Implicit Phasing for R6RS Libraries

Abdulaziz Ghuloum and R. Kent Dybvig

Department of Computer Science, Indiana University, Bloomington, IN 47408

{aghuloum,dyb}@cs.indiana.edu

Abstract

The forthcoming Revised^6 Report on Scheme differs from previous reports in that the language it describes is structured as a set of libraries. It also provides a syntax for defining new portable libraries. The same library may export both procedure and hygienic macro definitions, which allows procedures and syntax to be freely intermixed, hidden, and exported.

This paper describes the design and implementation of a portable version of R^{6}RS libraries that expands libraries into a core language compatible with existing R^{5}RS implementations. Our implementation is characterized by its use of inference to determine when the bindings of an imported library are needed, e.g., run time or compile time, relieving programmers of the burden of declaring usage requirements explicitly.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—modules, packages; D.3.4 [*Programming Languages*]: Processors—compilers; preprocessors

General Terms Languages

Keywords Scheme, binding phases, hygienic macros, libraries, macro expansion

1. Introduction

The language defined by the current draft of the Revised^6 Report on Scheme (Sperber et al. 2007b) differs from the Revised^5 Report (Kelsey et al. 1998) language most noticeably in that it is structured as a base library along with a set of additional libraries. It also provides programmers with a `library` form via which new libraries may be defined. Libraries define new syntactic constructs by associating exported keywords with macro transformers and new variable bindings by associating exported variables with computed values, which are often but not always procedures.

A useful feature of R^{6}RS libraries is that a library may export both variable and keyword bindings. Because the transformations implemented by the keyword bindings are *hygienic* (Kohlbecker et al. 1986), references to identifiers introduced by an exported macro resolve to references in the lexical scope of the macro definition, i.e., to bindings within the exporting library that may or may not be exported explicitly by the library. This allows programmers to keep hidden any bindings that should not be visible outside of a library without inhibiting their use in the output of a macro transformer. Requiring that keyword bindings be separated from the variable bindings, as one might do via header files in C (Kernighan and Ritchie 1988), would destroy this feature.

While useful, the export of both variable and keyword bindings complicates the evaluation model, because variable and keyword bindings are needed during different phases of evaluation: keyword bindings are needed when the importing code is compiled, while variable bindings are needed when the importing code is run. The situation is further complicated by the fact that macro transformers are themselves Scheme procedures that may need to reference bindings imported from other libraries *at compile time*. When these other bindings are themselves keywords, the bindings are needed when the compile-time code of the compile-time code is compiled, and so on.

The first public draft of R^{6}RS (Sperber et al. 2006) required programmers to declare the phases at which libraries are available for use. If a library is imported explicitly for *run* (the default), its keyword bindings are evaluated when the importing code is compiled, and its variable bindings are evaluated when the importing code is run. Similarly, if a library is imported explicitly for *expand*, both keyword and variable bindings are evaluated when the importing code is compiled. Generally, a library may be imported explicitly for a specific phase by specifying the *meta* level of its import, where *run* corresponds to meta-level 0 and *expand* corresponds to meta-level 1. For macros that expand into transformers, negative meta levels may also be needed. When the importing code is itself a library, the declarations and their semantics become even more complex.

These declarations can become unwieldy. The declarations are also imprecise in nature, as discussed in Section 4.4, which in turn leads to unnecessary compile-time overhead as bindings are evaluated in some cases when they are not actually needed.

A better alternative is to shift the burden of determining and declaring the phases at which each library's keyword and variable bindings must be evaluated from the user to the implementation. With this alternative, the implementation infers the phases at which the variable and keyword bindings of a library must be evaluated based on how the identifiers are actually used by the importing code. When a reference to a keyword imported from a library is encountered during the compilation of the importing code, the keyword bindings of that library are evaluated. Similarly, if the residual code after macro expansion contains a reference to an imported variable, the variable bindings of the imported library will be evaluated at the run time of the residual code. Both occur automatically whether the importing code is run-time code, compile-time code, or code used at some higher meta level.

During the formal comment period for the first public R^{6}RS draft, we developed such an implementation and found it indeed easier to use. It is also often significantly more efficient, since libraries are never loaded and initialized unnecessarily. The additional efficiency comes "only" at compile time, but we believe that compile time is important. Furthermore, compile time may coincide with run time through the use of Scheme's `eval` procedure or various run-time compilation techniques.

Having implemented and seen the benefits of implicit phasing, we lobbied the R^{6}RS editors to switch to implicit phasing. We got our way—partly. The subsequent draft (Sperber et al. 2007a) and each that followed allows implementations to support either implicit or explicit phasing. This means that programs to be run in an implementation that requires phase declarations must include them, but other programs can leave them out. A future version of the report, e.g., R^{7}RS, may mandate either implicit or explicit phasing once the community settles on a de facto standard, and we believe the relative simplicity and efficiency of implicit phasing will win out in the end.

This paper describes the implicit-phasing model and why we believe it to be superior to the explicit-phasing model. It also describes our implementation of R^{6}RS libraries, which consists of a *library manager* and a *library expander*. The library manager handles the dependencies among libraries, and the library expander handles the expansion of individual libraries. To allow for maximum portability, the library expander transforms a library into code in a small core language compatible with R^{5}RS Scheme. Both the library manager and library expander are themselves written as R^{6}RS libraries and thus expand into R^{5}RS compatible code. This makes R^{6}RS libraries readily available, now, and runnable on most existing R^{5}RS implementations. Additionally, the library expander produces straightforward code amenable to optimizations like copy propagation and procedure inlining. In particular, no run-time indirects are required to determine the values of either local or imported variable bindings. To date, we have tested the implementation under Chez Scheme, Chicken Scheme, Gambit, Gauche, Ikarus, Larceny, and MzScheme.

The remainder of this paper is organized as follows. Section 2 briefly describes the syntax and semantics of R^{6}RS libraries and macros. Section 3 describes how the library manager handles dependencies among libraries. Section 4 describes how the library expander transforms a single library, including its constituent definitions and expressions, into the core language. Section 4 also describes how the library expander triggers the evaluation of the keyword and variable bindings of other libraries as they are needed. Section 5 describes the three products of the library expander: information about bindings, code to evaluate keyword bindings, and code to evaluate variable bindings. Section 6 describes the core language of the code produced by the expander. Section 7 discusses the expansion algorithm. Section 8 discusses related work, while Section 9 presents our conclusions and possibilities for future research.

2. R^{6}RS Libraries and macros

A Scheme library, from a programmer's standpoint, is a set of private and exported definitions that provide functionality often related to a specific purpose. The R^{6}RS document defines a set of standard libraries that the programmer can extend using the `library` form. Bindings exported from one library can be imported into another library. We say that the imports define the language in which a library is written. A simple library of numeric procedures can be written as follows, where `---` represents elided code.

```
(library (numerics)
  (export fact ack fib)
  (import (r6rs))
  (define (fact n)
    (if (zero? n) 1 (* n (fact (- n 1)))))
  (define (ack n m) ---)
  (define (fib n) ---))
```

The `(numerics)` library imports all the identifiers that the `(r6rs)` library exports. From this set, `(numerics)` uses the keywords `define` and `if` and the variables `zero?`, `*`, and `-` among others. The `(numerics)` library can be imported into other libraries using the same syntax used for importing the `(r6rs)` library.

A library may need to initialize itself before any of the variables it defines are used. In the following example, a library of facts about Scheme is defined. The library uses the `(hash-tables)` library for quick access to the factoids. The hash table is populated when the library is initialized.

```
(library (scheme-factoids)
  (export fact)
  (import (r6rs) (hash-tables))
  (define ht (make-eqv-hash-table))
  (define (fact x)
    (hash-table-ref ht x #f))
  (hash-table-put! ht 120
    "Scheme macros are written in Scheme.")
  ---)
```

If two libraries export like-named identifiers that represent different keyword or variable bindings, they may not be imported together without qualification into a third library or top-level program. A set of import qualifiers are provided to allow programmers to work around this and, more generally, exercise finer control over the set of bindings imported from a library and the names used locally to refer to those bindings. An *import-set* consists of zero or more qualifiers wrapped around a library reference, which is itself a parenthesized sequence of names. These qualifiers are `only`, `except`, `rename`, and `prefix`:

> `(only` *`import-set identifier`* `...)` selects from the set of bindings selected by *import-set* only those named by the given *identifiers*.
>
> `(except` *`import-set identifier`* `...)` selects all of the bindings selected by *import=set* except those named by the given *identifiers*.
>
> `(rename` *`import-set`* `(`*`old new`*`) ...)` selects all of the bindings selected by *import-set*, using the local name *new* for each binding named by a corresponding *old*.
>
> `(prefix` *`import-set prefix`*`)` selects all of the bindings selected by *import-set*, prefixing the name of each binding with the given *prefix*.

An *import-set* consisting of an unqualified library reference selects all of the bindings exported by the named library. The qualifiers are illustrated by the following Scheme top-level program.

```
(import
  (only (r6rs) display)
  (rename (scheme-factoids) (fact scheme-fact))
  (prefix (except (numerics) ack fib) num:))
(display (scheme-fact (num:fact 5)))
```

In addition to extending the language by defining new variables, programmers can extend the syntax of the language by defining new keywords. Just as `define` binds variables to values (or locations holding values), `define-syntax` binds keywords to transformers. A transformer is a procedure that accepts as input a single value—a syntax object representing a macro call—and returns a new syntax object. The right-hand-side of a macro binding form can be any expression that evaluates to a transformer. Transformers can use the full Scheme language to implement their transformations, using `syntax-case` to match and destructure the input and `syntax` to construct the output (Dybvig 1992; Dybvig et al. 1992).

A transformer often simply rewrites the input in a straightforward manner, extracting portions of the input and inserting them into the output, as illustrated by the definition of when in terms of one-armed if below.

```
(define-syntax when
  (lambda (x)
    (syntax-case x ()
      [(when e0 e1 e2 ...)
       (syntax (if e0 (begin e1 e2 ...)))])))
```

The input to a transformer is always the entire syntactic form, including the identifying keyword, which is reflected by the pattern given in the when transformer's syntax-case expression. Pattern variables, such as e0, are used to name pieces of the input, with ellipses used to specify zero or more occurrences of the preceding pattern. These pattern variables are also used to insert the selected pieces of input into the output, as illustrated by the transformer's syntax template, using ellipses for pattern variables that represent zero or more input subforms.

The list appearing after the input in a syntax-case expression is a list of literal identifiers, used to recognize, e.g., the auxiliary keyword else in a cond or case expression. It is empty in the transformer expression above because when has no auxiliary keywords.

To avoid redundant specification of the keyword, an underscore, which matches any input, is often used in place of the keyword. Also, the form (syntax *x*) may be abbreviated to #'*x* just as (quote *x*) may be abbreviated to '*x*. So the definition above would typically be written more concisely as follows.

```
(define-syntax when
  (lambda (x)
    (syntax-case x ()
      [(_ e0 e1 e2 ...)
       #'(if e0 (begin e1 e2 ...))])))
```

For simple transformers like this, use of the syntax-rules form results in still shorter code.

```
(define-syntax when
  (syntax-rules ()
    [(_ e0 e1 e2 ...)
     (if e0 (begin e1 e2 ...))]))
```

The lambda expression and the syntax expression are implicit in syntax-rules, which conveniently reduces the amount of syntactic baggage when more generality is not required.

An important characteristic of each definition of when is that the identifier references inserted into the output of its transformer are scoped where the transformer code appears, not where the syntactic form to be transformed appears. So the keywords if and begin inserted into the output of the when transformer resolve to the if and begin bindings in effect where when is defined, regardless of any definition of if and begin in the context of a when expression.

A related characteristic is that bindings introduced by a transformer do not capture references in the input to the transformer. For example, given:

```
(define-syntax compose-self
  (syntax-rules ()
    [(_ p x) (let ([t p]) (t (t x)))]))
```

the expression

```
(let ([t 3])
  (compose-self (lambda (x) (+ x 1)) t))
```

expands into the equivalent of

```
(let ([t1 3])
  (let ([t2 (lambda (x) (+ x 1))])
    (t2 (t2 t1))))
```

so that the binding of t introduced by compose-self does not capture the reference to t in its input.

These characteristics are required by the *hygienic* nature of the macro expander (Kohlbecker et al. 1986) and essentially mean that syntax definitions, like procedure definitions, are lexically scoped. While lexical scoping is the default, transformers that violate lexical scoping, i.e., transformers that refer to bindings in the context of their input or that introduce bindings that capture references in their input, can also be written, but we have no space or need to describe such transformers here.

The following example illustrates a library that defines and exports the keyword cteval, whose transformer uses eval at expansion time to perform compile-time evaluation and also list to build up the result.

```
(library (compile-time-eval)
  (export cteval)
  (import (r6rs) (r6rs eval))
  (define-syntax cteval
    (lambda (x)
      (syntax-case x ()
        [(_ expression library)
         (list #'quote
           (eval (syntax->datum #'expression)
             (environment
               (syntax->datum #'library))))]))))
```

The defined syntax cteval (cteval (+ 1 2) (r6rs)) expands to (quote 3) by evaluating, at compile time, the expression (+ 1 2) in the language defined by the (r6rs) library, and quoting the resulting value.

While this degree of generality is seldom needed, the transformer for cteval illustrates how the full generality of Scheme can be employed by a macro.

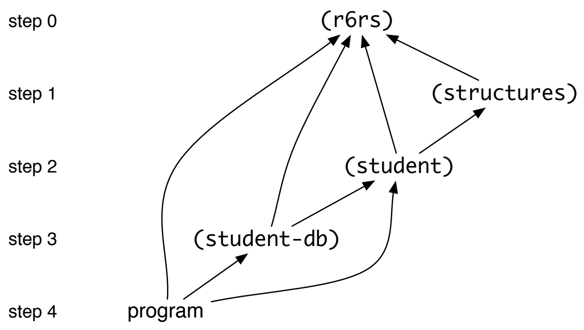
Any identifier imported from the (r6rs) library can be used both for run and expand, so even in the explicit declaration model, there is no need to declare the phases explicitly for the example above. Our implementation extends this to arbitrary libraries and phases. In other words, the import form of a library defines the language in which the library body is written, regardless of phases.

3. Library management

The libraries described in the preceding section each depend on other libraries, if only the built-in (r6rs) library. In general, each library is expanded before it is imported into dependent libraries. The import forms of a set of libraries implicitly define a dependency graph. The graph is necessarily acyclic because a library import relationships cannot be recursive. For example, suppose we have a top-level Scheme program that imports the student-db as follows:

```
(import (r6rs) (student) (student-db) ---)
(define print-student-record ---)
(new-repl
  (lambda () (display "Enter a student name: "))
  (lambda (x)
    (print-student-record (find-student x))))
```

The library manager determines from the import form of the top-level program that the program depends on (r6rs), (student), and (student-db). Each in turn depends on its own set of libraries. After following the (finite) chain of imports, the library manager determines a partial ordering of dependencies that is illustrated by the directed acyclic graph below.



Based on this partial order, expansion of this program must proceed in the steps identified in the graph, as elaborated below.

- Step 0 defines the `(r6rs)` library. Although the library is likely to be constructed from several other libraries, we treat this library as primitive, hence Step 0.
- Step 1 expands the `(structures)` library. This defines the `define-structure` macro that can be used in any library that imports `(structures)` at a later step.
- Step 2 expands the `(student)` library. In this step, the `student` structure is defined by constructing a unique identifier for the structure type along with a set of definitions and macros that provide the constructor, predicate, and accessor operations. Such procedures and macros (with the embedded structure identifier) are available for later imports. The student structure identifier does not change across invocations.
- Step 3 expands the `(student-db)` library. The expanded body of the library contains instances of student structures (generated at compile time) and a procedure `find-student`, for accessing the database.
- Step 4 expands the top-level program, using the outputs of previous stages.

3.1 Library visiting and invocation

In cooperation with the library expander, the library manager must arrange for the keyword and variable definitions of a library to be evaluated at the appropriate times. The evaluation of keyword definitions is referred to as *visiting* a library, while the evaluation of variable definitions and library initialization expressions is referred to as *invoking* a library. A library must be visited when its keyword bindings are required, directly or indirectly, for the expansion of another library or top-level program, and it must be invoked when its variable bindings are required for expansion of another library or top-level program or for a run of a top-level program. The process of visiting and invoking a library in our implementation is described in detail in Section 5.

3.2 Inconsistency of multiple expansions

Although not specifically required by the draft R^{6}RS, it turns out that every library must be expanded exactly once to avoid inconsistencies that may appear in the set of identifiers exported by a library. Using separately expanded versions of one library may yield to inconsistent and "unlinkable" programs. A simple example showing the possibility of producing such programs is shown below.

```
(library (E0)
  (export x)
  (import (r6rs))
  (define-syntax def
    (lambda (x)
      (define inline-constants?
        (cdr (assq 'inline-constants?
               (with-input-from-file "config.ss"
                 read))))
      (syntax-case x ()
        [(_ name val)
         (if inline-constants?
             #'(define-syntax name
                 (identifier-syntax val))
             #'(define name val))])))
  (def x 17))

(library (E1)
  (export f) (import (r6rs) (E0))
  (define f (lambda () x)))
```

If, during the expansion of `(E1)`, `(E0)` is expanded and the configuration file says not to inline constants, then the reference to `x` in `(E1)` will residualize to a reference to the variable `x` exported by `(E0)`. If linked with a version of `(E0)` that is expanded again, this time with inlining of constants enabled, the variable `x` will not exist at run time.

3.3 Exploiting single expansion

This single-expansion guarantee can be exploited by the programmer when a compile-time constant is needed. This is useful, for example, for defining structures[1] that are guaranteed to have the same type across library invocations, as illustrated in the following example.

```
(library (structures)
  (export define-structure)
  (import (r6rs) (guid))
  (define-syntax define-structure
    (lambda (x)
      (syntax-case x ()
        [(_ name (fields ...))
         (with-syntax ([id (generate-id)]
                       [maker ---]
                       [pred? ---]
                       ---)
           #'(begin
               (define (maker fields ...)
                 (vector 'id fields ...))
               (define (pred? x)
                 (and (vector? x)
                      (= (vector-length x)
                         (+ 1 (length '(fields ...))))
                      (eq? (vector-ref x 0) 'id)))
               ---))]))))
```

Without the single-expansion guarantee, the programmer would be forced to generate the unique identifier manually and to hard-code it at every structure definition.

The guarantee can be exploited further by the ability of the programmer to generate, at compile time, structures with the correct run time values. For example, suppose the library `(student)` defines a student structure as follows.

```
(library (student)
  (export make-student student? student-name ---)
  (import (structures))
  (define-structure student (name id major))
  ---)
```

[1] A structure is represented as a vector with a unique tag.

A user of the library can define at compile time a database of students which results in translating the entire database into a single quoted constant. Moreover, the call to `student-name` won't fail because of data mismatch in the structure identifier of compile-time generated structures.

```
(library (student-db)
  (export find-student)
  (import (r6rs) (student))
  (define-syntax define-students
    (lambda (x)
      (syntax-case x ()
        [(_ db-name [name id major] ...)
         (let ([students
                (apply map make-student
                  (syntax->datum
                    #'((name ...)
                       (id ...)
                       (major ...))))])
           #`(define db-name '#,students))])))
  (define-students student-db
    ["jane" "111-11-1111" journalism]
    ["john" "222-22-2222" psychology]
    ---)
  (define (find-student x)
    (memp
      (lambda (s) (equal? (student-name s) x))
      student-db)))
```

The single-expansion guarantee also allows a programmer to record information that was current as of the time of expansion. For example, the following library, `(compile-time)`, exports a single macro, `ctime`, that expands to a string representing the time the macro is called, supposing that the implementation provides a `(date/time)` library that exports a `time->string` procedure.

```
(library (compile-time)
  (export ctime)
  (import (r6rs) (date/time))
  (define-syntax ctime
    (lambda (_) (time->string (now)))))
```

Using `(compile-time)`, a Scheme top-level program can print a greeting message showing the time the system was compiled, supposing that the implementation provides a `(formatted-output)` library that exports a `printf` procedure.

```
(import (compile-time) (formatted-output))
(printf "This program was compiled on ~a\n"
  (ctime))
```

The entire program expands to the following code:

```
(printf "This is program was compiled on ~a\n"
  "2007/04/02 23:17:04")
```

We can define a library `(F)` that exports a single procedure `F-compile-time`. The procedure returns a string representing the time on which the library `(F)` was compiled.

```
(library (F)
  (export F-compile-time)
  (import (r6rs) (compile-time))
  (define (F-compile-time) (ctime)))
```

Intuitively, expanding `(F)` results in code that binds the variable `F-compile-time` with a procedure that returns a constant string:

```
(define F-compile-time
  (lambda () "2007/04/02 23:29:54"))
```

Consequently, any call to the procedure `F-compile-time` would return the same string. This is true regardless of the time or place in which the procedure `F-compile-time` is called. For example, the program

```
(import (r6rs) (F))
(let-syntax ([t (lambda (_) (F-compile-time))])
  (string=? (F-compile-time) (t)))
```

calls the `F-compile-time` once when it is expanded, and once again when resulting code is evaluated. The program expands to the following run time code which returns `#t` when evaluated.

```
(string=? (F-compile-time) "2007/04/02 23:29:54")
```

4. Phased expansion model

In Section 3, we discussed the order in which libraries must be expanded, based on the dependency graph implicit in the `import` forms. In this section, we discuss how the library body itself is expanded. To expand a library `L`, we assume that all imported libraries have already been expanded to core forms. We start with a discussion of local identifiers, then proceed to a discussion of imported identifiers.

4.1 Phase of local identifiers

We say that the *phase* of an identifier binding is the time at which the value of the binding is computed. The simplest phase is phase 0, which is the run phase in a Scheme top-level program. The right-hand-side expression of a top-level `define-syntax` form is a phase 1 expression. In general, if a macro definition appears in phase n code, then its right-hand-side expression is in phase $n+1$. Although there is no limit on the number of phases that an expression can have, the residual code of a phase n expression contains only phase n bindings.

An expression in phase 0 can access any phase 0 identifier that is in the lexical scope of the expression. In the following example, the variables `x`, `y`, and `f` are all phase 0 bindings because their values are determined at run time.

```
(let ([x 5])
  (define f (lambda (y) (+ y x)))
  (display (* x (f 3))))
```

Because an expression must be expanded before it is evaluated, transformers, which are evaluated at expression expansion time, cannot access the values of variables that are computed at run time. For example, the following code cannot be compiled because a reference to the run time variable `x` is referenced at expansion time, when the `lambda` expression is evaluated.

```
(let ([x 5])
  (define-syntax f (lambda (y) (+ x y)))
  (display (* x (f 3))))
```

We call the binding of `y` a phase 1 binding since it appears in the right-hand-expression of a syntax binding form. Our syntax system rejects such an expression because it does not make sense in the R^{6}RS language, which requires that expressions be fully expanded before they are evaluated. The program is rejected during the expansion of the transformer expression at the point where the reference to `x` is found in `(+ x y)`.

A similar restriction is enforced when a phase 1 binding is referenced in phase 0 code. In the following example, `(f 3)` expands to a reference to the compile-time variable.

```
(let ([x 5])
  (define-syntax f (lambda (y) #'y))
  (display (* x (f 3))))
```

This program is rejected because the phase 0 expression `(f 3)` expands to a reference to a phase 1 variable `y`. This program is rejected when the output of the call to `(f 3)` is re-expanded and the reference to `y` is discovered.

In general, our syntax system rejects any attempt to residualize a reference to a phase n local variable in a phase m context when $n \neq m$. It also rejects attempts to residualize a reference to a variable outside of its lexical boundaries. Only by using poorly styled code that uses expansion-time side effects can such anomalous cases be triggered.

4.2 Phases of imported identifiers

In contrast with local identifiers, imported identifiers represent code that has already been expanded, so there is no need to restrict their usage to specific phases. The R^{6}RS allows programmers to declare the phases in which an imported library can be used. With the explicit-phase specification, one would be required to specify the exact `run`, `expand`, `(meta 2)`, `(meta 3)`, etc., for every phase in which a library is used. This is done via the `for` syntax, which is a wrapper for the import sets described in Section 2: `(for` *`import-set phase`* `...)` restricts the identifiers specified by *import-set* to use only at the specified phases.

For example, one would have to annotate the top-level program from Section 3.3 as follows.

```
(import (r6rs) (for (F) run expand))
(let-syntax ([t (lambda (_) (F-compile-time))])
  (string=? (F-compile-time) (t)))
```

So, even though the two references to the `F-compile-time` variable refer to the same binding that is exported from `(F)`, an implementation of R^{6}RS may reject the original program because the programmer did not specify at which phases the references to the imported libraries are valid.

The two references to `F-compile-time` in the code above cannot reference two distinct bindings of the same name because the `import` form requires that all imported identifiers be unique. There is no ambiguity in deciding where the `F-compile-time` identifier came from, just as it is clear where the `lambda` identifier came from. The draft R^{6}RS specifies that the `(r6rs)` library exports its identifiers to both `run` and `expand` phases in order to make it "convenient for users who do not care about phases." We extend the convenience to all libraries and all phases.

In the first revision of the R^{6}RS document, R$^{5.91}$RS, explicit phase specifications were required and implementations were required to reject any library if its body was inconsistent with the declared phases of the imports. Soon after, two reference implementations of the proposed R^{6}RS library semantics were provided, ours and one by André van Tonder (Dybvig et al. 2006). Both implementors realized that one of the report's specified macros, `identifier-syntax`, was not implementable in the R^{6}RS library system. Each proposed a different fix. The suggestion of the implicit phasing camp lead to allowing implementations to infer the phases in which libraries are used while the library is being expanded. The suggestion of the explicit phasing camp lead to the inclusion of *negative meta levels*.

To see why this is needed, consider a library that exports macro helpers (i.e., procedures and macros that are normally used at macro-expansion time). For example, the library `(Q)` below exports the *procedure* `quote-5`, which returns a syntax object representing the quoted number 5.

```
(library (Q)
  (export quote-5)
  (import (r6rs))
  (define (quote-5) #'(quote 5)))
```

The library `(Q)` can be used at compile time in another library as follows:

```
(library (R)
  (export number-5)
  (import (r6rs) (Q))
  (define number-5
    (let-syntax ([m (lambda (x) (quote-5))])
      (m))))
```

In the explicit phase specification model, the programmer must specify that the library `(Q)` should be imported in `(R)` for `expand`, or `(meta 1)`, because it is used at library expansion time. Now because `(Q)` is imported for phase 1, the `quote` it produces is a phase 1 `quote` which cannot be inserted into the output of the macro `m`. Therefore, the `(r6rs)` library must be imported into `(Q)` for both `run` and `(meta -1)`. Therefore, the import of `(Q)` for `(meta 1)` combined with the import of `(r6rs)` for `(meta -1)` result in a `(meta 0)` `quote` that is inserted in the `run` code of `(R)`. Thus, we end up with the following, assuming the programmer does not mistakenly add unnecessary phase specifiers while trying to get the program to compile.

```
(library (Q)
  (export quote-5)
  (import (for (r6rs) run (meta -1)))
  (define (quote-5) #'(quote 5)))

(library (R)
  (export number-5)
  (import (r6rs) (for (Q) expand))
  (define number-5
    (let-syntax ([m (lambda (x) (quote-5))])
      (m))))
```

In both models, the `import` form defines the meaning of the imported identifiers. For example, one cannot import two identifiers named `quote` for two phases and have them mean two different things. If `quote` is imported from `(r6rs)`, then it is the same R^{6}RS `quote` at all phases. Explicit phasing merely restricts the phases in which the identifiers can be referenced.

In the implicit phase specification model, reference to the variable `quote-5` in compile-time code makes `(Q)` implicitly imported for that phase. Since the R^{6}RS-`quote` that `quote-5` produces is placed in the run-time code of the output of the `m` macro, the `quote` is placed in phase 0 automatically.

A question arises: Does the set of libraries that can be written using the implicit phase model differ from the set that can be explicitly specified? If the two sets are the same, then maybe we can write a program that statically derives the exact set of phases that may be required. In fact, however, the set of programs whose phases can be inferred is larger than the set that can be explicitly specified. This is due to the fact that the set of phases that the library can specify is fixed. The following contrived macro illustrates this problem by expanding to references to itself in as many phases deep as there are sub-expressions in its use site.

```
(define-syntax let-syntax***
  (syntax-rules ()
    [(_ () body) body]
    [(_ ((i* e*) ... (i e)) body)
     (let-syntax ([i (let-syntax*** ((i* e*) ...)
                       e)])
       body)]))
```

While we do not know if this problem will ever arise in practice, it does point out a limitation of the explicit phasing model.

4.3 Shared vs. separate bindings across phases

In the preceding section, we discussed two models for determining the phases in which an imported binding can be used. The explicit phasing model requires the programmer to restrict the availability of an identifier to a fixed set of phases, while the implicit phasing model makes all identifiers available at all phases.

Regardless of how the phases of the imported bindings are determined, the implementation must decide when libraries must be invoked. At a minimum, we know that if library X exports variable x that's referenced in the run time code of library Y, then X must be invoked before Y is invoked. Failure to do so would result in an invalid reference. Similarly, if v is referenced in compile-time code of library Z, then X must also be invoked at Z's compile time, and before any transformer code that might reference x is evaluated.

Our implementation invokes a library, minimally, only when it is needed, using a simple demand-driven approach. When the need arises to invoke the library, it is invoked, and not before. Once invoked, it is never invoked again in the same Scheme process. The library can actually be in one of three states: *uninvoked*, *invoking*, and *invoked*. It is set in the *invoking* state just before it is initialized so that the system can detect attempts at circular invocations, which can occur only via improper use of `eval`.

A similar mechanism is used to visit libraries on demand.

While the semantics described above is compatible with the R^{6}RS requirements, the report also allows other semantics in which separate instances of the same library exist for every phase in which the library is imported and even for the expansion or evaluation of different programs by the same Scheme process. Implementations that create separate instances in this manner must generally visit and/or invoke each library multiple times.

One reason why we chose to have no more than one instance is that repeatedly evaluating transformer and/or variable bindings and initialization code may be expensive, both in time and space. While this affects primarily compile time, compile-time costs should not be ignored, and compilation can happen at run time when Scheme's `eval` procedure is used.

Another reason why we chose to have no more than one instance of a library is that system programming requires the ability to manage resources, some of which cannot be replicated. An example of such a resource is the Scheme symbol table. If the symbol table is defined in a Scheme library, and a separate instance of that library is created for every phase, then calling the procedure `string->symbol` on the same string at two different phases would incorrectly yield two non-`eq?` symbols[2]. Management of non-generative records, pseudo random number generators, exception handlers, external devices, etc., would be problematic if separate instances of these resources exist independently.

4.4 Inaccuracies of phase requirements

Another drawback of explicit specification is that programmers are sometimes forced to overestimate import requirements in order to achieve the minimal requirements they need.

We illustrate why the explicit phase specification overestimates the requirements by first questioning what one might mean when one declares `(import (for (Q) expand))` in library `(R)`.

On the one hand, it is possible that `(R)` references one of `(Q)`'s exports in a *local macro* as shown in page 6. After `(R)` is expanded, its residual code contains `(define number-5 '5)` and, therefore, `(Q)`'s exports would not be used after `(R)`'s expansion. On the other hand, it may be that `(R)` imported `(Q)` for `expand` because `(R)` exports a macro that expands to a reference to `(Q)`'s exports in compile-time code. In this case, expanding `(R)` does not reference any of `(Q)`'s variables, but expanding the importer of `(R)` does.

An explicitly phased implementation would invoke `(Q)` once when `(R)` is expanded, and invoke it again when `(R)` is imported for expanding a third library. This is guaranteed to do more work than necessary if the bindings are not required in both situations, again increasing compile-time costs.

4.5 Benefits of explicit phasing?

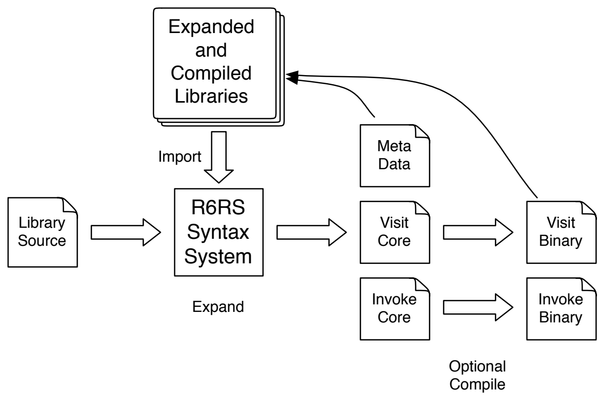
Given that the explicit phasing model requires more work on the part of the programmer, does not allow the programmer to specify phases precisely, generally incurs more compile-time overhead, and limits the set of programs that can be written, it is fair to ask whether it has compensating benefits. We believe it does not.

The only benefit of explicit phasing to the programmer is that it allows the programmer to express his or her understanding of a sufficient (but, unfortunately, often not necessary) set of phases at which the bindings of a library are required and to have this understanding tested by the expander. This might be useful if the user could use this information to determine when the side effects of initializing a library occur, but the implementation has broad latitude even under the explicit phasing model to initialize a library at the time or times of its choosing. Thus, side effects must generally be of the kind that affect only the library's own bindings, not the kind that are visible externally. Given this, explicit phasing seems, for the user at least, not to be a worthwhile exercise, and it is likely to be more frustrating than illuminating.

Furthermore, we have found that the explicit phasing model is at least as difficult to implement. With both models, the implementation must determine the phases at which an attempt to use each identifier is made. In the explicit phasing model, the implementation must also record and check phase restrictions and report them to the programmer. The only compensation is that the implementation can eagerly initialize a library based on the declared phases, obviating the need to do so on demand.

5. Components of an expanded library

When our expander processes a library, it reconstructs the library into a "core-library" form composed of three components organized by usage. The three components are the library meta data (describing the library's products and dependencies), the library *visit-time* code (used to evaluate transformer definitions), and the library *invoke-time* code (used to evaluate variable definitions and initialization expressions). Each component is described below.



[2] There would actually be multiple procedures named `string->symbol`, one per phase, each with its own copy of the symbol table.

5.1 Library meta data

When a library is expanded, the following meta data is recorded:

1. The library name and version. This information is used by the expander when it processes imports in order to determine whether the library satisfies the version requirements that the importing library specifies.
2. A library identifier. Every time a library is expanded, a globally unique identifier is given to the expanded instance in order to distinguish it from other expanded instances of the same library. This is used to detect violations of the single-expansion invariant that was discussed in Section 3, which might occur through improperly managed separate compilation.
3. Identifiers of imported libraries. This list of libraries is used to resolve re-exported identifiers.
4. Library substitution. This is an association list that maps the names of the set of exported identifiers to a set of unique *labels*. When the library is used to expand another library, some of the names may be removed, renamed, or prefixed with an identifier depending on the import modifier (`only`, `except`, `rename`, and `prefix`).
5. Keyword locations. This is a mapping from labels to locations where every location denotes a global macro binding that is defined in the library. Re-exported keywords are looked up through the chain of imports (item 3). The values of the transformers are obtained by evaluating the visit-time code (5.2).
6. Variable locations. This is a mapping from labels to locations where every location denotes a global variable binding that is defined in the library. Re-exported variables are looked up through the chain of imports (item 3). The locations are initialized by evaluating the invoke-time code (5.3).
7. Visit requirements: This is a list of library identifiers that must be invoked before the visit-time code is evaluated.
8. Invoke requirements: This is a list of library identifiers that must be invoked before the invoke-time code is evaluated.

5.2 Visit-time code

When a library is being expanded and a reference to an imported macro binding is found, the visit-time code of the exporting library must be evaluated to obtain a list of transformers (one for each exported macro definition). The list of labels and the list of transformers are joined to obtain the mapping. The list is cached and the visit code is never evaluated again. The visit-time requirements of the library must be invoked before evaluating the visit-time code in order to initialize any variables that may be referenced in the code.

5.3 Invoke-time code

The invoke-time code is evaluated in order to initialize the locations that a library defines. This code may be evaluated at compile time, before any code that references one of the library variables is evaluated. The code may also be invoked at run time if any of the library variables may be referenced at run time. The invoke-time code is evaluated at most once.

Both the visit-time code and invoke-time code is pre-expanded and is, therefore, composed of core Scheme expressions. The code may be run interpreted, may be compiled on the fly, or batch compiled to native form. Because the code does not contain any syntactic extensions, it is easy to cross-compile the code to different architectures. The same expanded code serves as a basis for many source-compatible binary instances of the library. This is also the main reason why we separate the library meta data, its visit code, and its invoke code. The visit-time code is needed only for compiling other libraries and need not be shipped in the final application if the application performs no run-time compilation.

6. Target language

The target language of a library expander is not specified by the report. Our system takes advantage of this lack of specification to target core language that can be evaluated by any complete implementation of R^{5}RS. This section describes the core language that the expander targets, how R^{6}RS code is evaluated in an R^{5}RS system, and how top-level locations are constructed.

6.1 Core forms

The expander transforms code from R^{6}RS library syntax to a core Scheme form. A core Scheme expression can be defined according to the following grammar.

```
⟨Expr⟩    → (quote ⟨datum⟩)
          → ⟨Primitive⟩              (e.g., cons, +, vector?)
          → ⟨Variable⟩
          → (if ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩)
          → (set! ⟨Variable⟩ ⟨Expr⟩)
          → (begin ⟨Expr⟩ ⟨Expr⟩ ...)
          → (letrec ([⟨Variable⟩ ⟨Expr⟩] ...) ⟨Expr⟩)
          → (⟨Expr⟩ ⟨Expr⟩ ...)
⟨Formals⟩ → () | ⟨Variable⟩ | (⟨Variable⟩ . ⟨Formals⟩)
```

This choice of core forms makes the core language language reasonably simple while avoiding loss of information during expansion. For example, `let` is not included among the core forms because expanding `let` to an application of a direct procedure does not lose any information, and we expect that optimizing compilers know how to treat them efficiently. On the other hand, `letrec` is included because expanding `letrec` to the equivalent set of bindings and assignments loses information, and would therefore inhibit certain optimizations (Waddell et al. 2005).

Two additional expression forms are also included in the core by default:

```
⟨Expr⟩ → (case-lambda [⟨Formals⟩ ⟨Expr⟩] ...)
       → (letrec* ([⟨Variable⟩ ⟨Expr⟩] ...) ⟨Expr⟩)
```

The first is a generalization of `lambda` to multiple formal parameter lists, each with a corresponding body, and is used to support the source-language `case-lambda` form, which is included in one of the standard R^{6}RS libraries. For an implementation that does not support `case-lambda` natively, the expander can be configured to produce `lambda` expressions instead, using a dynamic dispatch on the length number of arguments when other than one clause is appears. We expect that implementations that will target R^{6}RS will eventually provide native support for the `case-lambda` form.

The other, `letrec*`, is a variant of `letrec` that evaluates its bindings from left to right. This is useful for handling source-language `letrec*` expressions, library bodies, and `lambda` bodies. For an implementation that does not support `letrec*` natively, the expander can be configured to produce semantically equivalent set of bindings and assignments.

If desired, in fact, implementors can configure the expander to produce code in an even smaller core language, such as the following.

```
⟨Expr⟩  → (quote ⟨datum⟩)
        → ⟨Primitive⟩
        → ⟨Variable⟩
        → (if ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩)
        → (set! ⟨Variable⟩ ⟨Expr⟩)
        → (lambda ⟨Formals⟩ ⟨Expr⟩)
        → (⟨Expr⟩ ⟨Expr⟩ ...)
```

Implementors can also customize the expander to recognize core forms that are not included in the set above. This feature can be used to define implementation-dependent core-language syntax, e.g., for creating foreign functions. Such core forms would presumably appear in the expanded code only when non-standard libraries provided by the implementation are imported by a program's source code.

As evident in the core-language grammar, `lambda` bodies consist of a single expression, with no definitions, as do the bodies of `letrec` and `letrec*`. In addition, the names used for the local and global bindings produced by the expander are disjoint each from each other and also from the set of core-language keywords and primitive names. An implementor may (but need not) take advantage of these restrictions to simplify the parsing of core-language expressions when compiling or evaluating the output of the expander.

6.2 Core primitives

Our system is an expander that transforms R^6RS libraries to the host implementation's core forms. The system does not itself define any R^6RS primitives that are not related to the syntax system. Syntax-related procedures that are provided by the system include `syntax->datum`, `datum->syntax`, `free-identifier=?`, etc., in addition to the R^6RS `eval` and `environment` procedures. Implementations that target R^6RS would have to provide definitions of the other R^6RS primitives.

The code of the expander itself uses a subset of the primitives that is either shared between R^5RS and R^6RS or is portably available in all implementations.

6.3 Evaluation of core expressions

The existence of expressions that must be evaluated at compile time implies that the expander must perform some evaluation while a library is being expanded. We identify three places where the expander requires evaluation:

1. When the expander encounters a syntax binding form (such as `define-syntax`, `let-syntax`, etc.), it expands and evaluates the transformer expressions in order to obtain the transformer procedure.
2. When the expander encounters a reference to a macro that is defined in another library, the visit-code of the exporting library must be evaluated in order to obtain the values of the transformers.
3. When the expander encounters a reference to a variable that is exported from another library in compile-time code (i.e., phase n for $n > 0$), the invoke-code of the exporting library must be evaluated in order to initialize the exported locations.

The final output of the expander may also be evaluated, of course, but that evaluation occurs after expansion has been completed and is outside of the control of the expander.

6.4 Top level locations

Library variables are given global locations in the R^5RS top-level namespace. Every time a library is expanded, each of its variables is given a unique top-level location. Direct access to the underlying locations is restricted at the source level; the only way to access a top-level location is through a library that explicitly defines or imports it. Therefore, all library locations are shared in one flat namespace, and access to these locations is managed by the library expander.

When evaluating previously expanded or compiled libraries, the expander loads the appropriate files using the R^5RS `load` procedure. Every such file is structured as a series of top-level variable definitions followed by an expression that initializes all such variables (using `set!`). We generate the top-level `define` forms for all globally defined variables because some R^5RS implementations (legitimately) reject assignments to "undefined" variables. While most systems' `eval` procedures accept definitions, R^5RS does not require `eval` to handle definitions, so we use `load` rather than `eval` to avoid portability problems.

On the other hand, the evaluation of a transformer expression never creates new top-level locations, so R^5RS's `eval` usually suffices for evaluating transformer expressions. Transformers may reference top-level bindings, however, and so need access to the environment into which library bindings have been defined. In most implementations, this access is provided by calling `eval` with just one argument, the expanded transformer expression, or with two arguments, the expanded transformer expression and the environment returned by `(interaction-environment)`. If neither mechanism is supported by an implementation, some other, implementation-dependent mechanism, must be used to evaluate transformer expressions.

In order to avoid name clashes between different library identifiers, globally unique names must be generated for every global identifier. Implementations that provide a read/write invariant `gensym` can utilize that extension for generating the names. The portable implementation generates sequentially different names, with a sequence id that is incremented across invocations of the syntax system. The name generation routine can be made more robust by customizing it to the specific implementation. For example, an implementation can use a procedure specific to the operating-system to generate globally unique identifiers.

7. Expansion algorithm

The expansion algorithm used by our expander to process the definitions and expressions contained within a library is essentially the same as Waddell's `syntax-case` expansion algorithm (Waddell and Dybvig 1999; Waddell 1999). It differs only in the handling of references to identifiers that are determined not to be lexically bound. Waddell's algorithm looks for such bindings in the imports of the enclosing top-level module, if any, and if not found there, then in the top-level (interaction) environment. Our expander looks for such bindings only in the imports of the current library or top-level program, since the imports specify the entire environment of the code within the library or top-level program. Also, when an imported binding is found, the expander may trigger, through the library manager, the visiting or invocation of the exporting library.

8. Related work

8.1 Chez Scheme modules

Chez Scheme's support for libraries can be roughly divided among the `module`, `import`, and `eval-when` forms, and various file-level procedures (e.g., `compile-file`, `include`, `load`, `visit`, `revisit`). Modules control the visibility of bindings, but the user is required to manually load, visit, or invoke (revisit) code in order to make the modules available when needed. This is facilitated by the `eval-when` form, which allows the programmer to control manually when files are loaded. In most other respects, Chez Scheme's top-level module forms are similar to the R^6RS library form.

Chez Scheme's module system is freely available in a portable implementation. The system has been successful and used by many implementations including Chicken Scheme, Gambit, Ikarus, SISC in addition to Chez Scheme and Petite Chez Scheme.

We used the portable `syntax-case` implementation as the basis of our implementation, so the architecture of our system is similar. The major difference between the two systems is in the level of details that the user must specify. In our system, a library `import` form specifies the environment in which the library is defined and expanding the body of the library determines when various pieces must be made available. In Chez Scheme, such details must be handled explicitly by the user.

The syntax of libraries in our system differs from Chez Scheme's modules in the sense that libraries in our system are stand-alone entities while Chez Scheme's modules are syntactic forms that can be produced through macro expansion. And while modules in Chez Scheme are defined in an external environment that's determined by where and when the module is compiled and loaded, our library system has no external environment.

8.2 MzScheme modules

The R^{6}RS library system is similar to the MzScheme module system (Flatt 2002). Like an MzScheme module, a library must specify its imports explicitly, so the language in which the library's code is written does not rely on the ambient environment of the expander or compiler.

In contrast to our implementation, the expander in MzScheme instantiates a separate instance of every library for every phase in which it is imported; the expander enforces that uses of an imported library match the declared phases in which said library is imported. Additionally, MzScheme's module system guarantees the invocation of imported libraries at the declared phases regardless of whether the bindings of the module are used or not.

The motivation for this decision was mainly the desire of the MzScheme implementors to implement "compile-time registry" libraries into which different modules would intern compile-time information during macro expansion time. The problem one faces in this setting is that information interned while one library is compiled will not persist when the library is loaded at subsequent stages. To guarantee proper registration, compile-time code that side-effects the compile-time environment has to be invoked again when the library is visited.

8.3 Chicken eggs

Chicken Scheme's *eggs* (Winkelmann 2007) provide a distribution mechanism for Scheme code that differs from the other systems described here in that Chicken does not allow the mixing of hygienic macro definitions and procedure definitions in the same library. Instead, macros are defined separately and given global scope. Also, the programmer must write an "egg-description" file which lists the exact dependencies, exports, and macros of the library while the R^{6}RS library system is designed to allow that information to be extracted from the library itself.

9. Conclusion

The `import` form of an R^{6}RS library specifies the language in which the code contained within the library is written. With the implicit-phasing model described in this paper, the `import` form need not also specify when the bindings imported into the library are available. Instead, the implementation determines from the actual use of each imported library's identifiers at which phases the library's keyword and variable bindings are needed. Thus, the programmer is responsible for saying only *what* is needed, while the implementation is responsible for determining *when*.

This paper also describes a portable implementation of R^{6}RS libraries that supports the implicit phasing model. It expands R^{6}RS top-level programs and libraries into code in a core language that is a small subset of the R^{5}RS language supported by most existing Scheme systems, which should facilitate rapid adoption of the library system and the remainder of the R^{6}RS language. The system is compatible with a variety of evaluation models, including interpreters, incremental compilers, and batch compilers. It has already been ported to several popular Scheme systems: Chez Scheme, Chicken Scheme, Gambit, Gauche, Ikarus, Larceny, and MzScheme.

We have chosen in the implementation to evaluate the transformer bindings and variable bindings of a library at most once per compilation session. We have made this choice to allow sharing of system-managed resources like symbol tables and random-number generators across phase boundaries and also to reduce the cost of compiling programs that use many libraries, some of which may require nontrivial initialization, at multiple phases. It would be straightforward, however, to modify our implementation to maintain separate environments for each phase and to evaluate transformer and variable bindings once per phase. In particular, the implicit phasing model does not inhibit doing so in any way.

Since it is based on the portable `syntax-case` expander extracted from the Chez Scheme source code, our system supports a few useful extensions featured by that expander, including local modules, local imports, and meta definitions (Dybvig 2005). We plan also to implement local libraries, which differ from local modules in that a nested library's exports may be used in any phase because they cannot depend on the environment in which the library appears. Local libraries are useful because they are encapsulated within the enclosing top-level library (and thus invisible outside) and can be produced via macros.

Acknowledgments

Will Byrd, Ron Garcia, Michael Adams, and the anonymous reviewers all made suggestions that led to substantial improvements in the final version of this paper.

References

R. Kent Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

R. Kent Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical Report TR 356, Indiana University, 1992. URL `citeseer.ist.psu.edu/dybvig92writing.html`.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992. URL `citeseer.ist.psu.edu/article/dybvig93syntactic.html`.

R. Kent Dybvig, Abdulaziz Ghuloum, and André van Tonder. R5.91rs library and syntax-case reference implementations, 2006. URL `http://www.r6rs.org/refimpl/r6rs-syntax-case.tar.gz`.

Matthew Flatt. Composable and compilable macros: You want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83, 2002. URL `http://doi.acm.org/10.1145/581478.581486`.

Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998. URL `citeseer.ist.psu.edu/kelsey98revised.html`.

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-200-4. doi: http://doi.acm.org/10.1145/319838.319859.

Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, and Jonathan Rees. Revised (5.91) report on the algorithmic language Scheme, September 2006. URL `http://www.r6rs.org/versions/r6rs_91.pdf`. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. B. Findler, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, J. Matthews, D.

Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised (5.92) report on the algorithmic language Scheme, January 2007a. URL `http://www.r6rs.org/versions/r6rs_92.pdf`. With R. Kelsey, J. Rees, R. B. Findler, and J. Matthews.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised (5.97) report on the algorithmic language Scheme, June 2007b. URL `http://www.r6rs.org/versions/r5.97rs.pdf`. With R. Kelsey, W. Clinger, J. Rees, R. B. Findler, and J. Matthews.

Oscar Waddell. *Extending the Scope of Syntactic Abstraction*. PhD thesis, Indiana University Computer Science Department, August 1999. URL `http://cs.indiana.edu/~owaddell/papers/thesis.ps.gz`.

Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 203–213, New York, NY, 1999. URL `citeseer.ist.psu.edu/waddell99extending.html`.

Oscar Waddell, Dipanwita Sarkar, and R. Kent Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher Order Symbol. Comput.*, 18(3-4):299–326, 2005. ISSN 1388-3690. doi: http://dx.doi.org/10.1007/s10990-005-4878-3.

Felix Winkelmann. *CHICKEN User's Manual*, 2007.