Scheme Reports Working Group 1 Progress 2012

Alex Shinn
alexshinn@gmail.com

1. Introduction

After over two and a half years of work, the R^{7}RS "small" language specification is approaching completion. In the near future we will be publishing a seventh and semi-final draft, based on all of our ballots and the comments and feedback from the community. The Steering Committee will then form an electorate for ratification of the R^{7}RS "small" language, and after the ratification vote any editorial corrections will be incorporated into an eight and final draft.

This status report reviews the background and motivations of the standard, the process and tribulations the members encountered, and a summary of the resulting standard as specified in the upcoming seventh draft.

2. Background

Historically, the minimalism and simplicity of Scheme have led to it being widely used in academia, as a teaching language and tool for programming language research. Perhaps because of this image, or from a misconception that minimalism is unsuited to building large systems, Scheme never gained the commercial support that even its relatively unpopular sister Common Lisp enjoys.

The result was that while other languages enjoyed "practical" standards driven by commercial needs, or the advantage of no standard but only a single reference implementation, Scheme remained a collection of disparate communities centered around different implementations. Sharing code was almost unknown, and implementations spent time re-inventing the same utility libraries over and over.

An attempt at unification for practical programming was first made in 1998, after the completion of the R^{5}RS, with the introduction of the Scheme Request for Implementation (SRFI) process. Unfortunately, while many useful and well designed SRFIs were produced, there was still no mechanism for specifying how to "load" a SRFI portably, or to write a portable library of code. Despite all the syntactic power of Scheme, it was still impossible to write a program with libraries which would actually work on more than one implementation.

In 2003 work on a new and more ambitious standard, the R^{6}RS, was begun. Ratified in 2007, R^{6}RS not only provided a standardized module system and many new features, it attempted to greatly reduce the amount of unspecified semantics in the language. The whole report was re-organized, and changed in spirit from being more prescriptive than descriptive. Fairly or not, many implementors balked at the scope and style of the changes, and the uptake of R^{6}RS was not as widespread as could be hoped. A parallel can be made with the Common Lisp standardization process, in the following comment from Kent Pitman[1]:

> One problem was that Common Lisp was more descriptive than prescriptive. That is, if two implementation communities disagreed about how to solve a certain problem, CLTL was written in a way that sought to build a descriptive bridge between the two dialects in many cases rather than to force a choice that would bring the two into actual compatibility. This may even have been a correct strategy since it was most important in the early days just to get buy-in from the community on the general approach. The notion that it

2012 Workshop on Scheme and Functional Programming September 9, 2012, Copenhagen, Denmark.

[1] `http://www.nhplace.com/kent/Papers/cl-untold-story.html`

> mattered for two implementations to agree was at that point a mostly abstract concern. There were not a lot of programs moving from implementation to implementation yet. As the user base later grew and program porting became a more widespread practice, the community will to invest in such matters grew. But at the time when CLTL was published, a sense that the language design must focus on true portability had not yet evolved.

We are at the same point in the Scheme standardization process. At this point in time, it's far more important to get the different implementations on the same page first, and worry about finer portability issues later.

Hoping to take advantage of many of the improvements of R^{6}RS while at the same time producing a common ground that more implementations could agree on, the Scheme Steering Committee chose in 2009 to split the standard into two languages: a small language suitable for education, research and embedded systems, and a large language with all the batteries included for modern programming. The small language would focus on R^{5}RS compatibility, while the large language would be a superset of that, focusing on more features and R^{6}RS compatibility.

In this way the new standard would appeal both minimalists and pragmatists. Those who weren't ready for R^{6}RS could restrict themselves to the small language, but by doing so they would be using the same module system and core language as the large language, allowing for sharing of code and a smooth upgrade path. Those who wanted the large language would have an extended community from common core. The best features of the large language would no doubt find their way into even the most minimalist implementations. We would be one step closer to a unified Scheme community.

It is the small language which is nearing completion. As many members of the working groups formed to create the two languages overlapped, and because the large language is constrained by compatibility with the small language, work on the latter was postponed until completion of the former, and will thus be commencing soon.

3. Process

The charter effectively requires compatibility with both R^{5}RS and R^{6}RS, with the small language remaining roughly the same size as R^{5}RS. This is of course an impossible goal, and from the beginning it was clear that the small language would be for the most part a line drawn between the past two standards. This is not to suggest that there was little work to be done, nor that the job was easy. Quite the contrary, it was a constant and tiresome balancing act.

The work was divided into an initial fact-finding and issue collection phase, a series of ballots where actual changes were decided on, and a formal comments period from the community which resulted in additional ballots and changes. Because the entire process was open, with the working group member's discussion list readable by the public as well as a public discussion list open to anyone, we in fact received fairly constant feedback throughout the process. The public comments were overwhelmingly friendly and helpful, and we owe a large debt to the community for helping to make the standard what it is.

By now almost 500 tickets have been logged, resulting in 6 ballots and 7 drafts, all of which are publicly visible. Discussion among the members has been productive and collegial. As a group we have grown and learned; we've not been afraid to admit when we were wrong, change our votes, or revisit issues when there was an oversight or when new arguments have come to light.

On the other hand, we've tried hard to stay focused and progressive, and not needlessly revisit issues without good cause. Initially this was handled informally, with the chair reviewing items and deciding whether there was grounds for further discussion. More recently, as we've been nearing completion, the process was modified to require a formal nomination and second before bringing a change proposal up for vote.

4. Difficult Issues

Naturally much of the active discussion revolved around "bike-shedding" issues, syntactic extensions and names of procedures and such which are easy to conceive and for which variations have little or no interaction with deeper semantics. These were often also the first things to be commented on by the community. In particular, the names of bytevectors were initially called `blobs` to reflect the view that they were not just sequences of bytes but could be treated as sequences of integers of different sizes depending on need. They were only later changed to `bytevector` as a more descrip-

tive name which also provides R^{6}RScompatibility, and nearly changed back after comments from the community.

There were also deep issues that generated large amounts of discussion. The semantics of `eqv?` on numbers in particular was tricky. Because `eqv?` itself is a collection of different specifications on different types, it was natural to attack this one case at a time, such as what is the result of `(eqv? 0.0 -.0)` or `(eqv? +nan.0 +nan.0)`. Unfortunately this led to us not seeing the forest for the trees, with a collection of inconsistent cases without a single guiding principle of what `eqv?` "means". Once this was recognized we of course backtracked, and rephrased the question to define a high-level description of the semantics from which individual cases should fall out naturally. The result is in an operational semantics based on the IEEE 754 definition of numbers.

Some interesting comments from the community were simply beyond the scope of our standard. Oleg Kiselyov made a detailed and persuasive case against undelimited continuations as in `call/cc` in favor of delimited continuations as in `shift` and `reset`. Unfortunately, though delimited continuations are not by any means a new concept, they have relatively little support among implementations in comparison to `call/cc`. Moreover, our backwards-compatibility requirements make it impossible to remove such a fundamental part of the language. What we can and will do, however, is provide delimited continuations in addition to `call/cc` in the large language, and hope their support and use increases so this can be revisited in later standards.

5. Changes in the Language

The following is a summary of notable changes since the last two standards, as excerpted from the notes in the latest draft.

5.1 Incompatibilities with R^{5}RS

This section enumerates the incompatibilities between this report and the "Revised^5 report" [2].

- Case sensitivity is now the default in symbols and character names. This means that code written under the assumption that symbols could be written `FOO` or `Foo` in some contexts and `foo` in other contexts must either be changed, be marked with a `#!fold-case` directive, or be included in a library using the `include-ci` library declaration. All standard identifiers are entirely in lower case.
- The `syntax-rules` construct now recognizes _ (low line) as a wildcard, which means it cannot be used as a syntax variable. It can still be used as a literal, however.
- The R^{5}RS procedures `exact->inexact` and `inexact->exact` have been renamed to their R^{6}RS names, `inexact` and `exact`, respectively, as these names are shorter and more correct. The former names are still available in the R^{5}RS compatibility library.
- The guarantee that string comparison (with `string<?` and the related predicates) is a lexicographical extension of character comparison (with `char<?` and the related predicates) has been removed. The former set provide an implementation-defined comparison as in R^{5}RS; the latter set provide comparison by Unicode code point.
- Support for the # character in numeric literals is no longer required.
- The procedures `transcript-on` and `transcript-off` have been removed.

5.2 Other language changes since R^{5}RS

This section enumerates the additional differences between this report and the "Revised^5 report" [2].

- Various minor ambiguities and unclarities in R^{5}RS have been cleaned up.
- Libraries have been added as a new program structure to improve encapsulation and sharing of code. Some existing and new identifiers have been factored out into separate libraries. Libraries can be imported into other libraries or main programs, with controlled exposure and renaming of identifiers. The contents of a library can be made conditional on the features of the implementation on which it is to be used.
- Exceptions can now be signalled explicitly with `raise`, `raise-continuable` or SRFI 23[8] `error`, and can be handled with `with-exception-handler` and the `guard` syntax. Any object can specify an error condition; the implementation-defined conditions signalled by `error` have a predicate to detect them and accessor functions to retrieve the arguments passed to `error`. Conditions signalled by

`read` and by file-related procedures also have predicates to detect them.

- New disjoint types supporting access to multiple fields can be generated with SRFI 9's[7] `define-record-type`.
- SRFI 39-style[9] parameter objects can be created with `make-parameter`, and dynamically rebound with `parameterize`.
- *Bytevectors*, homogeneous vectors of integers in the range $[0, 255]$, have been added as a new disjoint type. A subset of the procedures available for vectors is provided. Bytevectors can be converted to and from strings in accordance with the UTF-8 character encoding. Bytevectors have a datum representation and evaluate to themselves.
- A `delay-force` procedure based on SRFI 45[10] `lazy` has been added, and `force` is required be properly tail-recursive when applied to it. Promises can be tested with the `promise?` predicate, and created with `make-promise`.
- Vector constants evaluate to themselves.
- The procedure `read-line` is provided to make line-oriented textual input simpler.
- *Ports* can now be designated as *textual* or *binary* ports, with new procedures for reading and writing binary data. The new predicate `port-open?` returns whether a port is open or closed.
- *String ports* have been added as a way to read and write characters to and from strings, and *bytevector ports* to read and write bytes to and from bytevectors.
- The procedures `current-input-port` and `current-output-port` are now parameter objects, as is the newly introduced `current-error-port`.
- The `syntax-rules` construct now allows the ellipsis symbol to be specified explicitly instead of the default `...`, allows template escapes with an ellipsis-prefixed list, and allows tail patterns to follow an ellipsis pattern.
- The `syntax-error` syntax has been added as a way to signal immediate and more informative errors when a macro is expanded.
- Internal `define-syntax` definitions are now allowed wherever internal `defines` are.
- The `letrec*` binding construct has been added, and internal `define` is specified in terms of it.
- Support for capturing multiple values has been enhanced with `define-values`, `let-values`, and `let*-values`. Standard forms which introduce a body now permit passing zero or more than one value to the continuations of all non-final forms of the body.
- The `case` conditional now supports a `=>` syntax analogous to `cond`.
- To support dispatching on the number of arguments passed to a procedure, `case-lambda` has been added in its own library.
- The convenience conditionals `when` and `unless` have been added.
- Positive infinity, negative infinity, NaN, and negative inexact zero have been added to the numeric tower as inexact values with the written representations `+inf.0`, `-inf.0`, `+nan.0`, and `-0.0` respectively. Support for them is not required. The representation `-nan.0` is synonymous with `+nan.0`.
- The procedures `map` and `for-each` are now required to terminate on the shortest list when the inputs have different lengths.
- The procedures `member` and `assoc` now take an optional third argument specifying the equality predicate to be used.
- The procedures `exact-integer? square`, and `exact-integer-sqrt` have been added.
- The procedures `make-list`, `list-copy`, `list-set!`, `string-map`, `string-for-each`, `string->vector`, `vector-append`, `vector-copy`, `vector-map`, `vector-for-each`, `vector->string`, `vector-copy!`, and `string-copy!` have been added to round out the sequence operations. Some of these support processing of part of a string or vector using optional start and end arguments.
- Implementations may provide any subset of the full Unicode repertoire that includes ASCII, but implementations must support any such subset in a way consistent with Unicode. Various character and string procedures have been extended accordingly. String comparison remains implementation-dependent, and is no longer required to be consistent with character comparison, which is based on Unicode code points. The new `digit-value` pro-

cedure has been added to obtain the numerical value of a numeric character.

- There are now two additional comment syntaxes: `#;` to skip the next datum, and `#| ... |#` for nestable block comments.
- Data prefixed with datum labels `#<n>=` can be referenced with `#<n>#`, allowing for reading and writing of data with shared structure.
- The aliases `#true` and `#false` have been added for `#t` and `#f`.
- Strings and symbols now allow mnemonic and numeric escape sequences, and the list of named characters has been extended.
- The procedures `file-exists?` and `delete-file` are available in the `(scheme file)` library.
- An interface to the system environment and command line is available in the `(scheme process-context)` library.
- Procedures for accessing time-related values are available in the `(scheme time)` library.
- A less irregular set of integer division operators is provided with new and clearer names.
- The `load` procedure now accepts a second argument specifying the environment to load into.
- The semantics of read-eval-print loops are now partly prescribed, requiring the redefinition of procedures, but not syntax keywords, to have retroactive effect.

5.3 Incompatibilities with the main R^{6}RS document

This section enumerates the incompatibilities between R^{7}RS and the "Revised^6 report" [1].

- The syntax of the library system was deliberately chosen to be syntactically different from R^{6}RS, using `define-library` instead of `library` in order to allow easy disambiguation between R^{6}RS and R^{7}RS libraries.
- The library system does not support phase distinctions, which are unnecessary in the absence of low-level macros (see below), nor does it support versioning, which is an important feature but deserves more experimentation before being standardized.
- Putting an extra level of indirection around the library body allows room for extensibility. The R^{6}RS syntax provides two positional forms which must be present and must have the correct keywords, `export` and `import`, which does not allow for unambiguous extensions. The Working Group considers extensibility to be important, and so chose a syntax which provides a clear separation between the library declarations and the Scheme code which makes up the body.
- The `include` library declaration makes it easier to include separate files, and the `include-ci` variant allows case-insensitive code to be incorporated.
- The `cond-expand` library declaration based on SRFI 0[3] allows for a more flexible alternative to the R^{6}RS `.impl.sls` file naming convention. The list of identifiers that `cond-expand` treats as true is available at run time using the `features` procedure.
- Since the R^{7}RS library system is straightforward, we expect that R^{6}RS implementations will be able to support the `define-library` syntax in addition to their `library` syntax.
- The grouping of standardized identifiers into libraries is different from the R^{6}RS approach. In particular, procedures which are optional either expressly or by implication in R^{5}RS have been removed from the base library. Only the base library is an absolute requirement.
- Identifier syntax is not provided. This is a useful feature in some situations, but the existence of such macros means that neither programmers nor other macros can look at an identifier in an evaluated position and know it is a reference — this in a sense makes all macros slightly weaker. Individual implementations are encouraged to continue experimenting with this and other extensions before further standardization is done.
- Internal syntax definitions are allowed, but all references to syntax must follow the definition; the `even`/`odd` example given in R^{6}RS is not allowed.
- The R^{6}RS exception system was incorporated as-is, but the condition types have been left unspecified. Specific errors that must be signalled in R^{6}RS remain errors in R^{7}RS, allowing implementations to provide their own extensions. There is no discussion of safety.
- Full Unicode support is not required. Normalization is not provided. Character comparisons are defined by Unicode, but string comparisons are implementation-dependent, and therefore need not

be the lexicographic mapping of the corresponding character comparisons (an incompatibility with R^{5}RS). Non-Unicode characters are permitted.

- The full numeric tower is optional as in R^{5}RS, but optional support for IEEE infinities, NaN, and -0.0 was adopted from R^{6}RS. Most clarifications on numeric results were also adopted, but the R^{6}RS procedures `real-valued?`, `rational-valued?`, and `integer-valued?` were not. The R^{6}RS division operators `div`, `mod`, `div-and-mod`, `div0`, `mod0` and `div0-and-mod0` are not provided.
- When a result is unspecified, it is still required to be a single value, in the interests of R^{5}RS compatibility. However, non-final expressions in a body may return any number of values.
- Because of widespread SRFI 1[4] support and extensive code that uses it, the semantics of `map` and `for-each` have been changed to use the SRFI 1 early termination behavior. Likewise `assoc` and `member` take an optional `equal?` argument as in SRFI 1, instead of the separate `assp` and `memp` procedures of R^{6}RS.
- The R^{6}RS `quasiquote` clarifications have been adopted, but the Working Group has not seen convincing enough examples of multiple-argument `unquote` and `unquote-splicing`, so they are not provided.
- The R^{6}RS method of specifying mantissa widths was not adopted.

5.4 Incompatibilities with the R^{6}RS Standard Libraries document

This section enumerates the incompatibilities between R^{7}RS and the R^{6}RS [1] Standard Libraries.

- The low-level macro system and `syntax-case` were not adopted. There are two general families of macro systems in widespread use — the `syntax-case` family and the `syntactic-closures` family — and they have neither been shown to be equivalent nor capable of implementing each other. Given this situation, low-level macros have been left to the large language.
- The new I/O system from R^{6}RS was not adopted. Historically, standardization reflects technologies that have undergone a period of adoption, experimentation, and usage before incorporation into a standard. The Working Group was unhappy with the redundant provision of both the new system and the R^{5}RS-compatible "simple I/O" system, which relegated R^{5}RS code to being a second-class citizen. However, binary I/O was added using binary ports that are at least potentially disjoint from textual ports and use their own parallel set of procedures.
- String ports are compatible with SRFI 6[6] rather than R^{6}RS; analogous bytevector ports are also provided.
- The Working Group felt that the R^{6}RS records system was overly complex, and the two layers poorly integrated. The Working Group spent a lot of time debating this, but in the end decided to simply use a generative version of SRFI 9, which has near-universal support among implementations. The Working Group hopes to provide a more powerful records system in the large language.
- R^{6}RS-style bytevectors are included, but provide only the "u8" procedures in the small language. The lexical syntax uses `#u8` for compatibility with SRFI 4[5], rather than the R^{6}RS `#vu8` style. With a library system, it's easier to change names than reader syntax.
- The utility macros `when` and `unless` are provided, but since it would be meaningless to try to use their result, it is left unspecified.
- The Working Group could not agree on a single design for hash tables and left them for the large language.
- Sorting, bitwise arithmetic, and enumerations were not considered to be sufficiently useful to include in the small language. They will probably be included in the large language.
- Pair and string mutation are too well established to be relegated to separate libraries.

Acknowledgments

Thanks to the members of the Steering Committee for their guidance, the editors John Cowan and Arthur Gleckler and the working group for all their hard work, and to the community for their invaluable feedback.

References

[1] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. The revised^6 report on the algorithmic language Scheme.

[2] Richard Kelsey, William Clinger, and Jonathan Rees, editors. The revised^5 report on the algorithmic language Scheme.

[3] Marc Feeley. SRFI-0: Feature-based conditional expansion construct. `http://srfi.schemers.org/srfi-0/`, May 1999.

[4] Olin Shivers. SRFI-1: List Library. `http://srfi.schemers.org/srfi-1/`, October 1999.

[5] Marc Feeley. SRFI-4: Homogeneous numeric vector datatypes. `http://srfi.schemers.org/srfi-4/`, May 1999.

[6] William D Clinger. SRFI-6: Basic String Ports. `http://srfi.schemers.org/srfi-6/`, July 1999.

[7] Richard Kelsey. SRFI-9: Defining Record Types. `http://srfi.schemers.org/srfi-9/`, September 1999.

[8] Stephan Houben. SRFI-23: Error reporting mechanism. `http://srfi.schemers.org/srfi-23/`, June 2001.

[9] Marc Feeley. SRFI-39: Parameter objects. `http://srfi.schemers.org/srfi-39/`, June 2003.

[10] Andre van Tonder. SRFI-45: Primitives for Expressing Iterative Lazy Algorithms. `http://srfi.schemers.org/srfi-45/`, 2002.