Enabling cross-library optimization and compile-time error checking in the presence of procedural macros

Andrew W. Keep  R. Kent Dybvig
Indiana University
{akeep,dyb}@cs.indiana.edu

Abstract
Libraries and top-level programs are the basic units of portable code in the language defined by the Revised\textsuperscript{6} Report on Scheme. As such, they are naturally treated as compilation units, with source compilation and certain forms of compile-time error checking occurring within but not across library and program boundaries. This paper describes a library-group form that can be used to turn a group of libraries and optionally a top-level program into a single compilation unit, allowing whole programs to be constructed from groups of independent pieces and enabling cross-library optimization and compile-time error checking. The paper also describes the implementation, which is challenging partly because of the need to support the use of one library’s run-time exports when another library in the same group is compiled. The implementation does so without expanding any library in the group more than once, since doing so is expensive in some cases and, more importantly, semantically unsound in general. While described in the context of Scheme, the techniques presented in this paper are applicable to any language that supports both procedural macros and libraries, and might be adaptable to dependently typed languages or template meta-programming languages that provide full compile-time access to the source language.

1. Introduction
A major difference between the language defined by the Revised\textsuperscript{6} Report on Scheme (R6RS) and earlier dialects is the structuring of the language into a set of standard libraries and the provision for programmers to define new libraries of their own [31]. New libraries are defined via a library form that explicitly names its imports and exports. No identifier is visible within a library unless explicitly imported into or defined within the library, so each library essentially has a closed scope that, in particular, does not depend on an ever-changing top-level environment as in earlier Scheme dialects. Furthermore, the exports of a library are immutable, both in the exporting and importing libraries. The compiler (and programmer) can thus be certain that if cdr is imported from the standard base library, it really is cdr and not a variable whose value might change at run time. This is a boon for compiler optimization, since it means that cdr can be open coded or even folded, if its arguments are constants.

Another boon for optimization is that procedures defined in a library, whether exported or not, can be inlined into other procedures within the library, since there is no concern that some importer of the library can modify the value. For the procedures that a compiler cannot or chooses not to inline, the compiler can avoid construct-

\footnote{Copyright \copyright 2010 Andrew Keep and R. Kent Dybvig. This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from the National Security Agency.}
effect. Even though the libraries are combined into a single object file, each remains visible separately outside of the group.

For most languages, such a form would be almost trivial to implement. In Scheme, however, the implementation is complicated significantly by the fact that the compilation of one library can involve the actual use of another library’s run-time bindings. That is, as each library in a library group is compiled, it might require another in the same group to be compiled and loaded. This need arises from Scheme’s procedural macros. Macros are defined by transformers that are themselves coded in Scheme. Macro uses are expanded at compile time or, more precisely, expansion time, which precedes compilation. If a macro used in one library depends on the run-time bindings of another, the other must be loaded before the first library can be compiled. This need arises even when libraries do not export keyword (macro) bindings, although the export of keywords can cause additional complications.

As with libraries themselves, the library-group implementation is entirely handled by the macro expander and adds no additional burdens or constraints on the rest of the compiler. This makes it readily adaptable to other implementations of Scheme and even to implementations of other languages that support procedural macros, now or in the future.

The rest of this paper is organized as follows. Section 2 provides background about the library form and Ghuloum’s library implementation, which we use as the basis for describing our implementation. Section 3 introduces the library-group form, discusses what the expander produces for a library group, and describes how it does so. Section 4 illustrates when cross-library optimization is be helpful. Sections 5 and 6 discuss related and future work, and Section 7 presents our conclusions.

2. Background

This section describes R6RS libraries and top-level programs, which are the building blocks for our library groups. It also covers those aspects of Ghuloum’s implementation of libraries that are relevant to our implementation of library groups.

2.1 Libraries and top-level programs

An R6RS library is defined via the library form, as illustrated by the following trivial library.

(library (A)
  (export fact)
  (import (rnrs))
  (define fact
    (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1)))))))

The library is named (A), exports a binding for the identifier fact, and imports from the (rnrs) library. The (rnrs) library exports bindings for most of the identifiers defined by R6RS, including define, lambda, if, zero?, *, and -, which are used in the example. The body of the library consists only of the definition of the exported fact.

For our purposes, library names are structured as lists of identifiers, e.g., (A), (rnrs), and (rnrs io simple). The import form names one or more libraries. Together with the definitions in the library’s body, the imported libraries determine the entire set of identifiers visible within the library’s body. A library’s body can contain both definitions and initialization expressions, with the definitions preceding the expressions. The identifiers defined within a library are either run-time variables, defined with define, or keywords, defined with define-syntax.

Exports are simply identifiers. An exported identifier can be defined within the library, or it can be imported into the library and reexported. In Scheme, types are associated with values, not variables, so the export form does not include type information, as it typically would for a statically typed language. Exported identifiers are immutable. Library import forms cannot result in cyclic dependencies, so the direct dependencies among a group of libraries always form a directed acyclic graph (DAG).

The R6RS top-level program below uses fact from library (A) to print the factorial of 5.

(import (rnrs) (A))
(write (fact 5))

All top-level programs begin with an import form listing the libraries upon which it relies. As with a library body, the only identifiers visible within a top-level program’s body are those imported into the program or defined within the program. A top-level-program body is identical to a library body.

The definitions and initialization expressions within the body of a library or top-level program are evaluated in sequence. The definitions can, however, be mutually recursive. The resulting semantics can be expressed as a letrec*, which is a variant of letrec that evaluates its right-hand-side expressions in order.

2.1.1 Library phasing

Figures 1, 2, and 3 together illustrate how the use of macros can lead to the need for phasing between libraries. The (tree) library implements a basic set of procedures for creating, identifying, and modifying simple tree structures built using a tagged vector. Each tree node has a value and list of children, and the library provides accessors for getting the value of the node and the children. As with library (A), (tree) exports only run-time (variable) bindings.

Library (tree constants) defines a macro that can be used to create constant (quoted) tree structures and three variables bound to constant tree structures. The quote-tree macro does not simply expand into a set of calls to make-tree because that would create (nonconstant) trees at run time. Instead, it directly calls make-tree at expansion time to create constant tree structures. This sets up a compile-time dependency for (tree constants) on the run-time bindings of (tree).

Finally, the top-level program shown in Figure 3 uses the exports of both (tree) and (tree constants). Because it uses quote-tree, it depends upon the run-time exports of both libraries at compile time and at run time.

The possibility that one library’s compile-time or run-time exports might be needed to compile another library sets up a library phasing problem that must be solved by the implementation. We say that a library’s compile-time exports (i.e., macro definitions) comprise its visit code, and its run-time exports (i.e., variable definitions and initialization expressions) comprise its invoke code. When a library’s compile-time exports are needed (to compile another library or top-level program), we say the library must be visited, and when a library’s run-time exports are needed (to compile or run another library or top-level program), we say the library must be invoked.

In the tree example, library (tree) is invoked when library (tree constants) is compiled because the quote-tree forms in (tree constants) cannot be expanded without the run-time exports of (tree). For the same reason, library (tree) is in-

---

1 This description suppresses several details of the syntax, such as support for library versioning, renaming of imports or exports, identifiers exported indirectly via the expansion of a macro, and the ability to export other kinds of identifiers, such as record names.

2 Actually, definitions and initialization expressions can be interleaved in a top-level-program body, but this is a cosmetic difference of no importance to our discussion.
(library (tree))
(export make-tree tree? tree-value tree-children)
(import (rnrs))
(define tree-id #xbacca)
(define make-tree (case-lambda
  [[()] (make-tree #f '())
  [(v) (make-tree v '())
  [(v c) (vector tree-id v c)])]]
(define tree? (lambda (t)
  (and (vector? t)
    (eqv? (vector-ref t 0) tree-id)))))
(define tree-value (lambda (t) (vector-ref t 1)))
(define tree-children (lambda (t) (vector-ref t 2))))

Figure 1. The (tree) library, which implements a tree data structure.

(library (tree constants))
(export quote-tree t0 t1 t2)
(import (rnrs) (tree))
(define-syntax quote-tree (lambda (x)
  (define q-tree-c (lambda (x)
    (syntax-case x ()
      [[v c* ...] (make-tree #\'v
        (map q-tree-c #\'(c* ...)))]
      [v (make-tree #\'v)])))
  (syntax-case x ()
    [[(c* ...)] #\'(quote-tree #\'f)]
    [[quote-tree v c* ...]
      #\'#,(make-tree #\'v
        (map q-tree-c #\'(c* ...)))]])
(define t0 (quote-tree))
(define t1 (quote-tree 0))
(define t2 (quote-tree 1 (2 3 4) (5 6 7))))

Figure 2. The (tree constants) library, which defines a mechanism for creating constant trees and a few constant trees of its own.

(import (rnrs) (tree) (tree constants))
(define tree->list (lambda (t)
  (cons (tree-value t)
    (map tree->list (tree-children t)))))
(write (tree->list t0))
(write (tree->list t1))
(write (tree-value (car (tree-children t2))))
(write (tree->list (quote-tree 5 (7 9)))))

Figure 3. A program using the (tree) and (tree constants) libraries.

The tree example takes advantage of implicit phasing [21]. R6RS also allows an implementation to require explicit phase declarations as part of the import syntax. The library-group form described in this paper, and its implementation, are not tied to either phasing model, so this paper has no more to say about the differences between implicit and explicit phasing.

2.2 Library implementation

The compiled form of a library consists of metadata, compiled visit code, and compiled invoke code. The metadata represents information about the library’s dependencies and exports, among other things. The compiled visit code evaluates the library’s macro-transformer expressions and sets up the bindings from keywords to transformers. The compiled invoke code evaluates the right-hand-sides of the library’s variable definitions, sets up the bindings from variables to their values, and evaluates the initialization expressions.

When the first import of a library is seen, a library manager locates the library, loads it, and records its metadata, visit code, and invoke code in a library record data structure as illustrated for libraries (tree) and (tree constants) in Figure 4. The metadata consists of the library’s name, a unique identifier (UID), a list of exported identifiers, a list of libraries that must be invoked before the library is visited, and a list of libraries that must be invoked before the library is invoked. The UID uniquely identifies each compilation instance of a library and is used to verify that other compiled libraries and top-level programs are built against the same compilation instance. In general, when a library or top-level program is compiled, it must be linked only with the same compilation instance of an imported library. An example illustrating why this is necessary is given in Section 3.3.

Subsequent imports of the same library do not cause the library to be reloaded, although in our implementation, a library can be reloaded explicitly during interactive program development.

Once a library has been loaded, the expander uses the library’s metadata to determine the library’s exports. When a reference to an export is seen, the expander uses the metadata to determine whether it is a compile-time export (keyword) or run-time export (variable). If it is a compile-time export, the expander runs the library’s visit code to establish the keyword bindings. If it is a runtime export, the expander’s action depends on the “level” of the code being expanded. If the code is run-time code, the expander merely records that the library or program being expanded has an invoke requirement on the library. If the code is expand-time code (i.e., code within a transformer expression on the right-hand-side of a define-syntax or other keyword binding form), the expander records that the library or program being expanded has a visit requirement on the library, and the expander also runs the library’s invoke code to establish its variable bindings and perform its initialization.

Since programs have no exports, they do not have visit code and do not need most of the metadata associated with a library. Thus, a program’s representation consists only of invoke requirements and invoke code, as illustrated at the top of Figure 4. In our implementation, a program record is never actually recorded anywhere, since the program is invoked as soon as it is loaded.

As noted in Section 2.1, library and top-level program bodies are evaluated using letrec* semantics. Thus, the invoke code produced by the expander for a library or top-level program is structured as a letrec*, as illustrated below for library (tree),
with — used to represent the definition right-hand-side expressions, which are simply expanded versions of the corresponding source expressions.

(\letrec* ([make-tree — ]
  [tree? — ]
  [tree-value — ]
  [tree-children — ])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children))

If the library contained initialization expressions, they would appear just after the letrec* bindings. If the library contained unexported variable bindings, they would appear in the letrec* along with the exported bindings.

We refer to the identifiers $make-tree, $tree?, $tree-value, and $tree-children as library globals. These are the handles by which other libraries and top-level programs are able to access the exports of a library. In our system, library globals are implemented as ordinary top-level bindings in the sense of the Revised* Report on Scheme [23]. To avoid name clashes with other top-level bindings and with other compilation instances of the library, library globals are actually generated symbols (gensyms). In fact, the list of exports is not as simple as portrayed in Figure 4, since it must identify the externally visible name, e.g., make-tree, whether the identifier is a variable or keyword, and, for variables, the generated name, e.g., the gensym represented by $make-tree.

It would be possible to avoid binding the local names, e.g., make-tree, and instead directly set only the global names, e.g., $make-tree. Binding local names as well as global names enables the compiler to perform the optimizations described in Section 1 involving references to the library’s exported variables within the library itself. Our compiler is not able to perform such optimizations when they involve references to top-level variables, because it is generally impossible to prove that a top-level variable’s value never changes even with whole-program analysis due to the potential use of eval. We could introduce a new class of immutable variables to use as library globals, but this would cause problems in our system if a compiled library is ever explicitly reloaded. It is also easier to provide the compiler with code it already knows how to optimize than to teach it how to deal with a new class of immutable top-level variables.

3. The library-group form

Having now a basic understanding of how libraries work and how they are implemented, we are ready to look at the library-group form. This section describes the form, its usage, what the expander should produce for the form, and how the expander does so. It also describes a more portable variation of the expansion.

3.1 Usage

Both the (tree) and (tree constants) libraries are required when the top-level program that uses them is run. If the program is an application to be distributed, the libraries would have to be distributed along with the program. Because the libraries and program are compiled separately, there is no opportunity for the compiler to optimize across the boundaries and no chance for the compiler to detect ahead of time if one of the procedures exported by (tree) is used improperly by the program. The library-group form is designed to address all of these issues.

Syntactically, a library-group form is a wrapper for a set of library forms and, optionally, a top-level program. Here is how it might look for our simple application, with — used to indicate portions of the code that have been omitted for brevity.

(library-group
  (library (tree) — )
  (library (tree constants) — )
  (import (rnrs) (tree) (tree constants))
  (define tree->list
    (lambda (t)
      (cons (tree-value t)
        (map tree->list (tree-children t)))))
  (write (tree->list t0))
  (write (tree->list t1))
  (write (tree-value (car (tree-children t2))))
  (write (tree->list (quote-tree 5 (7 9)))))

The following grammar describes the library-group syntax:
where `library` is an ordinary R6RS library form and `program` is an ordinary R6RS top-level program. A minor but important twist is that a library or the top-level program, if any, can be replaced by an `include` form that names a file containing that library or program. In fact, we anticipate this will be done more often than not, so the existing structure of a program and the libraries it uses is not disturbed. In particular, when `include` is used, the existence of the `library-group` form does not interfere with the normal library development process or defeat the purpose of using libraries to organize code into separate logical units. So, our simple application might instead look like:

```scheme
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "app.sps"))
```

In the general case, a `library-group` packages together a program and multiple libraries. There are several interesting special cases. In the simplest case, the `library-group` form can be empty, with no libraries and no program specified, in which case it is compiled into nothing. A `library-group` form can also consist of just the optional top-level program form. In this case, it is simply a wrapper for the top-level program it contains, as `library` is a wrapper for libraries. Similarly, the `library-group` form can consist of a single library form, in which case it is equivalent to just the `library` form by itself. Finally, we can have just a list of library forms, in which case the `library-group` form packages together libraries only, with no program code.

A `library-group` form is not required to encapsulate all of the libraries upon which members of the group depend. For example, we could package together just (`tree/constants`) and the top-level program:

```scheme
(library-group
  (include "tree/constants.sls")
  (include "app.sps"))
```

leaving (`tree`) as a separate dependency of the library group. This is important since the source for some libraries might be unavailable. In this case, a library group contains just those libraries for which source is available. The final distribution can include any separate, binary libraries. Conversely, a `library-group` form can contain libraries upon which neither the top-level program (if present) nor any of the other libraries explicitly depend, e.g.:

```scheme
(library-group
  (include "tree.sls")
  (include "tree/constants.sls")
  (include "foo.sls")
  (include "app.sps"))
```

Even for whole programs packaged in this way, including an additional library might be useful if the program might use `eval` to access the bindings of the library at run time. This supports the common technique of building modules that might or might not be needed into an operating system kernel, web server, or other program. The advantage of doing so is that the additional libraries become part of a single package and they benefit from cross-library error checking and optimization for the parts of the other libraries they use. The downside is that libraries included but never used might still have their invoke code executed, depending on which libraries in the group are invoked. This is the result of combining the invoke code of all the libraries in the group. The programmer has the responsibility and opportunity to decide what libraries are profitable to include.

Apart from the syntactic requirement that the top-level program, if present, must follow the libraries, the `library-group` form also requires that each library be preceded by any other library in the group that it imports. So, for example:

```scheme
(library-group
  (include "tree/constants.sls")
  (include "tree.sls")
  (include "app.sps"))
```

would be invalid, because (`tree constants`) imports (`tree`). One or more appropriate orderings are guaranteed to exist because R6RS libraries are not permitted to have cyclic import dependencies.

The expander could determine an ordering based on the `import` forms (including `local import` forms) it discovers while expanding the code. We give the programmer complete control over the ordering, however, so that the programmer can resolve dynamic dependencies that arise from invoke-time calls to `eval`. Another solution would be to reorder only if necessary, but we have so far chosen not to reorder so as to maintain complete predictability.

Libraries contained within a `library-group` form behave like their standalone equivalents, except that the `include` code of the libraries is fused. Fusing the code of the enclosed libraries and top-level program facilitates compile-time error checking and optimization across the library and program boundaries. If compiled to a file, the form also produces a single object file. In essence, the `library-group` form changes the basic unit of compilation from the library or top-level program to the `library-group` form, without disturbing the enclosed (or included) libraries or top-level programs.

A consequence of fusing the `include` code is that the first time a library in the group is invoked, the libraries up to and including that library are invoked as well, along with any side effects doing so might entail. In cases where all of the libraries in the group would be invoked anyway, such as when a top-level program that uses all of the libraries is run, this is no different from the standalone behavior.

Fusing the `include` code creates a more subtle difference between grouped and standalone libraries. The import dependencies of a group of R6RS libraries must form a DAG, i.e., must not involve cycles. An exception is raised at compile time for static cyclic dependencies and at run time for dynamic cyclic dependencies that arise via `eval`. When multiple libraries are grouped together, a synthetic cycle can arise, just as cycles can arise when arbitrary nodes in any DAG are combined. We address the issue of handling dynamic cycles in more depth in the next subsection.

### 3.2 Anticipated expander output

This section describes what we would like the expander to produce for the `library-group` form and describes how the expander deals with import relationships requiring one library’s run-time exports to be available for the expansion of another library within the group.

As noted in Section 2, the explicit import dependencies among libraries must form a directed acyclic graph (DAG), and as shown in Section 2.2, the invoke code of each library expands independently into a `letrec*` expression. This leads to an expansion of `library-group` forms as nested `letrec*` forms, where each li-

---

1 An included file can actually contain multiple libraries or even one or more libraries and a program, but we anticipate that each included file typically contains just one library or program.

2 Visit code is not fused as there is no advantage in doing so. 
Figure 5. A nested letrec* for our library group, with — indicating code that has been omitted for brevity.

library expands to a letrec* form containing the libraries following it in the group. The code for the top-level program is nested inside the innermost letrec* form. Libraries are nested in the order provided by the programmer in the library-group form.

Figure 5 shows the result of this nesting of letrec* forms for the first library group defined in Section 3.1. This is a good first cut. The references to each library global properly follow the assignment to it, which remains properly nested within the binding for the corresponding local variable. Unfortunately, this form does not allow the compiler to analyze and optimize across library boundaries, because the inner parts of the letrec* nest refer to the global rather than to the local variables.

To address this shortcoming, the code must be rewired to refer to the local variables instead, as shown in Figure 6. With this change, the invoke code of the library group now forms a single compilation unit for which cross-library error checking and optimization is possible.

Another issue remains. Loading a library group should not automatically execute the shared invoke code. To address this issue, the code is abstracted into a separate procedure, p, called from the invoke code stored in each of the library records. Rather than running the embedded top-level-program code, p returns a thunk that can be used to run that code. This thunk is ignored by the library invoke code, but it is used to run the top-level program when the library group is used as a top-level program. The procedure p for the tree library group is shown in Figure 7.

Unfortunately, this expansion can lead to synthetic cycles in the dependency graph of the libraries. Figure 8 shows three libraries with simple dependencies: (C) depends on (B) which in turn depends on (A).

We could require the programmer to include library (B) in the library group, but a more general solution that does not require this is preferred. The central problem is that (B) needs to be run after the invoke code for library (A) is finished and before the invoke code for library (C) has started. This can be solved by marking

Figure 6. A nested letrec* for our library group, with library-global references replaced by local-variable references.

(lambda ()
  (letrec* ([tree-id —]
    [make-tree —]
    [tree? —]
    [tree-value —]
    [tree-children —])
  (set-top-level! $make-tree make-tree)
  (set-top-level! $tree? tree?)
  (set-top-level! $tree-value tree-value)
  (set-top-level! $tree-children tree-children)
  (letrec* ([t0 —]
            [t1 —]
            [t2 —])
    (set-top-level! $t0 t0)
    (set-top-level! $t1 t1)
    (set-top-level! $t2 t2)
    (letrec* ([tree->list ]
               (lambda (t)
                 (cons ($tree-value t)
                       (map tree->list ($tree-children t)))))
      (write (tree->list t0))
      (write (tree->list t1))
      (write (tree-value (car (tree-children t2))))
      (write (tree->list (quote tree constant)))))))

(lambda ()
  (letrec* ([tree->list ]
               (lambda (t)
                 (cons (tree-value t)
                       (map tree->list (tree-children t)))))
      (write (tree->list t0))
      (write (tree->list t1))
      (write (tree-value (car (tree-children t2))))
      (write (tree->list (quote tree constant)))))))

Figure 7. The final invoke code expansion target.
library (A) (library (B) (library (C))
(export x) (export y) (export z)
(import (rnrs)) (import (rnrs) (B))
(define x 5) (define y (+ y 5))

Figure 8. Three simple libraries, with simple dependencies

(library-group (library (A) (library (C))

Figure 9. A library-group form containing (A) and (C)

(lambda ()
(letrec* ([x 5])
(set-top-level! $x x)
($mark-invoked! 'A)
($invoke-library '((B) '() 'B)
(letrec* ([z (+ y 5)])
(set-top-level! $z z)
($mark-invoked! 'C))))

Figure 10. Expansion of library group marking (A) as invoked
and invoking (B)

(lambda (uid)
(letrec* ([x 5])
(set-top-level! $x x)
($mark-invoked! 'A)
(let ([nested-lib
(lambda (uid)
(letrec* ([tree-id
(make-tree)
(letrec* ([tree-value
(tree->list)]
(if (eq? uid 'tree)
nested-lib
(nested-lib uid))))])

Figure 11. Final expansion for correct library groups

library (A) as invoked once its invoke code is complete and
explicitly invoking (B) before (C)’s invoke code begins. Figure 10
shows what this invoke code might look like.

This succeeds when (A) or (C) are invoked first, but results in
a cycle when (B) is invoked first. Effectively, the library group
invoke code should stop once (A)’s invoke code has executed.
Wrapping each library in a lambda that takes the UID of the library
being invoked accomplishes this. When a library group is invoked,
the UID informs the invoke code where to stop and returns any
nested library’s surrounding lambda as the restart point. Figure 11
shows this corrected expansion of the library group containing (A)
and (C). The invoke code for an included program would replace
the innermost nested-lib, and be called when #f is passed in
place of the UID.

Beyond the issues in the invoke code, we would also like to en-
sure that libraries in the group are properly installed into the library
manager. For the most part, libraries in the group can be handled
like standalone libraries. Metadata and visit code is installed into
the library manager as normal. The invoke code is the only twist.
We would like to ensure that each library in the library group is
invoked only once, the first time it or one of the libraries below
it in the group is invoked. Thus, each library is installed with the
shared invoke procedure described above. Figure 12 shows how
our library records are updated from Figure 4 to support the shared
invoke code. Figure 13 shows this final expansion for our tree li-
brary group. If the optional program were not supplied, the call to
the p thunk at the bottom would be omitted. When the optional
program is supplied, it always executes when the library group
is loaded. Programmers wishing to use the library group separately
can create two versions of the library group, one with the top-level
program and one without.
3.3 Implementation

A major challenge in producing the residual code shown in the preceding section is that the run-time bindings for one library might be needed while compiling the code for another library in the group. A potential simple solution to this problem is to compile and load each library before compiling the next in the group. This causes the library (and any similar library) to be compiled twice, but that is not a serious concern if the compiler is fast or if the library-group form is used only in the final stage of an application’s development to prepare the final production version.

Unfortunately, this simple solution does not work because the first compilation of the library may be fatally incompatible with the second. This can arise for many reasons, all having to do ultimately with two facts. First, macros can change much of the nature of a library, including the internal representations used for its data structures and even whether an export is defined as a keyword or as a variable. Second, since macros can take advantage of the full power of the language, the transformations they perform can be affected by the same things that affect run-time code, including, for example, information in a configuration file, state stored elsewhere in the file system by earlier uses of the macro, or even a random number generator.

For example, via a macro that flips a coin, e.g., checks to see if a random number generator produces an even or odd answer, the (tree) library might in one case represent trees as tagged lists and in another as tagged vectors. If this occurs, the constant trees defined in the (tree constants) library and in the top-level program would be incompatible with the accessors used at run time. While this is a contrived and whimsical example, such things can happen and we are obligated to handle them properly in order to maintain consistent semantics between separately compiled libraries and libraries compiled as part of a library group.

On the other hand, we cannot entirely avoid compiling the code for a library whose run-time exports are needed to compile another part of the group if we are to produce the run-time code we hope to produce. The solution is for the expander to expand the code for each library only once, as it is seen, just as if the library were compiled separately from all of the other libraries. If the library must be invoked to compile another of the libraries or the top-level program, the expander runs the invoke code through the rest of the compiler and evaluates the result. Once all of the libraries and the top-level program have been expanded, the expander can merge and rewrite the expanded code for all of the libraries to produce the code described in the preceding section, then allow the resulting code to be run through the rest of the compiler. Although some of the libraries might be put through the rest of the compiler more than once, each is expanded exactly once. Assuming that the rest of the compiler is deterministic, this prevents the sorts of problems that arise if a library is expanded more than once.

In order to perform this rewiring, the library must be abstracted slightly so that a mapping from the exported identifiers to the lexical variables can be maintained. With this information the code can be rewired to produce the code in Figure 13.

Since a library’s invoke code might be needed to expand another library in the group, libraries in the group are installed as standalone libraries during expansion and are then replaced by the library group for run time. This means that the invoke code for a library might be run twice in the same Scheme session, once during expansion and once during execution. Multiple invocations of a library are permitted by the R6RS. Indeed, some implementations always invoke a library one or more times at compile time.
and again at run time in order to prevent state set up at compile time from being used at run time.

This implementation requires the expander to walk through expanded code converting library-global references to lexical-variable references. Expanded code is typically in some compiler-dependent form, however, that the expander would not normally need to traverse, and we might want a more portable solution to this problem. One alternative to the code walk is to wrap the expanded library in a \texttt{lambda} expression with formal parameters for each library global referenced within the library.

4. Empirical Evaluation

One of the goals of the \texttt{library-group} form is to enable cross-library optimizations to take place. Optimizations like procedure inlining are known to result in significant performance benefits [36]. By using the \texttt{library-group} form, a program enables a compiler that supports these optimizations to apply them across library boundaries. This section characterizes the types of programs we expect to show performance benefits. Even when there are no performance benefits, programs still benefit from the single binary output file and cross-library compile-time error checking.

In general, programs and libraries with many cross-library procedure calls are expected to benefit the most. As an example, imagine a compiler where each pass is called only once and is defined in its own library. Combining these libraries into a library group is unlikely to yield performance benefits, since the number of cross-library procedure calls is relatively small. If the passes of this compiler use a common record structure to represent code, however, and a library of helpers for decomposing and reconstructing these records, combining the compiler pass libraries and the helper library into a single library group can benefit compiler performance significantly.

To illustrate when performance gains are expected, we present two example libraries, both written by Eduardo Cavazos and tested in Chez Scheme Version 8.0 [12]. The first program [8] implements a set of tests for the “Mathematical Pseudo Language” [10, 11] (MPL), a symbolic math library. The second uses a library for indexable sequences [7] to implement a matrix multiply algorithm [13].

Many small libraries comprise the MPL library. Each basic mathematical function, such as \texttt{+}, \texttt{*}, and \texttt{cos}, uses pattern matching to decompose the mathematical expression passed to it to select an appropriate simplification, if one exists. The pattern matcher, provided by another library [14], avoids cross-library calls, since it is implemented entirely as a macro. Thus, most of the work for each function is handled within a single library. The main program tests each algorithm a handful of times. Compiling the program with the \texttt{library-group} form showed only a negligible performance gain. This example typifies programs that are unlikely to improve performance with the \texttt{library-group} form. Since computation is mostly performed within libraries, the optimizer has little left to optimize across the library boundaries.

The matrix multiply example uses a \texttt{vector-for-each} function providing the loop index to its procedure argument, from the indexable-sequence library. The library abstracts standard data structure iteration functions that provide constructors, accessors, and a length function. The matrix multiply function makes three nested calls to \texttt{vector-for-each-with-index} resulting in many cross-library calls. Combining matrix multiply with the indexable-sequence library allows the optimizer to inline these cross-library procedure calls. A test program calls matrix multiply on 50 \times 50, 100 \times 100, and 500 \times 500 matrices. Using the \texttt{library-group} form results in a 30\% speed-up over the separately compiled version.

In both of our example programs the difference in time between compiling the program as a set of individual libraries and as a single \texttt{library-group} is negligible.

5. Related work

Packaging code into a single distributable is not a new problem, and previous dialects of Scheme needed a way to provide a single binary for distribution. Our system, PLT Scheme, and others provide mechanisms for packaging up and distributing collections of compiled libraries and programs. These are packaging facilities only and do not provide the cross-library optimization or compile-time error checking provided by the \texttt{library-group} form.

Ikarus [19] uses Waddell’s source optimizer [35, 36] to perform some of the same interprocedural optimizations as our system. In both systems, these optimizations previously occurred only within a single compilation unit, e.g., a top-level expression or library. The \texttt{library-group} form allows both to perform cross-library and even whole-program optimization. The Stalin [30] Scheme compiler supports aggressive whole-program optimization when the whole program is presented to it, but it does not support R6RS libraries or anything similar to them. If at some point it does support R6RS libraries, the \texttt{library-group} form would be a useful addition. MIT Scheme [22] allows the programmer to mark a procedure inlinable, and inlining of procedures so marked occurs across file boundaries. MIT Scheme does not support R6RS libraries, and inlining, while important, is only one of many optimizations enabled when the whole program is made available to the compiler. Thus, as with Stalin, if support for R6RS libraries is added to MIT Scheme, the \texttt{library-group} form would be a useful addition.

Although the \texttt{library-group} mechanism is orthogonal to the issue of explicit versus implicit phasing, the technique we use to make a library’s run-time bindings available both independently at compile time and as part of the combined library-group code is similar to techniques Flatt uses to support separation of phases [16].

Outside the Scheme community several other languages, such as Dylan, ML, Haskell, and C++, make use of library or module systems and provide some form of compile-time abstraction facility. Dylan is the closest to Scheme, and is latently typed with a rewrite-based macro system [27]. Dylan provides both libraries and modules, where libraries are the basic compilation unit and modules are used to control scope. The Dylan community also recognizes the benefits of cross-library inlining, and a set of common extensions allow programmers to specify when and how functions should be inlined. By default the compiler performs intra-library inlining, but \texttt{may-inline} and \texttt{inline} specify the compiler may try to perform inter-library inlining or that a function should always be inlined even across library boundaries.

The Dylan standard does not include procedural macros, so run-time code from a Dylan library does not need to be made available at compile time, but such a facility is planned [15] and at least one implementation exists [5]. When this feature is added to existing Dylan implementations, an approach similar to that taken by the \texttt{library-group} might be needed to enable cross-library optimization.

ML functors provide a system for parameterizing modules across different type signatures, where the types needed at compile time are analogous to Scheme macros. The MLton compiler [37] performs whole program compilation for ML programs and uses compile-time type information to specialize code in a functor. Since this type information is not dependent on the run-time code of other modules, it does not require a module’s run-time code to be available at compile time. If the type system were extended to support dependent types, however, some of the same techniques used in the \texttt{library-group} form may be needed. Additionally, MetaML [32] adds staging to ML, similar to the phasing in Scheme macros. Since
MetaML does not allow run-time procedures to be called in its templates though, it does not have the same need to make a module’s run-time code available at compile time.

The Glasgow Haskell Compiler [1] (GHC) provides support for cross-module inlining [33] as well as compile-time meta-programming through Template Haskell [28]. Thus, GHC achieves some of the performance benefits of the library-group form in a language with similar challenges, without the use of an explicit library-group form. A Haskell version of the library-group form would still be useful for recognizing when an inline candidate is singly referenced and for enabling other interprocedural optimizations. It would likely be simpler to implement due to the lack of state at compile time.

The template system of C++ [2, 4] provides a Turing-complete, compile-time abstraction facility, similar to the procedural macros found in Scheme. The language of C++ templates is distinct from C++, and run-time C++ code cannot be used during template expansion. If the template language were extended to allow C++ templates to call arbitrary C++ code, compilation might need to be handled similarly to the way the library-group form is handled.

Another approach to cross-library optimizations is link-time optimization of object code. Several different approaches to this technique exist and are being used to optimize compiler code [26] and compiler frameworks like LLVM [24]. Instead of performing procedure inlining at the source level, these optimizers take object code produced by the compiler and perform optimization when the objects are linked. The GOLd [6] link-time optimizer applies similar techniques to optimize cross-module calls when compiling Gambit-C Scheme code into C. Our decision to combine libraries at the source level is motivated by the fact that our system and others already provide effective source optimizers that can be leveraged to perform cross-library optimization.

6. Future work

The library-group form is designed to allow programmers the greatest possible flexibility in determining which libraries to include in a library group and the order in which they should be invoked. This level of control is not always necessary, and we envision a higher-level interface to the library-group form that would automatically group a program with its required libraries and automatically determine an appropriate invocation order based only on static dependencies.

The library-group form ensures that all exports for libraries in the library group are available outside the library group. In cases where a library is not needed outside the library group, we would like to allow their exports to be dropped, so that the compiler can eliminate unused code and data. This would help reduce program bloat in cases where a large utility library is included in a program and only a small part of it is needed. We envision an extended version of the library-group form that specifies a list of libraries that should not be exported. The compiler should still, at least optionally, register unexported libraries in order to raise an exception if they are used outside the library group.

Our current implementation of the library-group form can lead to libraries being invoked that are not required, based on the ordering of libraries in the group. It is possible to invoke libraries only as they are required by using a more intricate layout of library bindings, similar to the way letrec and letrec* are currently handled [20]. This expansion would separate side-effect free expressions in a library from those with side-effects, running the effectful expressions only when required. This approach would require other parts of the compiler be made aware of the library-group form, since the expander does not have all the information it needs to handle this effectively.

7. Conclusion

The library-group form builds on the benefits of R6RS libraries and top-level programs, allowing a single compilation unit to be created from a group of libraries and an optional top-level program. Packaging the run-time code in a single compilation unit and wiring the code together so that each part of the library group references the exports of the others via local variables allows the compiler to perform cross-library optimization and extends compile-time error checking across library boundaries. It also allows the creation of a single output binary. The implementation is designed to deliver these benefits without requiring the compiler to do any more than it already does. In this way it represents a non-invasive feature that can be more easily incorporated into existing Scheme compilers.

While this work was developed in the context of Scheme, we expect the techniques described in this paper will become useful as other languages adopt procedural macro systems. The PLOT language [25], which shares an ALGOL-like syntax with Dylan already provides a full procedural macro system, and a similar system has been proposed for Dylan [15]. The techniques described in this paper might also be useful for languages with dependent-type systems that allow types to be expressed in the full source language or template meta-programming systems that allow templates to be defined using the full source language.

Acknowledgments

Aaron Hsu first suggested that we support cross-library optimization and error checking, and Michael Lenaghan first suggested that we support the ability to create applications that consist of a single object file combining an application program and libraries. The desire to take both suggestions led naturally to the development of the library-group form, which accomplishes both simultaneously. Comments from the reviewers lead to improvements in the presentation.

References
