

Run Once, Write Anyway

Matthias Radestock
LShift Ltd
matthias@lshift.net

ABSTRACT

Scheme is one of only a few languages with a concise specification and formal semantics. With such a foundation it should be easy to write portable code that, without modification, runs correctly on different Scheme implementations. Yet this goal has proved elusive – there are more than fifty Schemes and writing programs that are portable between even just a few of these implementations is tedious at best and impossible at worst.

We examine the reasons behind the Scheme community's failure to provide developers with a basis for writing portable code. We look at the various Scheme standardisation efforts and identify their contributions and shortcomings. Finally we present a series of recommendations for resolving the key portability issues. Our proposals include a minor revision of the Scheme standard, specifically aimed at improving portability, the creation of a standard library, and a central code repository and package management system.

Categories and Subject Descriptors

D.3.0 [Software]: Programming Languages—*Standards*;
D.3.3 [Software]: Language Constructs and Features—*Modules, packages*; D.3.4 [Software]: Processors—*Preprocessors*

General Terms

Standardization, Languages, Design

Keywords

Scheme standardization, portability, re-use, code repositories, library design

1. INTRODUCTION

Many popular programming languages have thriving communities, including individuals, companies and academic institutions, that write and share code at a grand scale. There are Java, Perl, Python and, to a degree, Lisp libraries and

frameworks that provide ready-made, tested solutions to a vast range of problems. Writing applications in these languages is largely a task of picking and assembling the right combination of frameworks and libraries. In most cases the resulting applications will run unchanged on multiple platforms and different implementations of the language.

The situation with Scheme is quite different. It is difficult to even just share code between programmers using the same Scheme implementation, let alone different Schemes. This is particularly surprising given that Scheme, unlike most programming languages, has a concise specification and formal semantics – factors which should both be conducive to portability. In the first part of our paper we explore the reasons for this failure of Scheme to facilitating code sharing and re-use. We identify both the technical and social causes and explain why the situation is getting increasingly worse.

There is a growing awareness within the Scheme community that the lack of portability is a major obstacle to adopting Scheme as the language of choice for application development. Scheme programmers are frustrated with seeing valuable developer time wasted in the re-implementation and porting of the same fundamental features over and over again. A number of different initiatives have emerged that aim to address this issue; not necessarily as their primary aim, but certainly as an important aspect. The second part of this paper examines the achievements and failures of these initiatives. We also take a look at common techniques and tools for writing portable and re-usable Scheme code.

Based on our knowledge of the language, the community and past&present portability efforts we make a series of recommendations, some of which build on the existing initiatives while others are new. When combined they enable Scheme developers to share and re-use code at a significantly larger scale than is presently possible.

2. THE PROBLEM

Scheme is different from other programming languages in many respects. Probably the two most striking differences (apart from the details of the language itself) are the compactness of the language standard – weighing in at under fifty pages, including a formal semantics – and the sheer number (over 50) of implementations of the language. Both of these have profound implications for portability and code sharing, which we examine in this section.

The Scheme standard and the implementations are both products of the Scheme community and it is the members of that community which are struggling with portability issues. Clearly any investigation of these issues requires that we understand how the community operates. Furthermore, it is in the nature of the problem that a solution requires widespread acceptance by the community and the active cooperation of its members – not just the implementors, in order to handle the technical aspects, but also the users, since it is they who write the majority of the code of interest to other users. We therefore look at two aspects of the community: its traditional application areas and user base, and the challenges to portability posed by a new, emerging usage domain and user base.

2.1 The Scheme Community

Historically, Scheme has mainly been used in the following areas: teaching the fundamentals of programming at universities, as a testbed for programming language research (both in academia and “recreationally”), and for embedding & scripting. Consequently the main users have been

- academics, particularly in the area of programming language research, and programming language enthusiasts,
- ex-students who were taught Scheme at university and came to appreciate its unique features,
- application extenders who have been exposed to Scheme due to its use as an embedded/scripting language.

This is not an accident but the direct result of some of the properties of the language: Scheme has a small core that is easy to interpret and compile. The core is also reasonably close to the lambda calculus, which makes Scheme attractive for trying out ideas in programming language theory. The syntax is extremely uniform and light-weight, which, combined with the macro capabilities, makes Scheme easy to parse and allows language extensions to fit in naturally with the core language. Finally, Scheme’s representation of programs as data make it ideal for manipulating programs.

Over the last few years a new trend has emerged: There is a growing number of users who develop and deploy real-world applications with Scheme as the main implementation language, as an embedded language for handling tasks in which the advantages of Scheme over other languages are most prominent (domain specific languages and advanced control flow are two examples), and as a scripting language for extending and controlling applications.

To application developers, code sharing, re-use and portability are much more important than to academics. Application development is much broader in scope than the typical use of programming languages in teaching and research, which has quite a narrow focus. Thus, compared to academia, in application development there is much more code and many more features for which portability is important. Furthermore, the whole point of research is to come up with *novel* ideas, whereas a substantial part of the work

during application development is about finding the right *existing* solution to a problem.

The deployment of Scheme in the real world needs to take into account commercial considerations such as reducing risks by minimising dependencies on a single implementation, and improving long-term maintainability and reducing training and documentation costs by using code developed and widely tested by third parties. Portability is a key requirement for achieving these.

2.2 The Standard That Isn’t

If there was no language standardisation any kind of code sharing and portability would be impossible. However, portability is only one of the issues addressed by a language standard. We need to examine what the Scheme standard has to offer in this area, where its shortcomings are and, more importantly, why these shortcomings are present.

The main Scheme standard is Revisedⁿ Report on the Algorithm Language Scheme (RnRS), currently in its 5th revision[8], a variation of which is also an official IEEE standard[5]. RnRS has been remarkably successful. Most Schemes aim for compliance with it. No competing standards have been developed, which is a testament to the skills and knowledge of the authors in understanding their subject matter and audience.

Portability was not the main aim in the development of the standard.

Mostly we want to be able to read each others’ code when it appears in the literature ... Portability is only a secondary consideration.[11]

In fact, R5RS is a standard that *encourages* differences between implementations. It is really written for implementors and not users, leaving many things unspecified that users care about, but consequently giving implementors a lot of flexibility when implementing particular features, and facilitating the extension of the language with new features. One typical example of this is error handling. R5RS contains the following wording:

When speaking of an error situation, this report uses the phrase “an error is signalled” to indicate that implementations must detect and report the error. If such wording does not appear in the discussion of an error, then implementations are not required to detect or report the error, though they are encouraged to do so.

Since most errors defined by R5RS are in the latter category this provides a convenient get-out clause for implementors who either do not want to implement proper error handling or, more importantly, substitute some implementation-specific behaviour, i.e. replace the error case with a language extension.

RnRS was developed *by* programming language researchers *for* programming language researchers. If there was to be

any hope for it to be adopted by implementors then any areas on which there was disagreement needed to be avoided. For that reason RnRS was designed by a committee on the principle of consensus. In retrospect it is quite amazing that consensus was reached in as many areas as covered by R5RS, but it is also clearly obvious that many features were left out from the standard not because they were deemed unimportant but because opinions were divided on their best realisation. Error handling is once again an example: R5RS does not specify any means of programmatically throwing or catching an error – functionality that is very important to application developers but for which there are many conflicting designs.

An observation that one can make from looking at the various Scheme implementations claiming R5RS compliance is that it is easy to be *nearly* standards-compliant, but surprisingly difficult to be *fully* standards-compliant. The reasons are:

- The language in the standard is extremely terse. There are many places where missing just a single word would result in a non-compliant implementation.
- The standard does not repeat itself. That, in itself, is a good thing, but unfortunately there is also a lack of cross-referencing. As a result, implementors working on a specific feature may miss important considerations mentioned elsewhere.
- There are some errors and areas of ambiguity in the standard. These are rare and most of them have subsequently been corrected/clarified by the authors. However, the corrections/clarifications have not been combined and released as official errata to the standard. This inevitably has led to divergence between implementations.
- While most of Scheme is easy to implement, there are some areas where the most apparent implementation strategy results in non-compliance and instead considerably more complex techniques are required.
- There is no easy way to catch non-compliances. The standard contains some examples of correct and incorrect behaviour but these are far from comprehensive and, more importantly, since the standard does not specify any error handling mechanism and is generally permissive about what happens when errors occur, it is impossible to devise portable automated compliance tests.

There are other aspects of R5RS that hurt portability. For example it defines no mechanism for extending the reader, and the standard macro system is not general enough. This pushes many desirable language extensions from portable library code into implementation-specific code. Crucially, the only mechanism provided by R5RS for re-using code written by another party is for that party to supply the code as Scheme source files and for the developer to load these files using R5RS's `load` function. There is no mechanism for declaring what features are provided by a piece of code and what features it requires.

2.3 Where Is The Standard Library?

The design of programming languages cannot be separated from that of standard libraries – a programming language without libraries might be an interesting theoretical subject, but writing real programs requires the ability to represent and manipulate data in a variety of ways, employ rich control structures, and access operating system resources. R5RS is lacking in all three, for example

- Lists are the fundamental composite data type in Scheme, yet many of the common functions for accessing and manipulating lists, as well as control constructs operating on lists are not defined by R5RS.
- R5RS defines vectors as a second heterogeneous data structure alongside lists, but it does not define other commonly used collection types, such as dictionaries, or uniform vectors (to complement strings, which in R5RS are really uniform vectors of characters), or a polymorphic collection API. It is also lacking any mechanism for users to define their own composite data types. The character and string data types do not (and cannot) support unicode characters.
- R5RS only provides a very basic interface to files – there is no way to access directory information, create files, delete files, read or write binary files, etc etc. Functions for accessing other operating system resources such as sockets, threads and time are missing completely.

Some of the above can be implemented on top of R5RS, but for every application developer to do this is clearly incredibly inefficient and, since the same feature is likely to be implemented in different ways, not conducive to code re-use and composability. More seriously, the majority of missing features can only be implemented in an implementation-specific manner, which is beyond most application developers and results in code that is not portable between different Schemes.

2.4 50 Implementations And Counting

The more implementations of a language there are the more important portability becomes and the more difficult it is to achieve. Most programming languages have just one or at most a handful of implementations. However, with Scheme the situation is quite different: there are more than 50 Schemes and the number is increasing.

There are many factors that contribute to this increase. Scheme is very easy to implement – a single person can write an interpreter in a few days. Given that the traditional main user group of Scheme are people with an interest in programming language design and implementations, this is a recipe for guaranteeing a steady stream of new Schemes. The main motivation for developing yet another new Scheme is often to implement a particular set of features that are not specified by the standard and not offered by any of the existing Schemes, or to implement a different design of some features. Since the standards leaves out so many useful features, and there are many different ways for implementing any specific feature, the potential number of interesting Scheme implementations is astronomical.

The significant differences between Scheme implementations has led some members of the Scheme community to conclude that Scheme is a *family* of languages, i.e. each Scheme implementation is its own language or dialect. This is reinforced by the tendency of Scheme implementations to implement new features on top of their core implementation-specific features.¹ Thus we get implementation-specific towers of features with increasingly less commonality and portability across implementations.

3. THE WORKAROUNDS

The failure of the Scheme standard to focus on portability, the absence of a comprehensive standard library, and the increasing difficulties this causes in writing code that is portable across the growing number of Scheme implementations, is well-known in the Scheme community. Various initiatives and sets of tools/techniques have been created that facilitate the creation of common libraries and the sharing/porting of code. In this section we explore the strengths and weaknesses of the most prominent of these efforts.

3.1 SLIB

SLIB[7] is the closest thing Scheme has to a standard library. It is written in pure Scheme, is very portable, gets updated regularly and covers a wide range of functionality.

SLIB includes a simple package system. Scheme code can declare the dependency on a particular feature using the statement

```
(require '<feature-name>)
```

Features are mapped to source files via a multi-level catalogue hierarchy comprising:

1. standard SLIB packages,
2. additional packages of interest to all users of the machine on which SLIB is installed,
3. packages specific to the user's Scheme implementation,
4. packages the user wants to always have available,
5. packages for the application the user is running.

SLIB follows a simple strategy for resolving conflicts resulting from a feature having more than one mapping: The last mapping is used in preference to earlier mappings.

A mapping can declare what kind of macro package is used by the source file to be loaded. It can also declare a file as being in compiled rather than source form. Features can be “virtual” – mapped to a collection of other features – and provided explicitly as a side-effect of loading a file.

Portability of SLIB across different Scheme implementations is achieved in two ways:

¹For good reasons since often this is the most efficient way to implement a feature and ensure a degree of coherence between all the features available for an implementation.

- For each Scheme implementation there is an SLIB configuration file that must provide the hooks required by SLIB for interfacing with implementation-specific functionality. SLIB does not have many prerequisites for working, and these configuration files are therefore easily created for most Schemes.
- Some features depend on features not defined within SLIB. These system features need to be provided by the implementation; if they are not then the dependant SLIB features are not available.

SLIB comes with good documentation on installation, porting and the provided features.

The functionality provided by SLIB is not comprehensive enough to serve as a realistic substitute for a standard library. It is particularly lacking in features to access o/s services, such as files, sockets, time, threads. Partially this is because providing these features would require more dependencies on functionality provided by specific Scheme implementation. However, there is no reason why SLIB could not simply just provide the *documentation* for the *expected* functionality provided by a feature and then leave it up to implementations to provide the feature. This would then allow SLIB packages to be written that are in themselves written in pure and portable Scheme but depend on these other non-SLIB packages.

SLIB is really a collection of independent features. Some features internally depend on other features but there is little structure and design for the library as a whole. As a result, SLIB does not possess the degree of coherence and integration one typically expects from a standard library.

SLIB feature dependencies are directly embedded in the Scheme code, rather than being specified separately in a declarative fashion. This complicates the development of tools that analyse dependency information, e.g. for the purposes of compilation and dynamic loading of features from different sources.

The conflict resolution strategy in SLIB does not cater for cases where a feature can be provided in many different ways, i.e. depending on different sets of other features only some of which are available. The multi-level catalogue partially alleviates this problem since these kind of dependency alternatives are typically associated with differences in implementations, and user/application preferences. It is, however, not a general solution.

While SLIB ships with configuration files for many Schemes, it does *not* ship with implementation-specific feature code. As a result, not all SLIB features are available on all systems and there are also many cases where SLIB features overlap with existing implementation features and does not take advantage of implementation-specific optimisations. Arguably the implementation-specific feature code could ship with the Scheme implementations, but for one reason or another this does not seem to be the case.

One final drawback of SLIB is that it is not a community effort but largely the work of a single individual; there is no

defined process for people to write new SLIB packages and get them included in the distribution.

3.2 FFIs

An alternative to writing a Scheme standard library is to leverage existing libraries of code from other programming languages. Given that one of the main application areas for Scheme is embedding and scripting, an interface between Scheme and a host language can serve a dual purpose of interacting with the embedding environment and providing access to libraries available for that language. Such so-called Foreign Function Interfaces (FFIs) are very common among Scheme implementations and, indeed, implementations for many other programming languages. There are even toolkits such as SWIG[4] for building FFIs to C/C++ code for many languages, including different implementations of Scheme.

The main problem with FFIs is that they are inherently tied to the host language, which makes them (and code relying on them) non-portable across Schemes implemented in different languages. Additionally there is often a significant impedance mismatch between the host language and Scheme which makes it awkward to use FFIs compared to libraries specifically designed for Scheme.

There is also a problem with composition that imposes limits on the kind of functionality an FFI-based library can offer: While it is usually reasonably straightforward to call a foreign function from Scheme and calling Scheme from a foreign function, combining the two is often problematic. Key properties of Scheme programs, such as tail-recursiveness and continuation semantics do not cross language barriers in the general case.²

FFIs are not just language but *implementation*-specific. Theoretically they do not have to be, but in practise they are. There are efforts within the Scheme community to standardise a C FFI and it is conceivable that this might get widely adopted since the impedance mismatch between C and Scheme is not that great – C procedures can be modelled as Scheme functions – and hence there is less scope for Scheme implementors to disagree on what such an FFI should look like. The prospects for standardising FFIs to other languages are much less promising. For instance a Java FFI has to map concepts like classes, methods, exceptions (in all their different flavours) to appropriate Scheme concepts, and offer a concise syntax for instantiation, field access, method invocation, exception handling etc. Furthermore, there are differences in the capabilities of the Java reflection API compared to compiled Java code (e.g. the former can penetrate encapsulation, whereas the latter cannot, and the latter can create sub-classes whereas the former cannot) and standardising on an FFI that requires both is likely to be unpopular with implementors.

Finally, relying on FFIs for library access effectively makes Scheme a second-class language that cannot exist independently of a host language. Hence FFIs cannot possibly be a substitute for a Scheme-specific standard library.

²They can if the foreign functions have been specially designed to preserve these properties but this is obviously not the case when the aim is to re-use existing library code.

3.3 SRFIs

The Scheme Request for Implementation (SRFI) process[3] was developed by the community with the specific purpose of facilitating the writing of portable Scheme code. The process allows authors to submit designs and reference implementations for libraries and language extensions, have these discussed and refined, and eventually finalised or withdrawn. The entire process is driven by the author, including the decision of whether to finalise or withdraw. There is a committee of SRFI editors whose role is to ensure that the SRFI process is followed, and that documents adhere to the structure set out by the process. They can reject SRFIs on procedural/structural grounds but not on grounds of their substance. Thus the SRFI process is remarkably different from the RnRS process.

The SRFI process has been quite successful. Since the beginning of 1999 authors have submitted 44 SRFIs, only six of which were subsequently withdrawn. The submitted SRFIs address many of the problems we have identified in R5RS as being detrimental to portability. There are SRFIs for:

- manipulating Scheme’s standard data types,
- new data types, and generic data type constructors,
- control constructs, including exception handling,
- access to operating system services such as threads, time and random numbers,
- syntactic sugar for writing more succinct code

More and more Schemes implement an increasing number of SRFIs and more and more Scheme developers are starting to utilise SRFIs when writing new code.

The main problem with SRFIs acting as a substitute for a more complete language and standard library is that there is no overall design – SRFIs are very much a collection of largely independent features. Consequently there is a lack of uniformity, coherence and coverage. The SRFI process also does not prevent SRFIs from providing overlapping or conflicting functionality. This presents implementors with a problem: it might be impossible for a system to implement two conflicting SRFIs, e.g. conflicting reader macros. Even if conflicting/overlapping SRFIs can be supported by an implementation, users are faced with an integration nightmare when combining code that uses such conflicting features.

There is nothing stopping SRFI authors from finalising a SRFI which is ill-conceived or on which there is widespread disagreement. One would expect Scheme implementors to ignore such SRFIs, in which case the development of these SRFIs is just a waste of time but otherwise causes no damage. However, in reality there is a “box-ticking” mentality among both users and implementors. Some users select Scheme implementations based on the number of features, ill-conceived or not, they provide. Implementors, in striving for popularity for their own Scheme, are therefore incentivised to equip their implementations with as many SRFIs as possible.

SRFIs have not deeply penetrated implementation code. Implementors often do not use SRFIs internally. Partially that is because SRFIs were not around when most current Schemes were first written. Another reason is that there are many different ways of providing the same feature and some implementations, for good reasons (e.g. in order to fit in with implementation internals) or purely subjective reasons choose one over the other. As long as implementations continue to provide their own alternatives to SRFIs, users will write application code that uses the former and thus be non-portable. There is further incentive for users to do so: In the vast majority of cases SRFIs are implemented by porting the reference implementation. Since SRFI authors are encouraged to submit reference implementations that can be ported easily, the reference implementations are often sub-optimal and limiting. Thus application developers are forced to choose between a non-portable but efficient and comprehensive implementation of a feature and a portable but limited implementation.

Most of the problems we have identified above were well known at the time the SRFI process was conceived and are an inevitability given the desire for openness and counterbalancing the authoritarian nature of the RnRS process. In practise there have been few occasions where these problems have manifested themselves. However, this is most likely just due to the fact that the process is quite young and the number of submitted SRFIs is still small.

3.4 SRFI-7

The SRFI process does not define a mechanism for distributing and installing SRFI implementations. Neither is there a way for users to contribute different implementations of a SRFI, e.g. for different Schemes, using different techniques etc. There are however two SRFIs, SRFI-0 and SRFI-7 that allow programs to declare dependencies on specific features and conditionally select program fragments based on the presence/absence of features. The main difference between SRFI-0 and SRFI-7 is that the former specifies an inline conditional expansion construct, i.e. a macro that conditionally selects a textual program fragment, whereas the latter defines a configuration language that is processed independently from the actual source code.

SRFI-7 does not handle the *provision* of features, i.e. programs can declare dependencies on features but cannot declare what features they provide. It is therefore not suitable as a general means for constructing a feature library. Furthermore, feature identifiers are required to be names of SRFIs and hence one cannot express dependencies on non-SRFI features.

SRFI-7 allows Scheme code to be embedded directly in the configuration language, which excludes its use for code that depends on extensions to the Scheme reader.

3.5 Portability Techniques & Tools

Scheme developers have come up with a number of techniques and tools to improve portability of their code. The main technique is to lift all functionality for which there is divergence between implementations into a prelude which is then instantiated per implementation. Tools such as *hive*[10] manage the installation of prelude-equipped libraries by

selecting the required prelude and implementation-specific files and options.

Another approach is to translate the source code of a library between different dialects of Scheme. Several tools have been developed to accomplish this, e.g. *S2*[12] and *scmxlate*[13]. Because the variations between Schemes are so great, no general automatic translation is possible. Package providers have to supply configuration files containing instructions for the translator. In most cases the bulk of the translation is performed by referring to implementation-specific preludes, just as explained above. However, additionally more complex rewrites of the source code can be performed, which makes translators a more flexible tool than preludes on their own.

The main drawback of the prelude and translator approach is that any substantial piece of code requires a whole raft of semi-standard functionality that does not come “off-the-shelf” in many Schemes. Thus preludes of different projects often contain the same chunks of code, resulting in significant duplication and, worse, clashes when several of these libraries are combined in a single application. Preludes and translators also do not take into account installation and user-specific customisations, e.g. a user may have extended their Scheme installation with a particularly efficient implementation of a certain feature but the precluded library will simply replace it instead of using it. A further problem is distribution and synchronisation. Are the preludes and translator configuration files shipped as part of the code that requires them, or as part of the Scheme implementation, or separately? How are the library, preludes, translator configuration files and implementations kept synchronised?

4. THE SOLUTION

The main challenge in addressing the portability, code sharing and re-use problems facing Scheme developers is to come up with a solution that retains the character of the Scheme language, does not alienate the existing Scheme implementors and users, and builds on the existing efforts in this area. We are proposing a solution in three parts: a minor revision to the Scheme standard, the creation of a standard library, and the creation of a centralised archive and package management system for Scheme code.

4.1 R6RS

We believe that a minor revision of the language standard ought to be published as soon as possible. The purpose of this revision is to improve portability between implementations of the *existing* standard (i.e. R5RS), while being backward compatible with existing pure-R5RS code.

4.1.1 Changes

We recommend that the following changes are made to R5RS.

- correct errors discovered since publication of R5RS,
- add clarification and cross-references where needed,
- be more precise about memory usage requirements³

³R5RS clearly defines the requirements for proper tail recur-

and about the behaviour of control constructs in conjunction with `call/cc`⁴

- be more explicit about identifying areas where implementations can differ in observable behaviour,
- be more strict about error reporting, requiring the majority of error situations to *signal* an error,
- add a mechanism to programmatically throw and catch errors, and for each standard procedure/syntax define the errors it can throw,
- make the type system extensible, i.e. add the ability to define new data types that are disjoint from existing Scheme types,
- allow extension of the Scheme reader,
- publish a portable compliance test suite.

Any existing portable code will still be portable in the revised standard. The main improvements are that it will be significantly easier to write new portable code within the revised standard, and that more implementations will fully conform to it.

4.1.2 New Features

The only new features in R6RS are error handling, the extensible type system and reader extension mechanisms.

Error handling is a major source of incompatibilities between Schemes that needs to be eradicated. It is also an essential requirement for being able to provide a portable compliance test suite. SRFIs 34 (exception handling), 35 (conditions), 36 (I/O conditions) are good starting points for the design.

The ability to add new data types is imperative for constructing libraries. This is evident from the fact that several SRFIs employ SRFI-9 (defining record types), which provides such a mechanism. Many Schemes offer SRFI-9 or similar, broadly compatible mechanisms. We therefore do not anticipate their standardisation to be a major stumbling block.

Just like exception handling, reader extensions are an area of major divergence between Schemes. Reader extensions are the only way by which succinct literal notations for new data types can be added to the language. Without this capability, new data types would always be noticeably distinct from Scheme's code data types. SRFI-10 (sharp-comma external form) addresses this particular type of reader extension and can therefore serve as an initial design for this feature in R6RS. However, there are also other types of reader extensions – in principle one may wish to extend the reader to

sion, but does not specify other requirements on space usage. Without the latter it is impossible to reason about the space complexity of programs. Will Clinger published a paper to correct this[6], but this did not make it into R5RS due to bad timing.

⁴For instance in some implementations of `map` invoking a continuation captured inside the procedure passed to `map` may modify already returned result lists, which is not desirable behaviour.

parse a language with a syntax that is completely different from Scheme. Providing such a general extension mechanism perhaps goes a step too far. The alternative is to identify the most common reader extensions and incorporate them directly into the reader design. One example of such a common extension is SRFI-38 (external representation for data with shared structure).

4.1.3 Beyond R6RS

Once R6RS has been published, the community should turn its attention to a more radical revision of the language with the aim of eradicating present warts (e.g. in the following areas: multiple return values, the interaction of `call/cc` and `dynamic-wind`, case sensitivity, non-unicodeness, eval and environments, macros, to name but a few) while retaining the character of the language. Cleaning up the language in this way will improve portability since many Schemes presently provide implementation-specific extensions in these areas.

4.2 A Standard Library

In parallel with the revision of the language standard, efforts should be focused on the design of a comprehensive, coherent and well-integrated standard library. Many of the existing SRFIs should be incorporated, with some tweaks to achieve greater consistency between them. SRFIs for missing features should be solicited. Thus the SRFI process will handle the bulk of the detailed library design and implementation. One should not underestimate the magnitude of the effort though – it is not going to be easy to reach agreement on what should go into the standard library and what exactly the design for each of the features should be.

The standard library broadly needs to cover the same areas already addressed by SRFIs, but do so much more thoroughly and comprehensively:

- APIs for manipulating Scheme's standard data types,
- new data types such as collections (both uniform and heterogeneous), unicode characters, regular expressions,
- APIs for accessing operating system services such as I/O, threading, signals, time, random numbers,
- control constructs for things like resource management, pattern matching on arguments,
- various syntactic extensions that succinctly capture common usage patterns,
- APIs for text processing, sorting, maths and many other areas

Just like the language standard, the library can be divided into primitive procedures/syntax and derived procedures/syntax, with the latter fully implementable in terms of the former. The primitive procedures/syntax need to have complete coverage of commonly available operating system services. Scheme can take its cue here from other languages and existing Schemes which have achieved portability across many operating systems.

The final outcome of the standard library design process is the documentation of all the functionality provided, a portable reference implementation for all the derived procedures/forms, and a portable test suite for the entire library. This allows Scheme implementations to offer the standard library by just implementing the primitive procedures/syntax.

4.3 CSAN

One way to encourage the sharing and re-use of code is to put in place a facilitating infrastructure. The integration of package management systems with open online code repositories has been tremendously successful in this regard – the amount of useful code that gets contributed to Comprehensive Perl Archive Network (CPAN)[2] or the Debian[1] Linux distribution in this way and the degree of re-use in the implementation of these packages is astonishing. We therefore propose that such an infrastructure be created for Scheme – a Comprehensive Scheme Archive Network (CSAN).

4.3.1 Repositories

CSAN online repositories, of which there can be several, provide catalogues containing declarative information about *features*, *packages* and *bundles*. A feature is a documented piece of functionality. Packages provide features depending on the availability of other features. A single package defines several *flavours* – mappings to installation files of combinations of required features and resulting provided features. Installation files contain ordinary Scheme code that typically perform tasks such as source code translation (along the same lines as `scm2late`) and compilation, and produce module definitions for particular flavours of module system. Bundles contain several packages and are made available as archive files by repositories. The files in a package can be Scheme files or files in any other source code or binary format.

The declarative information held by the CSAN catalogues is similar to a combination of SRFI-7 program declarations and SLIB provide/require clauses. Unlike SRFI-7 and SLIB there is a clear separation of interface (feature), implementation (package), and distribution (bundle). Additionally, meta information such as the author, licensing details, and links to documentation is stored.

4.3.2 Tools

CSAN packages are installed by the CSAN package installer. Given a set of a-priori provided features (i.e. features present in a particular user's Scheme installation) and a set of required features, it determines what flavours of what packages need to be installed and the dependencies between them. For all required package flavours that have not been installed yet, the package flavour's installation files are loaded. The package installer operates on a local catalogue and local repository of bundles. It tries to provide required features using packages from locally available bundles. If that is not possible, it informs the users what bundles need to be downloaded and installed. A portable version of the CSAN package installer is provided as a CSAN package.

Complementing the CSAN installer is a portable query and update tool. It allows users to update their local catalogues

by merging catalogues from multiple remote repositories, find out what features can be provided using the currently installed bundles, what bundles need to be installed in order to provide a specific set of features etc, and install specific bundles or all bundles required in order to provide a certain set of features.

CSAN online repositories are maintained through online tools that allow the creation and update of feature, package and bundle descriptors, and the uploading of distribution bundles. Programmatic and command line versions of these tools are also made available. They, and the portable installer and query/update tools are built on top of a CSAN API. This is the foundation for building implementation-specific and other CSAN tools.

4.3.3 Content

To encourage uptake of CSAN by Scheme implementors and library developers, an initial online CSAN repository ought to be created and seeded with SLIB and SRFI packages and some of the major portable Scheme libraries, such as SSAX[9]. The whole of SLIB can be re-packaged and distributed as a CSAN bundle with little effort, with SLIB features becoming CSAN features. Similarly, SRFI implementations can be packaged and distributed as a CSAN bundle and programs using SRFI-7 can be transformed into CSAN packages. SSAX and other libraries that use the prelude-approach to portability can be hosted on CSAN by defining CSAN features which capture all the required functionality, turning the preludes into packages that require some of these features and provide others, and turning the main library into a package with dependencies on the features provided by the standard prelude packages. CSAN can also be the distribution mechanism for the proposed standard library and compliance test suites.

We fully expect that many CSAN packages will initially have dependencies on particular implementations (which are just features in CSAN). This will in effect partition the CSAN repository into implementation-specific sets of packages. However, there will be an increasing incentive of developing packages that provide features across a variety of implementations. CSAN provides a framework for doing this and sharing the results. Thus the partitions will gradually disappear.

5. CONCLUSIONS

The lack of portability of Scheme code is becoming an increasing problem for the Scheme community and seriously impedes the deployment of Scheme outside an academic and experimental context. Existing efforts by the community to address this have only been partially successful. It is in the nature of the problem that any solution requires broad acceptance by the Scheme community, which includes both developers of Scheme systems and users. Our proposed solution – a minor revision to R5RS, the development of a standard library, and a code repository and package system for Scheme code – avoid radical revision of the Scheme language or standardisation of controversial features, and try to build on past and present community efforts.

The standard revision is specifically aimed at improving portability, does not alter the character of the language

and retains backward compatibility with existing portable code. The development of the standard library takes into account the work that has already been done in this area and utilise the existing SRFI process as the primary source for detailed design and implementation. The CSAN code repository and package system require no alterations to the language and leave implementations with ample scope for differentiation. Furthermore, existing code libraries such as SLIB and SRFIs can be easily ported to CSAN.

Perhaps the biggest concession we have made for the sake of attaining widespread agreement on our proposals is not to attempt to standardise on a module system. The concept of modules covers a vast spectrum of interrelated features such as namespace management, encapsulation / information hiding, separate compilation, dependency definition and analysis, phase separation, parameterisation, language selection, distribution, versioning, dynamic loading of code. All of them, in one way or another, are related to making code more reusable. Programming language researchers have come up with many different ways of handling each of these aspects; opinions are divided on which is better and in many cases it is quite clear that no perfect solution has been found yet. Module systems are also tightly coupled to macro expansion, compilation and other very implementation-specific tasks. An attempt of standardisation in this area is therefore doomed to fail. CSAN's package system overlaps with module systems in some areas, but generally the two are quite separate and CSAN should work well alongside a variety of module system.

Many of the details of our proposal still need to be worked out. Given that the existing RnRS process appears to be stalled⁵, a new committee is required to drive the process of agreeing on the proposed R6RS, its future revisions, and the standard library. How this committee is selected and under what rules it should operate is up for debate. Also, clearly a lot of work remains to be done on the design and implementation of CSAN. We have our own ideas in each of these areas but have deliberately not included them in this paper since we did not want to preempt discussion on these matters. We also fully expect these discussions to lead to alternative proposals.

The main aim of this paper has been to set the scene for and stimulate a serious and focused debate on resolving problems of portability, code sharing and re-use in Scheme. We believe that the implementation of our or similar proposals will significantly change the dynamics of the Scheme community. People will start writing new features based on other people's existing features, and port existing features to other Schemes – rather than re-inventing everything themselves – and Scheme will finally have been turned into a language for serious application development.

6. ACKNOWLEDGEMENTS

We would like to thank the Schemers at LShift, the SchemeUK user group, and everyone on `comp.lang.scheme` and `#scheme` for the many exhilarating discussions on the subject matter of this work. We are particularly grateful

⁵The most recent version of the standard was issued in 1998 and it was only a relatively minor revision on R4RS from 1991.

to Tom Berger, Michael Bridgen, Taylor Campbell, Tony Garnock-Jones and Scott Miller for their insightful comments on early drafts of this paper.

7. REFERENCES

- [1] Basics of the debian package management system. The Debian GNU/Linux FAQ http://www.debian.org/doc/FAQ/ch-pkg_basics.en.html.
- [2] Comprehensive perl archive network. <http://www.cpan.org/>.
- [3] Scheme requests for implementation. <http://srfi.schemers.org/>.
- [4] Simplified wrapper and interface generator. <http://www.swig.org/>.
- [5] IEEE standard for the scheme programming language. IEEE Std 1178-1990, 1995.
- [6] W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.
- [7] A. Jaffer. SLIB. <http://www.swiss.ai.mit.edu/~jaffer/SLIB.html>.
- [8] R. Kelsey, W. Clinger, and J. Rees. Revised⁵ report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.
- [9] O. Kiselyov. S-exp-based XML parsing / query / conversion. <http://ssax.sourceforge.net/>.
- [10] K. Lisovsky. Hive source code manager. <http://www196.pair.com/lisovsky/scheme/hive/index.html>.
- [11] J. Rees. Mailing List Archive of the RnRS authors discussions <http://zurich.ai.mit.edu/pipermail/rrrs-authors/1986-June/000471.html>, June 1986.
- [12] D. Rush. S2 - a scheme to scheme compiler. <http://mangler.sourceforge.net/>.
- [13] D. Sitaram. scmxmlate. <http://www.ccs.neu.edu/home/dorai/scmxmlate/scmxmlate.html>.