Guiding Requirements for the Ongoing Scheme Standardization Process

Mario Latendresse
SRI International
Information and Computing Sciences
333 Ravenswood Ave, Menlo Park, CA 94025, USA
email: latendre@iro.umontreal.ca

Abstract

The Scheme standardization process has produced several Scheme revisions, the most recent one being R^{6}RS. The R^{7}RS standardization process is underway with an amended charter. The new charter has introduced two language levels, Small Scheme and Large Scheme, succinctly described as "lightweight" and "heavyweight", respectively. We analyze this new charter and propose some modifications to it that we believe would help the standardization of Scheme, and in particular steer it towards greater use by the software developer community. We suggest that the Steering Committee establish guiding requirements for the two Scheme levels. We discuss some examples of concrete guiding requirements to include in the standardization process for maintenance and debugging. We also discuss the need for an additional general principle for Small Scheme and suggest that, besides the general principle of a small language specification, the notion of efficiency of execution is also at the core of Small Scheme.

1. Introduction

In 1975, when first introduced by Sussman and Steele, Scheme was presented as based on the lambda calculus and as a dialect of Lisp [10, 9]. Over a period of 35 years, Scheme has undergone several revisions under a standardization process described in its charter. Table 1 presents the revisions that were done over the last 35 years from Scheme inception in 1975 until now (circa 2010).

Revision	**Year Ratified or published**	**Authors**
Scheme	1975	Sussman and Steele
R^{1}RS	1978	Steele and Sussman
R^{2}RS	1985	Clinger
R^{3}RS	1986	Clinger and Rees
R^{4}RS	1991	Clinger and Rees
R^{5}RS	1998	Kelsey *et al.*
R^{6}RS	2007	Sperber *et al.*
R^{7}RS	201x	

Table 1. The Scheme revisions from its creation in 1975 until now (circa 2010). The 7th revision is underway and the exact date of its ratification is unknown.

For the R^{7}RS standardization process, the Steering Committee introduced two language levels, "small" and "large" [3]. Eventually, these levels will be given new names. In this paper, we will simply call them Small and Large. Large will be an extension of Small in the sense that any program executing without error on a conforming implementation of Small must run properly and give the same result as executing it on a conforming implementation of Large.

Two working groups will define these two levels. As of June 2010, the working group 1, to define Small, is composed of 16 members, including the chair; the working group 2, to define Large, has 11 members, including the chair. The charter expects these two groups to interact with each other to fulfill the goal of compatibility of the two levels.

The two-level approach was also used for the definition of EuLisp. The two levels are called level-0 and level-1 [8]. Euscheme [4] is a Scheme implementation of level-0. EuLisp was developed to be less minimalist than Scheme (R^{4}RS) and less complex than Common Lisp. The working groups, as well as the Steering Committee, could gain valuable insight from these previous language designs.

The charter states a few guiding requirements for the working groups, but their description is very short with no specific details. This weak guiding approach of the Steering Committee is similar to the previous revisions, but we believe that this should be changed.

So, our main observation is that the Steering Committee has a role that is too weak to effectively guide the two groups in definite directions. We propose the following general amendments to the current standardization process:

1. The Steering Committee should define (and eventually update) a set of detailed guiding requirements for the design of the language levels. These requirements should be created using an iterative process based on the rationales provided by the working groups. This iteration cycle should be short enough to enable timely adjustments of the requirements.
2. A priority mechanism should be put in place to resolve conflicting requirements.
3. The working groups should follow the requirements and their priorities defined by the Steering Committee to support their decisions. They also provide rationales to further support their decisions when current requirements are not sufficient. For some guiding requirements (e.g., efficiency, simplicity of implementation), implementation details need also to be provided by the working groups.

4. One major guiding principles for Small, besides a minimalist approach, should be the efficiency of implementation of the features of Large.

In the following sections we analyze the R^{7}RS charter and show that the proposed amendments would be beneficial to the standardization process.

2. The Current Standardization Process

In August 2009, the Scheme Steering Committee published an amended charter for theR^{7}RS standardization process (see documents at [2]).

The major modification, from the previous charters, was the introduction of two working groups, one for each Scheme level.

2.1 Steering Committee Position Statement and Charters

A Steering Committee published a position statement accessible on the Web [3]. That document makes the following general statement (we have kept the formatting almost intact):

A programming language is a notation that's supposed to help programmers to construct better programs in better ways:

1. *quickly*
2. *easily*
3. *robustly*
4. *scalably*
5. *correctly*

A language that doesn't enable programmers to program is, in the end, doomed to irrelevance.

The goals of the Steering Committee are succinctly stated as

- *we don't standardise or otherwise define the language;*
- *rather, we oversee the process of its definition.*
- *...and, when necessary, we may steer the effort a bit.*

That is, we enable the Scheme community to prosecute the development of the language—its definition, growth, extension, standardisation and implementation. Our chief mechanism is by granting charters to various committees that will carry out the actual tasks, and stamping the results of these efforts with our imprimatur.

Two charters were also defined by the Steering Committee, one for each working group. Both charters have a succinct section on "goals and requirements". In Charter 1, for working group 1, the general goal is to be compatible with and the same size as R^{5}RS. In Charter 2, the specified general goal is to be compatible with a subset of R^{6}RS, essentially taking into account the language features introduced in the latest revision.

Do such criteria form enough guidance in the standardization process? It is a starting point but it does not form a set of guiding requirements on which to base future design decisions. These goals and mechanisms are currently too undirected to make the process efficient and to reach the overall goal of a Scheme language usable by the software developer community.

So what do we propose?

Essentially, that the Steering Committee would gain in defining in much greater details guiding requirements that would be used in their decision for *imprimatur*. In essence, these requirements are pre-defined rationales that the working groups must consider. To be practical, the definitions of the guiding requirements could be refined based on the working groups rationales. This implies a healthy iterative standardization process on which the Steering Committee can build a stronger consensus.

3. Two Guiding Requirements

It is not the aim of this paper to give a list of guiding requirements that would ensure the success of a Scheme design. That would be a far reaching goal. Instead, we suggest two general guiding requirements that we believe are often considered peripheral in a programming language—and certainly has been the case for the previous Scheme standardization processes— but which ought to be considered in the design of a programming language to make it successful in the software developer community: maintainability and debugging capabilities.

We first analyze a maintainability case scenario: patching code. In Subsection 3.2, we succinctly discuss the advantage of adding debugging as a guiding requirement .

3.1 Maintainability

We present an example of applying a guiding requirement to direct the design of two features of Scheme: optional named parameter and multiple values. We will show that software maintainability supports well the need for these language features with some specific semantics. We do not claim that only the following semantics are possible for these features, based on the requirement of software maintainability, but show that there is a need for such basic principle to help guide the semantics.

We first introduce a concrete scenario as part of the maintainability requirement and then discuss the design of two features of Scheme related to this scenario.

3.1.1 Patching Code

As part of the evolution and maintenance of software it is useful to provide software updates that can be downloaded and installed without the intervention of a user. Imagine a software already installed on thousands of computers and needing the correction of a serious defect. The defect requires the modification of a procedure that should behave differently for some callees and return a value of a different type in such a case. The modifications to the procedure should not modify the callees unaffected by the defect. We believe that the simplest mechanisms to correct such a defect is to introduce an optional named parameter and an "optional returned value", which is essentially a multiple values language feature with a semantics that allow such optional returned value.

We analyzed what already exist in the current Scheme implementations and comment on their design based on this scenario and language features.

3.1.2 Optional Named Parameters

In SRFI 89, titled "Optional positional and named parameters," authored by Marc Feeley, the notion of optional named parameters is rationalized, in part, by the following text:

In the software development process it is common to add parameters to a procedure to generalize its function [sic]. The new procedure is more general because it can do everything the old procedure does and more. As the software matures new features and parameters are added at each refinement iteration. This process can quickly lead to procedures with a high number of parameters.

The process of refinement or extension of programs, via procedure modifications, is a maintenance issue: programs are modified with a backward compatibility goal. New parameters are introduced to generalize a procedure.

The implementation of SRFI 89 depends on SRFI 88, which for its efficient implementation requires the modifications of the lexical parser, which is definitely a low level aspect for about any Scheme implementation. In terms of language level, it can be argued that

SRFI 88 belongs to Small if we assume that Small has, as one of its primary goal, the efficient implementation of language features.

3.1.3 Multiple Values

The multiple values feature was introduced in R^{5}RS. It is based on two procedures: `values`, which generates multiple values, and `call-with-values`, which consumes multiple values via a procedure. In R^{5}RS, it is an error if the number of values generated do not match the number of values consumed; in particular, it is an error if the consumer accepts only one value and more than one value is generated. The rationale for raising an error is not given, but one can conjecture that it is a simple approach that catches common type of errors.

One can argue, though, that such an enforcement makes the use of multiple values too cumbersome. In Common Lisp, it is not an error when the number of generated values do not match the number of consumed values. Also, in contrast to Scheme, many primitives of Common Lisp generate multiple values. For example, the function `parse-integer` parses a string to extract an integer and returns two values: the integer value and the index of the characters in the string after the parsing completed. It would be very cumbersome for programmers to always have to specify a context of two values for such a function as `parse-integer` since in most cases only the first value is used.

The multiple values semantics of R^{5}RS is not well suited for the patching scenario since if a function that was returning a single value needs to be modified to return multiple values, to correct a defect, an error will be raised for all callees that are still expecting a single value. If a guiding requirement in defining R^{5}RS existed that promoted debugging, it could be argued that the Common Lisp semantics is preferable.

The R^{6}RS standardization process delivered a rationale document [1] for the design of Scheme version 6, in particular for the semantics of its multiple values feature (Section 11.9.3). In R^{6}RS, it is undetermined if passing the wrong number of values to a continuation is a violation or not. It is left to the implementer to decide between two possible semantics: raising an error if the number of values do not match its continuation or never raise an error in such cases, in particular to use the first value in the case of a continuation accepting a single value. This position is even worse than the R^{5}RS, as far as the patching scenario suggests, since it complicates program portability—which is certainly another guiding requirement cherished by the software developer community. What is more, the R^{6}RS rationale document does not really provide a rationale but only shows the apparent difficulty of the working group to decide between two possible semantics. No rationales, as given above about the cumbersome use of multiple values returned by functions in Common Lisp, are given for either of the two possible semantics. We believe that this a sign of a lack of guiding requirements that R^{7}RS ought to avoid.

3.2 Debugging Capabilities

In 1973, Hoare [6] introduced his main ideas on language design by stating that

> *This paper (Based on a keynote address presented at the SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, October 1-3, 1973) presents the view that a programming language is a tool which should assist the programmer in the most difficult aspects of his art, namely program design, documentation, and debugging.*

Debugging has never ceased to be a necessary task for programmers, simply because programs are like mathematical proofs, they are rarely created without defects or shortcomings.

Debugging is a common programming activity. It spans many types of tasks from simply correcting compilation errors to correcting erroneous results. Almost all language specifications do not include any details about the debugging capabilities that could be provided by an implementation. Debugging capabilities are typically considered as part of a development environment.

We believe this is an error on which the Scheme community could show a distinctive attitude by embedding in the description of its language (at the low level) a set of primitives that enable portable debugging environments to be implemented. The higher level would provide more advanced debugging capabilities based on the low level primitives. We do not advocate the specification, in the standard, of any complex interactive debugging environment. Implementations would provide the complex programmer interactive environment. These could be graphical or text oriented environment, but that would be outside the scope of the language specification.

For some programming languages (e.g., Perl, Python) the debugging capabilities are somehow specified as part of the language as the (unique) implementation of these languages form a *de facto* standard. But Scheme has no such *de facto* standard; this is an indirect weakness of Scheme on which these languages take advantage.

Designing a set of efficient and correct primitives to enable the implementation of useful and precise debugging capabilities is not a trivial task [5]. Depending on the programming language features used, e.g., lazy evaluation, the debugging approach might also rely on quite different techniques [7].

During the last Scheme revision of R^{6}RS, some members of the Scheme community have pointed out that the lost of the `load` primitive and the REPL top level interactive facility impeded on providing a useful (simple) debugging environment. If a requirement for debugging had been part of the standardization process, a strong argument could have been made to keep the REPL.

Essentially, the Scheme standardization process would benefit by seriously considering the mundane task of debugging as an essential part of a programming language. Moreover, it should be recognized that some essential features of R^{5}RS implicitly depends on the requirement of some debugging facility. This requirement should be made explicit by the Steering Committee to ease the standardization process.

4. Efficiency and Separation of Small and Large

In the R^{7}RS charter, Small is described as "lightweight" and Large as "Heavyweight". The Steering Committee makes it clear that Small R^{7}RS should follow the main precept that directed previous Scheme revisions:

> *Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.*

This general guiding requirement, besides other more specific requirements regarding compatibility with R^{5}RS, would benefit by considering efficiency of execution. Some design decisions from past revisions were probably also based on efficiency, but this guiding principle was never stated explicitly.

Efficiency consideration is made more important with the two-level approach since some language features might not belong to Large but to Small base on the observation that their implementation requires some low level details that only Small can handle efficiently. For example, the efficiency observations mentioned in Subsection 3.1.2 for SRFI 88 show that these language features belong to Small and not to Large if we take efficiency as a primary guiding principle.

On the other hand, some language design decisions from previous Scheme revisions were probably influenced by efficiency but they were not proven to be true or at least not supported by any concrete implementation details.

For example, Waddell *et al.* [11] shows that the R^{5}RS `letrec` construct can be implemented efficiently even if the "letrec restriction" is detected and reported. On the other hand, the R^{5}RS specification does not state that an error should be reported. This decision was probably based on efficiency consideration but this fact is undocumented and would need to be revised if it were so.

In summary, only detailed implementation considerations can draw the line between the correct level (Small or Large) to use to implement the language features. Acknowledging this fact in the guiding requirements of the charter would benefit the 2010 Scheme standardization process.

5. Conclusion

The Scheme standardization process has reached an evolutionary point today where there is a need for more rationalization to support its design decisions. We believe that these rationalizations must play a vital role in the form of guiding requirements defined and revised by the Steering Committee.

The R^{7}RS definition process have taken an approach to satisfy two communities of users/programmers with a two-level approach. We have argued that the Steering Committee would benefit by providing more written guidance to define these two levels. This guidance should come in the form of guiding requirements defined to meet the needs of the software developer community. These requirements are essentially pre-defined rationales that the working groups could not ignore in writing their own rationales. The guiding requirements should be iteratively defined, by the Steering Committee, through an iterative standardization process based on the working groups rationales.

We believe that *efficiency* is at the core of the separation between Small and Large, besides the main Small intrinsic precept of lightweight. It would be beneficial if this separation were supported by concrete arguments based on compiler and run-time technologies during the R^{7}RS standardization process.

6. Acknowledgements

We thank the reviewers for constructive criticisms and advises for this paper.

References

[1] Scheme Steering Committee (R^{6}RS). The revised^6 report on the algorithmic language Scheme. Website, 2007. `http://www.r6rs.org/`.

[2] Scheme Steering Committee (R^{7}RS). Scheme Reports Process. Website, 2009. `http://www.scheme-reports.org`.

[3] Scheme Steering Committee (R^{7}RS). Scheme Steering Committee Position Statement. Website, August 2009. `http://www.scheme-reports.org/2009/position-statement.html`.

[4] Russell Bradford. Euscheme: the sources. Website, 2010. `http://people.bath.ac.uk/masrjb/Sources/euscheme.html`.

[5] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.

[6] C.A.R. Hoare. Hints on Programming Language Design. Technical Report CS-73-403, Stanford Artificial Intelligence Laboratory, 1973.

[7] Guy Lapalme and Mario Latendresse. A debugging environment for lazy functional languages. *Lisp and Symbolic Computation*, 5(3):271–287, Sept 1992.

[8] Julian Padget. Eulisp. Website, 2010. `http://people.bath.ac.uk/masjap/EuLisp/eulisp.html`.

[9] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998. Reprinted from the AI Memo 349, MIT (1975), with a foreword.

[10] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An Interpreter for Extended Lambda Calculus. Technical Report 349, MIT, December 1975.

[11] Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of scheme's recursive binding construct. *Higher Order Symbolic Computation*, 18(3-4):299–326, 2005.