R6RS instead of the R7RS.

Larceny's `read` procedure is implemented by a machine-generated finite state machine and strong LL(1) parser that accept the union of R7RS, R6RS, R5RS, and Larceny-specific syntax. Action routines called by the state machine and parser perform all of the checking necessary to determine whether the syntax is allowed by the input port. The complexity of these checks makes it impractical for Larceny to allow easy customization of its `read` procedure.

7. Unicode

Larceny uses the R6RS reference implementation of Unicode written by Mike Sperber and myself, upgraded to Unicode 7.0 [19, 25]. A trivial conversion of this reference implementation to use R7RS library syntax and R7RS libraries has made the R6RS `(rnrs unicode)` library available to any implementation of the R7RS that can represent Unicode characters and strings [4, 23].

The R6RS requires implementations to support Unicode characters and strings, but the R7RS standard made that optional.

I tested ten implementations of the R7RS: the eight listed in section 9.1, plus Picrin and Husk Scheme. Of those ten, Picrin is the only one that cannot represent arbitrary Unicode characters. Chicken can represent all Unicode characters but defaults to strings limited to the Latin-1 subset of Unicode; I am told that Chicken can also support full Unicode strings. The other implementations support Unicode strings as well as characters.

The R7RS `(scheme char)` library is almost a subset of the R6RS `(rnrs unicode)` library, adding only `digit-value` while omitting `char-general-category`, three procedures that implement title case, and four procedures that convert strings to Unicode normalization forms NFC, NFD, NFKC, or NFKD. The R6RS and R7RS specifications of `char-numeric?` look slightly different, but that's a minor mistake in the R6RS that was corrected in R7RS.

The R6RS requires `string-downcase` to handle Greek sigma as specified by Unicode Standard Annex #29 [24]. This implies detection of word boundaries to decide whether to use final or non-final sigma. Even so, the Unicode specification does not handle all Greek text correctly, because there are situations that cannot be distinguished without knowing what the text means. The R7RS explicitly allows `string-downcase` to convert every upper-case sigma to a non-final sigma. Of the ten R7RS implementations tested, Gauche, Kawa, native Larceny, Petit Larceny, and Sagittarius appear to handle Greek sigma as specified by the R6RS and Unicode 7.0.

R6RS Section 11.12 says implementors *should* make `string-ref` run in constant time, and it does in all six implementations of the R6RS I tested. The R7RS standard says "There is no requirement for this procedure to execute in constant time." Of the eight R7RS systems tested that normally support Unicode strings, only Foment, Larceny, Petit Larceny, and Sagittarius define a `string-ref` that runs in constant time.

Although the Scheme standards have done an excellent job of specifying a string data type that can accomodate Unicode without assuming any particular representation or character set beyond ASCII, mutable strings of fixed length are now a local pessimum in the design space. Scheme Working Group 2 is therefore considering the addition of a new data type for immutable sequences of Unicode characters [6]. This new data type would provide efficient sequential access in both directions, efficient extraction of substrings, efficient searching, and space efficiency approaching that of UTF-8. What's more, this new data type could be implemented so random accesses run in O(1) time.

8. Assessment

Interoperability between R7RS and R6RS code is illustrated by Larceny's use of R6RS standard libraries to implement most of the R7RS libraries, and by the mix of R7RS/R6RS libraries Larceny uses to implement more than 50 SRFI libraries.

Interoperability is also demonstrated by using Larceny's `--r7rs` and `--r6rs` modes to run conformance tests, benchmarks, and tests of SRFI libraries.

8.1 Racket's R6RS tests

Racket's implementation of the R6RS includes a test suite that runs 8897 tests of conformance to the R6RS standard. Petite Chez Scheme appears to be the only free implementation of the R6RS for Linux that passes all of those tests. Racket v6.1.1 fails three tests; two of those failures involve Unicode title case, and are caused by not implementing Unicode Standard Annex #29 [15]. Sagittarius version 0.6.4 fails three tests, including one in which it detects a violation of the `letrec` restriction at compile time instead of run time and then refuses to run the program. (R6RS Section 11.4.6 says implementations *must* detect `letrec` violations during evaluation of the expression, which implies run time.) Vicare v0.3d7 fails six tests, including two in which it detects violations of the `letrec` restriction at compile time and refuses to run the program.

In `--r6rs` mode, native Larceny and Petit Larceny both fail one test by allowing it to run to completion despite a violation of the `letrec` restriction that goes undetected because the variables involved in the violation are not used. In `--r7rs` mode, native Larceny and Petit Larceny both fail a second test when `(log 0)` throws an exception whose R7RS-conforming condition object doesn't belong to the specific condition class demanded by the R6RS.

So Larceny is reasonably compatible with the R6RS even when operating in `--r7rs` mode.

8.2 Larceny's R7RS tests

Using Racket's R6RS tests as a starting point, I implemented a test suite that (as of this writing) runs 2156 tests of conformance to the R7RS standard. That number is a bit misleading, because many of those tests would have been split into several distinct tests had they been written in the Racket style. Comparing lines of code, our R7RS test suite is slightly larger than Racket's R6RS test suite.

In `--r7rs` mode, native Larceny and Petit Larceny both fail one test because they have not yet implemented the generalized ellipsis form of `syntax-rules`.

When the R7RS tests are run in Larceny's `--r6rs` mode, that mode's strict enforcement of R6RS syntax rejects four sections of the R7RS test suite that use R7RS syntax for bytevectors or strings. When an `#!r7rs` flag is added at the beginning of those four files, Larceny passes 2117 of the tests while failing 39:

- 1 test failed as it did in R7RS mode.
- 1 test failed because the `error` procedure used R6RS semantics.
- 11 tests of `(scheme write)` failed because R6RS syntax was written.
- 13 tests of `(scheme read)` failed because R7RS data were read from a string that did not contain an `#!r7rs` flag.
- 10 tests of `(scheme repl)` failed.
- 3 tests of `(scheme load)` failed.

These failures show how strict enforcement of R6RS mustard interferes with interoperability. Smooth interoperation between R7RS and R6RS code is achieved only by Larceny's more liberal R7RS modes.

8.3 Benchmarks

We have collected 68 R6RS benchmarks and translated 57 of them into R7RS benchmarks [1]. The untranslated bench-

marks test features such as hashtables or sorting routines that have no counterpart in R7RS.

In `--r6rs` mode, native Larceny runs all of the R6RS benchmarks successfully.

In `--r7rs` mode, Larceny should be able to run all R6RS benchmarks but does not. In `--r7rs` mode, Larceny returns an incorrect result for the R6RS `read0` benchmark because it accepts R7RS-legal symbols that begin with the `@` character even after an `#!r6rs` flag has been read from the input port. This is a bug in the `read` procedure's enforcement of R6RS syntax, discovered only during preparation of this paper; this bug will be fixed in Larceny v0.99, which should be released by the end of August 2015.

In `--r7rs` mode, native Larceny runs all of the R7RS benchmarks successfully.

Even in `--r6rs` mode, native Larceny runs all of the R7RS benchmarks successfully. They were, after all, translated from R6RS code without making any special effort to introduce R7RS-specific syntax or features.

8.4 SRFI tests

The source distribution of Larceny contains 49 R7RS programs that test SRFI libraries whose names follow the R7RS convention and another 45 R6RS programs that test SRFI libraries whose names follow the SRFI 97 (R6RS) convention.

All of the R6RS test programs can be run in either `--r7rs` or `--r6rs` mode.

All but one of the R7RS test programs can be run in either `--r7rs` or `--r6rs` mode. The R7RS test program for `(srfi 115 regexp)` contains an R7RS symbol syntax that's rejected by the `--r6rs` mode's strict enforcement of R6RS syntax.

9. Portability

Portability of R7RS or R6RS code is determined as much or more by the available implementations as by the standards themselves.

9.1 Implementations

In May 2015, I was able to benchmark eight free implementations of the R7RS on the Linux machine we use for benchmarking:

- Chibi Scheme 0.7.3
- Chicken Scheme Version 4.9.0.1
- Foment Scheme 0.4 (debug)
- Gauche version 0.9.4
- Kawa 2.0
- Larceny v0.98
- Petit Larceny v0.98
- Sagittarius 0.6.4

I was able to install three more implementations that claim to implement at least part of R7RS, but was unable to get them to run enough of the R7RS benchmarks to make benchmarking worthwhile.

I was able to benchmark six free implementations of the R6RS on that same Linux machine:

- Larceny v0.98
- Petit Larceny v0.98
- Petite Chez Version 8.4
- Racket v6.1.1
- Sagittarius 0.6.4
- Vicare Scheme version 0.3d7, 64-bit

I tried to install several other implementations of the R6RS without success; most had not been updated for several years. Vicare is a fork of Ikarus, which I did not try to install because it is no longer being maintained.

As should be expected, R6RS and R6RS/R7RS systems tend to be more mature than R7RS-only systems. Half of the six R6RS systems were able to run all of Larceny's R6RS benchmarks, and the other half failed on only one benchmark. Chibi, native Larceny, and Sagittarius were the only R7RS systems able to run all of Larceny's R7RS benchmarks, with two others (Chicken and Petit Larceny) able to run all but one benchmark [3].[1]

The R6RS systems also tended to run faster. Taking the geometric mean over all benchmarks, Petit Larceny was the third slowest implementation of the R6RS but the second fastest implementation of R7RS.

I am, however, impressed by the promise of the R7RS implementations.

9.2 File naming conventions

The R7RS and R6RS standards do not specify any mapping from library names to files or other locations at which the code for a library might be found. R6RS non-normative appendix E emphasizes the arbitrariness of such mappings. R7RS Section 5.1 meekly suggests

> Implementations which store libraries in files should document the mapping from the name of a library to its location in the file system.

Fortunately, *de facto* standards have been emerging.

An R6RS library named `(rnrs io simple (6))` is typically found within a file named `rnrs/io/simple.sls`. (The version is typically ignored. On Windows systems, backslashes would be used instead of forward slashes.) An R7RS library named `(srfi 113 sets)` is typically found within a file named `srfi/113/sets.sld`. That file may consist of a `define-library` form that specifies the exports and imports but includes its definitions from another file. If

[1] The next release of Kawa is expected to run all of the R7RS benchmarks.

```
(define-library (baz)
  (export x y)
  (import (scheme base))
  (begin
   (define x 10)
   (define y (+ x x))))
```

Figure 1. An R7RS library in a file named `baz.sld`.

```
(import (scheme base)
        (scheme write)
        (scheme process-context)
        (baz))
(write (list x y))
(newline)
(exit)
```

Figure 2. An R7RS program in a file named `pgm`.

```
(import (scheme base)
        (scheme load)
        (scheme write)
        (scheme process-context))
(load "baz.sld")
(import (baz))
(write (list x y))
(newline)
(exit)
```

Figure 3. A similar R7RS program in a file named `pgm2`.

so, the included file is typically named `sets.body.scm` and placed within the same directory as the `sets.sld` file.

For the `(include "sets.body.scm")` convention to work, implementations must search for the included file within the directory of the including file. Chicken, Gauche, Kawa, Larceny, and Petit Larceny do so, and the development version of Foment 0.5 is said to do so as well.

9.3 Auto-loading conventions

The R7RS standard does not say whether library files must be loaded explicitly before the libraries they contain can be imported. This underspecification is impeding the portability of R7RS programs.

Some implementations of the R7RS apparently require library files to be loaded (using the `load` procedure of `(scheme load)`) before the libraries they contain can be imported.

Other implementations of the R7RS load library files automatically when the libraries they contain are imported, using file naming conventions and a search path to locate those libraries. All tested implementations of the R6RS use this approach as well.

Consider, for example, the `baz` library of Figure 1 and the R7RS program shown in Figure 2. If the `baz.sld` and `pgm` files are located within the current working directory of a Linux machine, then seven of the ten implementations I tested will run the program using command lines shown in the appendix.

If the closely related program of Figure 3 is contained within a file named `pgm2` in that same directory, then it too can be run by seven of the ten implementations. (For details, see the appendix.)

Of the implementations tested, Foment, native Larceny, Petit Larceny, and Sagittarius appear to be the only ones that can run both versions of the program without changing the file names or source code. The portability of R7RS programs will be enhanced if implementors follow their example.

9.4 Lightweight libraries improve modularity

If `baz.sld` and `pgm` are concatenated into a single file, then Chicken, Foment, Gauche, Kawa, native Larceny, Petit Larceny, and Sagittarius will run the program. This appears to be the most portable way to distribute a complete R7RS program that defines its own libraries.

That's a substantial shift from R6RS practice. Native Larceny and Petit Larceny seem to be the only implementations that allow an R6RS program's libraries to be defined within the same file that contains the top-level program itself.

The R6RS editors appear to have thought of R6RS libraries as a mechanism for distributing code that would probably have to be translated into implementation-specific module systems and go through a fairly heavyweight installation process, as with Racket collections, before they could be imported into a program [14].

As can be seen in reference implementations of recent SRFIs, the R7RS community thinks of R7RS libraries as a lightweight and portable tool for constructing more modular programs. I believe that's progress.

9.5 R6RS standard libraries

It's all very well to say the R6RS is a proper subset of R7RS as implemented by Larceny, but how easy would it be to make that happen in other implementations of the R7RS?

Most of the standard R6RS libraries have been ported to R7RS and can be downloaded from `snow-fort.org`:

```
(r6rs base)
(r6rs unicode)
(r6rs bytevectors)
(r6rs lists)
(r6rs sorting)
(r6rs control)
(r6rs exceptions)
(r6rs files)
(r6rs programs)
(r6rs arithmetic fixnums)
```

```
(r6rs hashtables)
(r6rs enums)
(r6rs eval)
(r6rs mutable-pairs)
(r6rs mutable-strings)
(r6rs r5rs)
```

These correspond to the `(rnrs *)` libraries of R6RS, but have been renamed to avoid conflict with the original R6RS libraries as provided by R6RS/R7RS implementations such as Sagittarius and Larceny. These libraries use `cond-expand` to import the corresponding `(rnrs *)` library if it is available, which guarantees full interoperability with any of the standard R6RS libraries that may be provided by implementations of the R7RS.

If the corresponding `(rnrs *)` library is not available, `cond-expand` will `include` portable code that implements the library on top of R7RS standard libraries. The portable implementation of `(r6rs base)` implements `identifier-syntax` as a stub that generates a syntax error when used. The portable implementation of hashtables relies on `(rnrs hashtables)` if that library is available, or builds upon `(srfi 69)` if that library is available, or builds upon a portable implementation of `(srfi 69)` if nothing better is available. The `(r6rs *)` libraries listed above are otherwise equivalent to their `(rnrs *)` counterparts.

Implementation of the `(r6rs arithmetic flonums)` and `(r6rs arithmetic bitwise)` libraries should be straightforward. The `(r6rs io simple)` library is more interesting because it should support both R6RS and R7RS lexical syntax.

The following components of the R6RS are hard to implement portably atop R7RS without sacrificing interoperability with corresponding components of the R7RS implementation:

- R6RS lexical syntax
- R6RS `library` syntax
- the R6RS record system
- the `(rnrs conditions)` library
- parts of the `(rnrs io ports)` library
- the `(rnrs syntax-case)` library

R6RS libraries and programs may contain non-R7RS syntax for bytevectors, identifiers, strings, and even a few characters. Translation from R6RS to R7RS lexical syntax is trivial, but the need for translation will interfere with interoperability in implementations that reject non-R7RS syntax.

Some implementations of the R7RS may hard-wire their `(scheme read)` and `(scheme write)` libraries so tightly they can't be replaced, which will force code that also imports `(rnrs io simple)` or `(r6rs io simple)` to rename one of the two versions of the `read` and `write` procedures.

Translation from `library` to `define-library` syntax is trivial, so the R6RS `library` syntax is the easiest of the listed components for R7RS systems to support natively. Without built-in support, the need for a separate translation step degrades interoperability. The R7RS standard does not allow `define-library` forms as the output of macro expansion. In seven of the ten implementations tested, `library` can be defined as a hygienic macro that expands into code that uses `eval` to evaluate the corresponding `define-library` form in the `interaction-environment`; three of those seven already support `library` natively.

The R7RS (large) standard will probably include a record system similar to SRFI 99, which can implement the procedural and inspection layers of R6RS records with full interoperability between those layers and the R7RS, SRFI 9, and SRFI 99 record systems. R7RS (large) is also likely to include a macro system capable of implementing the R6RS syntactic layer on top of SRFI 99.

Implementing the `(rnrs conditions)` library on top of SRFI 99 records is straightforward, but integrating its condition objects into an R7RS system's native exception system cannot be done portably.

The `(rnrs io ports)` library includes several individual features that sounded good in isolation but do not combine well. Unsurprisingly, those are the features that cannot be implemented portably on top of the R7RS i/o system:

- port positions
- custom ports
- bidirectional input/output ports

If these problematic features are dropped, then the rest of the R6RS i/o system can be approximated more or less crudely in R7RS. To emphasize the crudity of the approximation, consider the R6RS `transcoded-port` procedure. In R7RS systems that don't distinguish between binary and textual ports, this procedure can just return its first argument whenever its second argument is the native transcoder. Output ports are hardly ever passed to `transcoded-port`, so an implementation restriction that rejects any attempt to add non-native transcoding to a binary output port will seldom cause trouble. If conversions from interactive binary input ports to textual are also limited to native transcoding, then non-native transcoders will be allowed only when the first argument corresponds to a bytevector or file, so the `transcoded-port` procedure can copy all remaining bytes from the binary port into a bytevector or temporary file, which it can then open as a textual port using the specified transcoder.

The `(rnrs syntax-case)` library might be approximated using an `eval` trick as described above for R6RS `library` syntax, but that would be unpleasant even if it works. It's more practical to wait for R7RS (large), which is expected to include a macro system with enough power to approximate `syntax-case`. Larceny, for example, imple-

ments (rnrs syntax-case) on top of an explicit renaming macro system, as outlined by SRFI 72 [26, 27].

10. Conclusion

Implementations of the R7RS can achieve near-perfect backward compatibility with the R6RS.

R7RS programmers can derive some benefit from R6RS libraries even in systems that don't support the R6RS standard. Most R6RS standard libraries have been implemented on top of R7RS [4]. Some of the R6RS standard libraries that can't be implemented in R7RS (small) are likely to become implementable in the anticipated R7RS (large) standard.

R7RS (large) is also expected to include standard libraries that go well beyond those provided by the R6RS.

The usefulness, portability, and interoperability of R7RS code are more likely to be limited by the availability and quality of implementations, and by practical issues such as file naming and auto-loading conventions, than by incompatibilities between the R7RS and R6RS standards.

A. Appendix

The program in Figure 2 can be run in Chibi Scheme, Foment, Husk Scheme, native Larceny, Petit Larceny, and Sagittarius by incanting

```
chibi-scheme -I . < pgm
foment pgm
huski pgm
larceny --r7rs --path . --program pgm
sagittarius -r7 -L . pgm
```

That program can be run in Gauche by copying baz.sld to a file named baz and incanting

```
gosh -r7 -I . -l pgm
```

The program in Figure 3 can be run in Chicken, Foment, Gauche, Kawa, and native Larceny or Petit Larceny by incanting

```
csi -require-extension r7rs pgm2
foment pgm2
gosh -r7 -I . pgm2
kawa --r7rs -f pgm2
larceny --r7rs < pgm2
sagittarius -r7 pgm2
```

The --path and -L options of Larceny and Sagittarius can be omitted here because pgm2 loads the library file explicitly. For some reason, Gauche must be given the analogous -I option even with pgm2.

The incantation shown for Chicken uses the csi interpreter because that fits on a single line. When benchmarking, I ran Chicken's compiler (csc) with five command-line options to enable various optimizations; running the compiled program then becomes a separate step.

As noted at the end of Section 4, the R7RS prose specification of real? refers to the imag-part procedure, which is available only in implementations that provide the optional (scheme complex) library. One of the ten tested implementations of the R7RS does not support that library, but all of the nine mentioned in this appendix do provide it. Six of the nine—including Chibi Scheme, which was written by the chair of Working Group 1 and served as a reference implementation for the R7RS standard—agree with the R7RS by saying (real? -2.5+0.0i) evaluates to false. Of the three implementations that disagree with this R7RS example, one (Husk Scheme) violates R7RS semantics by refusing to compute (imag-part 2.5), so it also violates the R7RS prose specification of the real? procedure. Two implementations behave as specified by the R7RS prose. All but one of the ten implementations behave as specified by my suggested repair of that prose, as would the outlier (Husk Scheme) if its imag-part bug were fixed.

Acknowledgments

I am gratified by the assistance given me by implementors of the R6RS and R7RS systems named here. We aren't all working on the same implementation, but we are certainly working to implement the same or similar language(s), and have much to offer one another.

I am also grateful to the editors of the R6RS and R7RS documents, who made enormous progress while creating standards that allow backward compatibility and interoperability.

John Cowan, an editor of the R7RS standard and chair of Working Group 2, improved this paper by commenting upon its first two drafts. He is of course not responsible for my opinions and outright mistakes, nor is he responsible for my speculations concerning the R7RS (large) standard being developed by Working Group 2.

I believe the program committee's suggestions helped to improve this paper. I do not know whether the program committee shares my belief.

References

[1] W. D. Clinger. Larceny home page. URL www.larcenists.org.

[2] W. D. Clinger. r6rs-editors email archives, May 2006. URL http://www.r6rs.org/r6rs-editors/2006-May/001251.html.

[3] W. D. Clinger. Larceny benchmarks, Mar 2015. URL http:www.larcenists.org/benchmarks2015.html.

[4] W. D. Clinger and T. U. Bayirli/Kammer. R6RS standard libraries for R7RS systems, 2015. URL snow-fort.org/pkg.

[5] J. Cowan. PlebisciteObjections. URL http://trac.sacrideo.us/wg/wiki/PlebisciteObjections.

[6] J. Cowan. Character span library, 2015. URL http://trac.sacrideo.us/wg/wiki/CharacterSpansCowan.

[7] K. Dybvig, W. Clinger, M. Flatt, M. Sperber, and A. van Straaten. R6RS status report, 2006. URL www.schemers.org/Documents/Standards/Charter/status-jun-2006/status-jun06.html.

[8] A. Ghuloum. The portable R6RS library and syntax-case system, 2008. URL https://launchpad.net/r6rs-libraries/.

[9] D. V. Horn. SRFI libraries, 2008. URL http://srfi.schemers.org/srfi-97/srfi-97.html.

[10] Internet Engineering Task Force. IETF RFC 2119: Key words for use in RFCs to indicate requirement levels, Mar 1999. URL http://www.ietf.org/rfc/rfc2119.txt.

[11] T. Kato. Implementing R7RS on an R6RS Scheme system. In *Scheme and Functional Programming Workshop*, Nov 2014. URL http://www.schemeworkshop.org/2014/.

[12] R. Kelsey, W. Clinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *Journal of Higher Order and Symbolic Computation*, 11(1):7–105, 1998. URL http://www.scheme-reports.org/.

[13] Larcenists. Larceny user manual. URL http://www.larcenists.org/doc.html.

[14] Racketeers. Installing libraries. URL http://docs.racket-lang.org/r6rs/Installing_Libraries.html.

[15] Racketeers. R6RS conformance. URL http://docs.racket-lang.org/r6rs/conformance.html.

[16] W. Shakespeare. *Hamlet*. 1602. Act 1, scene 4.

[17] A. Shinn, J. Cowan, and A. A. Gleckler. Revised^7 Report on the Algorithmic Language Scheme. 2013. URL http://www.scheme-reports.org/.

[18] M. Sperber. Revised^6 Report on the Algorithmic Language Scheme — Rationale. 2007. URL http://www.r6rs.org/.

[19] M. Sperber and W. D. Clinger. Unicode library, 2007. URL https://github.com/larcenists/larceny/tree/master/tools/Unicode/r6rs-unicode.

[20] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised$^{5.92}$ Report on the Algorithmic Language Scheme — Standard Libraries. Jan. 2007. URL http://www.r6rs.org/history.html.

[21] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2007. URL http://www.r6rs.org/.

[22] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme — Non-Normative Appendixes. 2007. URL http://www.r6rs.org/.

[23] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. Revised^6 Report on the Algorithmic Language Scheme — Standard Libraries. 2007. URL http://www.r6rs.org/.

[24] Unicode Consortium. Unicode Standard Annex #29, 2014. URL http://www.unicode.org/reports/tr29/.

[25] Unicode Consortium. The Unicode Standard, 2014. URL http://unicode.org/.

[26] A. van Tonder. Hygienic macros, 2005. URL http://srfi.schemers.org/srfi-72/srfi-72.html.

[27] A. van Tonder. R6RS libraries and macros, 2007. URL http://www.het.brown.edu/people/andre/macros/.