

# Formal Verification of Scheme Module and Library Transformations

Aaron W. Hsu and Michel Salim  
*Computer Science, Indiana University*

28 April 2009

## Abstract

Scheme has a wide and varied history of module forms. These forms are sometimes called libraries, but they almost always differ in semantic verbosity. The Sixth Revised Report on Scheme (R6RS) defines a library form, but few people enjoy the tedium of manually translating one module form to another. Reliably transforming one library or module form to another is a problem most people have chosen to avoid. This report details the authors' efforts to create formally verified transformations over a subset of library forms that preserve the semantic meaning of the library form. A simplified set of library syntaxes are examined and their transformations proved. Problems and limitations of the library subset are considered, along with solutions or possible extensions to the semantic model that would remove the limitations.

---

## 1. Problem Description

It is desirable to have verified transformation algorithms from one Scheme module form into another, to facilitate the automatic porting of given libraries. Given a verified algorithm, some degree of certainty could be obtained in automatic tools which convert libraries. This makes them more useful to deploy without intervention in public interfaces such as library repositories.

For example, Chez Scheme has a module system, a simple example of which may look as follows:

```
(module example_library
  (proc1 proc2)
  (import scheme)
  (import dependent_library)
  ...)
```

Which which transforms to the equivalent R6RS form:

```
(library (example_library)
  (export proc1 proc2)
  (import (scheme)
    (dependent_library))
  ...)
```

Each of these libraries is just an S-expression, and they both have the same semantic meaning to some extent. Names are different in R6RS, but if a standard naming convention is used, then these libraries are interchangeable. At the root of the problem, one wishes to take a library form written for one system, and construct the equivalent form

for another system, such that the source contained in the library requires minimal modifications.

This requires a definition of equivalence. Some definition of equivalence must be constructed that states in a formal manner the notion that a given set of libraries forming a program in one system, when transformed according to a verified transformation procedure, will result in a set of libraries that will evaluate and form an equivalent program in the target system.

For this example, consider the following grammars for Chez modules and R6RS Libraries:

```
chez-module →
  (module name ( name ... )
    (import name) ... )
r6rs-library →
  (library (name)
    (export name ...)
    (import (name) ...))
```

Implementing a simple transformer in Scheme, one might use something similar to the following:

```
(define (chez->r6rs mod-expr)
  '(library (, (cadr mod-expr))
    (export ,@(caddr mod-expr))
    (import
      (map (lambda (e)
            (list (cadr e)))
          (filter
            (lambda (e)
              (eq? 'import (car e)))
            (caddr mod-expr))))))
```

Now, some notion of equivalence for this example should be developed. Three obvious parts to the library represent the main components, which will form the basis for a notion of semantic equivalence. Firstly, there is the library name, which is represented here by a symbol. Secondly, the set of symbols in the export expression of each form. Thirdly, the ordered set of import names. While the exports may be rearranged without consequence to the semantic meaning of either of these forms, in Chez Scheme, the order of imports does matter, and thus, the third criterion must respect ordering, while the second need not. Here, one might obviously use a set of sets — unordered and ordered — to represent libraries. From each of these forms, a complete set of symbols for exports, imports, and a symbol for the name might be extracted. The simplest notion of equivalence then, that appears useful, is whether the same set is extracted from any two forms, given a proper extraction function. This notion will receive more complete development in the later sections.

This report comes in two main sections. The first will discuss the approach taken to solve a restricted set of library transformation problems. It will discuss some of the proof approaches as well as the overall semantic model and the general relationships that ought to be proved when doing this kind of work. The second section will discuss limitations in the chosen semantic model for libraries, and will approach each of the objections to the model individually. For problems that have solutions which maintain a pragmatic model, the necessary features of such extensions to the original model will be given and the approach listed.

## 2. Proving and Implementation

Before examining the implementation in PVS of module transformations, what actually ought to be proved? The goal is to prove that a given transformation from one module form to another, and its inverse transformation, that these transformations maintain semantic equivalence between the two module forms. In order to do this, some semantic model of libraries must be created which adequately expresses the salient features of module forms. We are concerned with three main features: the name of the library, its exports, and its imports. We can specify the model more formally as follows:

$$\begin{aligned}\mathcal{S} &= \mathcal{N} \times \mathcal{E} \times \mathcal{I} \\ \mathcal{N} &= \{ n \mid n \text{ is a name} \} \\ \mathcal{E} &= \{ e \mid e \in \mathcal{N}^* \} \\ \mathcal{I} &= \{ [n_0, n_1, \dots, n_k] \mid n_i \in \mathcal{N} \}\end{aligned}$$

The above represents abstractly the features of a library. That is, it represents its name, which we restrict here to being symbols; its exports, which we say are sets of names; and, its imports, which are ordered sets of names. This suffices to adequately represent our model of libraries in the restricted sense we give it. This restricted set obeys the following rules, each library:

- 1) exports only procedures, not syntax;
- 2) specifies all its imports and exports at the head of the form;
- 3) uses only a single symbolic name for its name;
- 4) and, uses only libraries having common names between the two implementation module forms.

This is enough to represent simple libraries, but as shall be seen in the next session, suffers from serious limitations.

Given this representation, we also presume that we have a domain of libraries and modules:

$$\begin{aligned}\mathcal{M}_c &= \{ m \mid m \text{ is a valid Chez } \mathbf{module} \} \\ \mathcal{M}_l &= \{ l \mid l \text{ is a valid R6RS } \mathbf{library} \}\end{aligned}$$

Given this, for each transformation, we must define two functions that convert back and forth from the syntactic forms to the semantic model, for Chez and R6RS forms, this means that we have four total transformation functions:

$$\begin{aligned}\tau_c &: \mathcal{M}_c \rightarrow \mathcal{S} \\ \tau_l &: \mathcal{M}_l \rightarrow \mathcal{S} \\ \tau_c^{-1} &: \mathcal{S} \rightarrow \mathcal{M}_c \\ \tau_l^{-1} &: \mathcal{S} \rightarrow \mathcal{M}_l\end{aligned}$$

Note here that while the main transformation functions  $\tau : \mathcal{M}_t \rightarrow \mathcal{S}$  should be proper functions, having only one semantic representation for every module given to it, this does not mean that these “inverse” functions are likewise proper functions. In fact, we completely expect and desire them to not be proper functions.

While a single library form ought to have one semantic interpretation (ambiguity would be a poor quality for any library system), it could happen that

multiple library forms may actually have the same semantic meaning. This should come as no surprise, given that there is more than one way to write a program to do any one task. This means, however, that we can not simply check to see that  $\forall x \in \mathcal{M}(\tau^{-1}(\tau(x)) = x)$  because  $\tau^{-1}(x)$  may actually legally evaluate to a syntactic form that is *different* than the original syntax  $x$ , while still having the same semantic meaning.

Instead of checking this property of each library transformation pair, we can instead think about classes or sets of library forms each of which has the same semantic meaning.

$$\Gamma_m = \{ s \mid s \in \mathcal{M}_m^* \\ \wedge \forall l_1, l_2 \in s [\tau_m(l_1) =_\sigma \tau_m(l_2)] \}$$

That is, the above is the set of all sets of libraries where each of the modules has the same semantic representation as determined by  $\tau_m$ . Of course, thus far, we have given no semantic equivalence for  $\mathcal{S}$ , but we will do so now. Semantic equivalence (as used by the  $=_\sigma$  operator) operates on the elements of the 3-tuple of  $\mathcal{S}$ , and we may define it thus:

$$\langle n_1, e_1, i_1 \rangle =_\sigma \langle n_2, e_2, i_2 \rangle \\ \iff \\ n_1 = n_2 \wedge e_1 = e_2 \wedge i_1 = i_2$$

This assumes that we have a definition for symbolic equality as well as for the equality of sets. This is not hard to do, so we assume that we have these. Given this, we now have enough to build a proper function that will help us to verify the consistency of the  $\tau$  functions. That is, we want to verify that converting to and from a semantic representation will not result in a module that is not semantically equivalent to the first.

To develop this idea of consistency, we first need a function whose domain and co-domain are the same, so that our later efforts are easier to characterize. We will call this function  $\gamma_m$ , whose signature is defined thus:

$$\gamma_m : \Gamma_m \rightarrow \Gamma_m$$

We define  $\gamma_m$  as:

$$\gamma_m(s) = s \cup \{ \tau_m^{-1}(\tau_m(l)) \mid l \in s \}$$

With this function, we can then utilize the idea of a fixed point to develop a theorem for the consistency of a pair of  $\tau$  transformations:

### Consistency lemma for $\tau_m$ transformations.

$\tau_m$  and  $\tau_m^{-1}$  are consistent if and only if  $\forall x \in \mathcal{M}_m [\exists y \in \Gamma_m (x \in y \wedge \gamma_m(y) = y)]$

This states that any module we may use has a corresponding set which contains it that satisfies the requirements of  $\Gamma_m$  and that repeated applications of  $\gamma_m$  do not return a different set. That is, every library form has a corresponding fixed point set for  $\gamma_m$ .

Having demonstrated that the transformations from a given syntax to semantic representation and back is consistent, we may trivially specify a function to transform one library form to another library form as follows:

$$\rho_{m,n} : \mathcal{M}_m \rightarrow \mathcal{M}_n$$

defined as:

$$\rho_{m,n}(l_m) = \tau_n^{-1}(\tau_m(l_m))$$

We leave the reader to consider the form of the theorem to prove for this function, which is very similar to the consistency lemma.

## 2.1. Using PVS to Prove Consistency

Initially, most of the definitions were straightforward to design. However, it should be noted that some of the proofs resulted in conditions and requirements for proofs that went beyond simply using `grind` or the like strategies to solve the problem quickly. Instead of attempting to prove each of these without the assistance of `grind`, it was found much easier to rewrite some of the code to match the needs of `grind`. This mean removing the use of sets [available in the prelude of PVS] and the reimplementation of a limited subset of sets in what amounts to Scheme code. Additionally, PVS was very helpful in determining what limitations existed in our assumptions about what libraries should look like, and helped to create the type checking and `well_formed` predicates without the tedium of actually thinking about them ourselves.

The further we progressed in writing the code, the more closely the programs looked like Scheme code, which may or may not indicate anything interesting. However, it is interesting that the form of Scheme code one would normally write for naive and simple implementations of certain structure is so easily proved in PVS.

To represent the syntax of library forms, we used S-expressions defined as a datatype:

```
sexp: DATATYPE
BEGIN
  nil: null?
  symbol(value: string): symbol?
  cons(car: sexp, cdr: sexp): pair?
END sexp
```

To define the abstract semantic model for libraries, we used a record to each of the most important features for libraries:

```
module : TYPE =
  [# lib_name: sym_t,
   lib_exports: list_of_names,
   lib_imports: list_of_names #]
```

Note that we use `list_of_names` to represent exports as well as imports. Since imports are ordered, it makes sense to represent them as lists. However, why do we use lists for exports? Originally, we implemented this using just sets for exports. However, this requires the use of extensionality to prove some things. Rather than bother with this, creating our own member and set equivalence functions proved to be much easier to do. It was then possible to prove results related to `lib_exports` without the need of extensionality.

We then defined some convenience predicates to define types such as `list_of_names` and the like, but these are not of interest. We define our semantic equivalence as follows:

```
mod_eqv?(m1: module, m2: module):
  bool =
    symbol_eq?(lib_name(m1),
               lib_name(m2))
    & list_set_equivalent?(
      lib_exports(m1),
      lib_exports(m2))
    & list_equal?(lib_imports(m1),
                  lib_imports(m2))
```

Our transformation definitions were no more difficult than some record accessors and a few conversion utilities, and overall, writing the code was not that difficult.

```
r6rs_consistency: LEMMA
FORALL (x: r6rs):
  EXISTS (y: valid_lib_sets):
    member(x, y)
    and set_equiv?(g_l(y), y)
```

The real issue arises when the weaknesses of this system are detailed. The number of additional issues scales up greatly when you get away from such a restricted set of library forms.

### 3. Limitations and Extensions

Before we discuss some of the issues related to our semantic model, it would help to have a better picture of what modules and libraries actually do in Scheme. Scheme incorporates the use of Syntax and the potential for syntactically oriented module systems, which creates a set of new problems to deal with when translating between different models, because these different models are not all semantically equivalent.

For example, Chez Modules are built on the Syntactic model of Modules, where modules might be considered convenience forms that allow more fine grained control over lexical scoping of names. However, R6RS Libraries solve a fundamentally different problem, which is the issue of how to package a set of code definitions and expressions into a form that permits distribution and incorporation into other systems without modification. While it may not do this entirely, it still follows this model, and thus, the two libraries diverge semantically.

At their heart libraries try to make a set of names and bindings available to a piece of code which relies on that library. It also itself relies on other libraries that import other pieces of code, and have their own exports of names and bindings. However, in Scheme, procedures are not the only names and bindings that can be imported or exported. Scheme also permits the exportation of syntactic bindings to names. While the names and namespaces are the same as for procedures, their are fundamentally different entities when it comes to their evaluation rules. Syntax forms are evaluated during expansion time (usually) and evaluate to a form of syntax which the compiler or interpreter for the implementation understands enough to run without further expansion.

During this expansion phase, this means that there are also implicit dependencies on the forms to which a syntax form expands that must also be handled (in some cases) by the expander of the library forms. More results relating to this problem will be dealt with in this section.

Our semantic model assumes that all exports and imports are procedures, and thus, they do not have the same kind of abstract “dependency” on other procedures that syntaxes do.

Moreover, it could also happen that libraries themselves could be first class objects. This would require that you could export the name of a library and allow it to be used in an importing form. Chez Scheme has a partially capable form of first class

modules using its `module` form, but it is incomplete. R6RS libraries do not have this capability at all, and arguably, should not have this capability.

The following sections detail some of the issues and present possible solutions and extensions to our semantic model that may allow for these limitations to be removed. No attempt has been made to prove these, however, and their proofs may be more difficult. Even with these extensions, it is expected that some level of competency could be reached in a semantic model that would admit almost all library forms currently in use, while still being formally verifiable.

Many of the issues that arise in the following sections result from our choice of avoiding the actual code expansion that usually, but not necessarily, takes place during library evaluation and dependency graph building. If we were to retain all of the code in the library, then we could expand all that code, however tedious, and derive all the needed information. Such a task is obviously too large for a formal proof, and would be riddled with errors. Instead, a simplified model can be constructed, usually with the help of somewhat magic oracles that provide us needed information from a black box, where proofs dare not enter. Doing so avoids having to prove the validity of an entire Scheme implementation, while still providing useful results based on the unproven, but safe assumption that a Scheme implementation's expander can give us accurate information regarding the code.

### 3.1. Within a Module

#### 3.1.1. Nested Modules and Import Ordering

Chez imposes an ordering on imports: a module can only be imported successfully after all its module dependencies have been imported.

Some module systems, such as R6RS, do not specify such an ordering. There is thus a problem when translating from an order-independent system to an order-dependent one

```
(module ()
  (module A (B)
    (module B (...) ...)
    ...))
(import A)
(import B))
```

In this case, B is only visible after A is imported, and reordering the import would cause a problem.

The module might also be defined using some `include` directives, or generated on-the-fly; in these cases, import lines cannot be reordered.

This is a class of module forms that cannot be trivially ported to R6RS; however, it is instructive to discuss how the semantic model can be extended to support such a module system.

Since our model already enforces an ordering on imports, it is easy to go from a more limited system back to Chez, since the expression is not there. Going from Chez back to R6RS may not actually be possible though, except for limited cases. For example, we may have to rely on code analysis to tell us that the `include` expressions interspersed throughout the code with corresponding `import` forms only load clean module forms into the system, in which case we may be able to transfer the code out without having to rely on an ordering, and thus, could make it work for R6RS. For more complex configurations however, this may not be possible.

Additionally, this presents a problem to the PVS model, since there is no obvious way to model this except by restricting the forms to not have such features. Given an oracle that could analyze a code form and tell us whether it was order independent, this would result in an almost insignificant change to the PVS code, but it would require us to prove more about the actual oracle, which would likely be relatively complex if it were to be truly useful.

#### 3.1.2. Interspersing imports

The Chez module system allows for imports to be used whenever definitions can be used. This is in contrast to R6RS, where imports have to be done at the beginning of the library definition.

In theory, this could introduce semantic difference in the translation process. In practice, the limitation that imported names could not shadow already-defined names probably means that most libraries will have position independent imports. In fact, some imports in Chez are interspersed purely to keep the imports together with the code that relies on them. However, this would, again, require a level of code analysis to determine whether the imports were actually position independent and order independent. The code for this oracle would probably be less than the other, if we eliminated the requirement for `include` support.

Potentially, it is possible that the interspersing of imports only affects code that is incorrect. That is, it serves mostly to catch errors or uses of names which ought not to be available. If this were the case, it could be conceivably legal to restrict our

sense of semantic equivalence to only performing equivalently on good code that does not produce errors. In this case, if it could be proved that this is the only affect on semantics that Chez’s interspersing feature has, no oracle would be necessary, and we could assume all imports occurred first.

### 3.1.3. Macros and Implicit imports

In Chez, the list of export can contain either identifiers, or a list of identifiers, the first of which is visible in the import scope, while the rest of identifiers is only visible in the syntax expansion of the first identifier. This allows for syntaxes to be exported which rely on names not anticipated to be visible within the importing module without being forced to export these dependencies explicitly, and maintaining a nicer scoping. Here is an example of such a library:

```
(module a ((my-syn a)
  (import scheme)
  (define a "Done!")
  (define-syntax my-syn
    (syntax-rules ()
      [(_)
       (printf "~a~%" a)])))
```

If this library were exported without the `a`, it would result in an identifier out of context error. In R6RS, this is not needed: all the necessary identifiers are implicitly exported, with the same behavior as in Chez, namely, that they are only visible locally. This means that any syntax which expands to a dependency which is implicitly exported will find that name visible, and no out of context error will occur.

Transforming from Chez to R6RS form is trivial: the extra identifiers are superfluous and can be stripped. The question is what to do in the reverse transformation. In order for a valid Chez module to be outputted, the identifiers referred to by any exported macros have to be identified and turned into implicit imports.

We can extend the model of exports in a model to incorporate this dependent information, and use an expander oracle which tells us the additional dependencies. This would mean that we would modify  $\mathcal{E}$  as follows:

$$\mathcal{E} = \{e \mid e \in \{\mathcal{N} \times \mathcal{N}^*\}^*\}$$

### 3.1.4. Renames

Some library and module systems support concepts that allow one to limit the exports of one module that are imported into another. These typically take the form of `rename`, `except`, `only`, and `alias`. However, not all the library systems have all of these features. Some of them can be simulated by the others. For example, `alias`, which usually associates a second, additional name for a name that will be imported from a library, can be simulated by writing an additional definition in the code of the importing library, or it may be able to do some other import, such as importing a library twice, using `only` and `rename` to import the aliased name in. It is relatively easy to represent these additional features in the semantic representation by having a function  $\mu_m : \mathcal{N} \rightarrow \mathcal{N} \cup \perp$  that determines whether a library should be imported, and what the imported name should be. If these functions could be identified by name, then they could be transformed reliably back into syntax.

### 3.2. Versioning

Despite the flexibility we could introduce for libraries remapping names and selectively importing names in the previous section, we cannot represent R6RS versions in Chez Scheme’s module system. While we could implement a subset of such versioning, by using some sort of naming Scheme, it would require some additional work, and would not be worth it, since the translation would be forced to adhere to a set of naming conventions which the transformer has no possibility to enforce on the user.

Since R6RS allows you to associate a “version” number set of some kind with libraries, the easiest way of dealing with this would be to embed these into the semantics of the imports and names the same way that we did with renaming, except using a version set or tuple instead of a function. This would work for names as well as imports. Some library information may be lost in the translation from R6RS to Chez modules, however, and this would irrevocably break the main theorems unless we allowed for only a restricted subset of equivalence in going one way and the other.

What this means is that we cannot guarantee semantic equivalence at this level.

#### 3.2.1. Export

As mentioned before, it is possible to go one way in version transformations, but not necessarily the other. We can treat an unversioned module in a version-optional module format as if it is at

version 0, and fill in the missing version information before transformation, thus reducing the cases to be considered to unversioned and versioned.

When transforming from versioned to unversioned, it is necessary to encode the version number in the module name. However, this is a hack that cannot guarantee the semantics of the system.

Further trouble would arise when the two module systems have incompatible version schemes.

### 3.2.2. Import

In addition to the issues above, imports in some dialects may specify a *\*range\** of versions rather than a single one. The most general one is that of R6RS:

```
version reference →
  ( subvref1 ... subvrefn )
  ( and version reference ... )
  ( or version reference ... )
  ( not version reference ... )
subvref →
  sub-version
  (>= subv )
  (<= subv )
  ( and subvref ... )
  ( or subvref ... )
  ( not subvref ... )
```

As we currently do not export version information, we also do not handle versioned imports. Handling this would require an extension of our model:

- The import list is no longer a list of names, but a list of tuples  $\langle name, version\_expr \rangle$
- A version expression will look like the R6RS version reference definition

## 3.3. Library Name Transformation

Different Scheme variants have different naming schemes for standard modules, which are currently ignored. Some issues are listed below:

### 3.3.1. Paths

Module systems vary in the way modules can be named: Chez restricts module names to symbols; PLaneT requires a three-tier hierarchy: author, package, followed optionally by a filename defaulting to `main.ss`; whereas R6RS allows a multi-level path.

In addition to that, PLaneT's old-style naming uses strings, whereas both Chez and R6RS use symbols for the components of module names. This

is trivial to handle, as strings and symbols can be used interchangeably in this case. It might be a good idea to use strings internally, to preserve case sensitivity, but this is not a pressing concern, as Scheme identifiers are normally case-insensitive.

Storing the module name as a list (of length 1 or more) of symbols would be sufficient: when converting to a format that takes only a single token, an encoding could be used to join the list elements. More challenging is the conversion to a 2-element format (such as PLaneT's): going from a 1-element name, an identifiable token can be inserted (and removed on the reverse transformation). Going from a 2-or-more element name, there are two choices: either introduce the same token, or take the first element as the first part. Either way, the rest of the path need to be joined to form a single element.

For uniformity, using a token for both might be more consistent. Consider the R6RS case:

```
(import (rnrs) (rnrs r5rs))
```

If a token is only added when necessary, the two would become `(TOKEN rnrs)` and `(rnrs r5rs)` and thus prefix matching can no longer be used to determine the relationship between two module paths.

$$\mathcal{N}_{Path} = \{ [n_0, n_1, \dots, n_k] \mid n_i \in \mathcal{N} \}$$

### 3.3.2. Mapping

In the simplest case, a module in one Scheme implementation corresponds to a module with the same name in another implementation. This is assumed to be the default, and no name translation need to be done during the transformation.

Generally, however, modules on different systems do not correspond exactly to each other. The standard Chez module, `scheme`, does not export exactly the same set of definitions as the standard R6RS library `rnrs`.

Fixing this would require having a bidirectional mapping between identifiers and the modules where they are defined.

The import directive then needs to be checked; any module that are known to be problematic (because we take the trouble to map its definitions) would be removed from the import list, and the module body would have to be scanned to see which definitions from that import are used.

This is likely to yield a lot of imports for anything but the most trivial library, but we note that this is the accepted norm in, e.g. Java development, where the imports are handled semi-automatically by the IDE.