Automatic Cross-Library Optimization *

Andrew W. Keep
University of Utah
akeep@cs.utah.edu

R. Kent Dybvig
Cisco Systems Inc.
dyb@cisco.com

Abstract

The library construct added to Scheme by the Revised[6] Report on Scheme (R6RS) provides a natural boundary for compilation units, particularly for separate compilation. Unfortunately, using the library as the compilation unit for Scheme programs can interfere with optimizations such as inlining that are important for good performance of compiled programs. Our Scheme system provides a way for specifying larger compilation units, the library group, which allows the source code from several libraries and, optionally, a program to be compiled as a single compilation unit. The library group form works well, but is not a good fit for situations where all of the source code is not available at compile time, particularly in the case where a library is distributed in binary form to be used by other library or application developers. In order to handle situations like this, we have introduced a new, automatic, cross-library optimization mechanism. The automatic cross-library optimization mechanism provides some of the benefits of the library group form without requiring modifications to the program and without requiring libraries to be compiled together. Cross-library optimization is supported by recording additional information in the library binary that can be used when the library is imported by another library or program. This paper describes our automatic cross-library optimization and compares it with the existing library group system.

1. Introduction

The Revised[6] Report on Scheme (R6RS) [20] introduced a new way of structuring programs, separating them into libraries and top-level programs. Each library or top-level program begins in an empty environment, and the programmer is required to import identifiers explicitly. Identifiers exported from a library are immutable. This means, for instance, that if a library imports the `(rnrs)` library, the programmer (and compiler) can be sure that `car` and `cdr` refer to built-in primitives and cannot be arbitrarily redefined. This allows further optimizations, such as open-coding of primitives, to occur within the library. The closed environment also makes it easier for the compiler to statically check that argument counts are correct for primitive calls and for procedure calls, when the procedure is defined within the library.

Organizing a program into a set of libraries and a top-level program makes separate compilation an easier task, since the libraries and the top-level program each have a closed environment with known dependencies. The additional static information in a library or top-level program allows the compiler to perform further optimizations within the library. Unfortunately, separating an application into several libraries and a top-level program interferes with other important optimizations, such as inlining, constant propagation, and related optimizations [10], that could be done if a program were compiled as a single compilation unit, which are now effectively stopped at the library boundaries.

The `library-group` form [14] allows programmers to combine several libraries and, optionally, a top-level program into a single compilation unit. This provides the benefits of cross-library optimization and allows the programmer to specify which code should be compiled together. A library group works well when all of the source code the programmer wishes to combine is available. A library might, however, be pre-compiled and provided only in binary form by a third party, or a programmer might wish to use a precompiled binary across several programs.

For this reason, our Scheme compiler now performs cross-library optimization, when possible, without requiring the use of a `library-group` form. It does so automatically, with no intervention from the programmer. To support this, when a library is compiled to a file, extra information is recorded in the resulting binary file, that allows cross-library optimization to occur when the library is imported into another library or program. Our automatic cross-library optimization applies techniques similar to the approaches used for cross-module optimization in other functional language compilers, including the Glasgow Haskell Compiler [1], Standard ML of New Jersey [23], and OCaml [2], which provide varying levels of cross-module inlining and related optimizations. Our contribution is adapting these approaches to R6RS libraries.

This paper describes the design and implementation of our automatic cross-library optimization, and how it ties into our

macro expander and source-level optimizer. It describes the limitations of the automatic cross-library optimization that result both from our desire to limit optimizations to those that are likely to be profitable and from the requirements of the Scheme standard. Finally, it compares the effectiveness of automatic cross-library optimization with the existing library-group mechanism.

The remainder of this paper is organized as follows. Section 2 provides brief background information on R6RS libraries and our implementation of them. Section 3 describes our automatic cross-library optimization and its limitations. Section 4 illustrates how our automatic cross-library optimization compares with un-optimized code and code optimized using library groups. Section 5 presents related work, and Section 6 concludes.

2. Background

The R6RS standard created a new way to organize code for an application into a set of libraries and a top-level program. For instance, we could create a new library, `(factorial)`, that exports the standard factorial function, `fact`, as follows:

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

This simple example illustrates the `library` form, which starts with a name, followed by an export clause, followed by an import clause, followed by the body of the library. The `(factorial)` library exports a single identifier, `fact`, and imports the standard R6RS library, `(rnrs)`.

Within the body of a library, definitions must precede expressions. Semantically, variable definitions at the top level of a library body are treated as being in the same `letrec*` form, much like definitions within the body of a `lambda`.

A top-level program that uses the `(factorial)` library can also be defined, as follows:

```
(import (rnrs) (factorial))
(display "factorial of 5: ")
(write (fact 5))
(newline)
```

A top-level program starts with an import clause, followed by the body of the top-level program. Unlike the `library` form, the top-level program does not have a syntactic form that indicates its start and end. Definitions and expressions can also be interleaved in the body of a top-level program. In this program, the `(rnrs)` and `(factorial)` libraries are imported, and the program writes the string `"factorial of 5: "` followed by the calculated factorial of 5, followed by a newline.

2.1 Library Implementation

The `library` and top-level program forms are both handled entirely within the expander of our Scheme compiler. When a library is compiled, either to a file or in memory, a unique identifier is assigned to the compilation instance of the library. The unique identifier is used to ensure that the same compilation instance of a library is always loaded by a library or top-level program that depends upon it. This is important in the presence of procedural macros, where a macro call in a library could produce different code each time the library is compiled. Using the incorrect compilation unit could result in functionality with an inconsistent interface to be imported. For instance, if a macro in a library sometimes uses a list to implement a tree data structure and sometimes uses a vector, two compilation instances of the same library might produce different, and incompatible, results.

As mentioned above, the variable definitions can be semantically expressed using a `letrec*` form. In our implementation, each library is expanded into a `letrec*`, with an entry for each definition, and the expressions of the library, if there are any, in the body of the `letrec*`. In addition to any expressions in the body, a statement setting a top-level global corresponding to each exported identifier is set to the local binding for the definition in the `letrec*` form. For instance, our `(factorial)` example would expand into something like the following:

```
(letrec ([fact (lambda (n)
                 (if (zero? n)
                     1
                     (* n (fact (- n 1)))))])
  ($set-top-level-value! 'fact.7 fact))
```

where `fact.7` is a generated symbol used to represent the `(factorial)` library export. The value of the exported identifier is set using the `$set-top-level-value!` primitive. This makes the `fact` procedure available to libraries or programs that import this library.

In addition to the `letrec*` that defines the library, the expander also produces code to install the library in the library manager. This code also defines a compile-time binding to a library global for each exported identifier. The library global contains both the unique identifier for this compilation instance of the library and the name of the identifier. The expander uses the library global information to determine when a dependency needs to be loaded and which top-level global should be referenced for a library global reference. When a library is imported, the library globals for that library are made available to the importing library. The library is also wrapped in a `lambda` expression that provides an entry point in the compiled library. The argument passed into

the `lambda` expression indicates if the library is being visited (loaded for its compile-time exports, e.g., macros) or invoked (loaded for its run-time exports, e.g. procedure definitions).

For instance, the `(factorial)` library expands into code like the following:

```
(lambda (tmp)
  ($sc-put-cte
    'fact.7
    '(library-global factorial.5 fact.7)
    #f)
  ($install-library '(factorial) '() 'factorial.5
    '(#[libreq (rnrs) (6) $rnrs]) '#() '() '()
    '() void
    (lambda ()
      (letrec* ([fact
                 (lambda (n)
                   (if (zero? n)
                       1
                       (* n (fact (- n 1)))))])
        ($set-top-level-value! 'fact.7 fact)))))
```

The `$sc-put-cte` call creates a new library global binding for the exported `fact` procedure in the compile-time environment. Here, `$sc-put-cte` adds an entry for the library export `fact.7`, the generated symbol that represents the exported `fact` function, associating it with the library global, a list with the library unique identifier, `factorial.5`, and the exported symbol `fact.7`. The library global binding is an important part of our automatic cross-library optimization. The final argument to `$sc-put-cte` indicates top-level global information. Since library exports are available when the library is imported, it is not globally visible, so the final argument is `#f` to indicate it is not global.

The `$install-library` call installs the library into the library manager and sets up the code to be executed when the library is visited (for its compile-time exports) or invoked (for its run-time exports). The first three arguments indicate the library name, `(factorial)`, library version, in this case unspecified, hence `()`, and the unique identifier for the library, `factorial.5`. The next six arguments indicate various types of library dependencies. These include the original import requirements, listed first and indicating that the `(rnrs)` library is imported, and includes the visit dependencies, which must be loaded before the library is visited, and invoke dependencies, loaded before the library is invoked. The other dependencies are used internally to determine when libraries should be recompiled, for instance if an included source file is updated, and to more precisely specify when each dependency is needed. The last two arguments are the visit code, a thunk called when the library is visited, and the invoke code, a thunk called when the library is invoked. In this case, the visit code is simply `void` because this library does not export any compile-time identifiers, i.e., it does not export macros. The invoke code creates the `fact` procedure and sets `fact.7` library global export.

3. Automatic Cross-Library Optimization

Naively, one approach to cross-library optimization would be to include a representation of the source code for the library so that this source code can be incorporated when the library is imported. The challenge in this approach is that any shared-mutable state set up by the library cannot be duplicated without changing the semantics, when the same library is imported into the same session through two different libraries. The other downside of this approach is that it needlessly increases the size of library binaries. Needlessly, because large procedures would not normally be copied by the source optimizer [24], which is set up to limit code growth due to inlining. Hence, source code stored for large procedure would never be used.

Instead, our approach limits the inclusion to a representation of the source code for exported *inlinable* procedures and the constant value for an exported *copyable* constant. This information is attached to the exported identifier in the library global binding. A constant is considered *copyable* when copying it will not change the semantics of a program that uses the constant. The driving decider for this is how `eq?` handles the constant. A structured constant, such as a string, a vector, or a pair, cannot be copied because the constant must be `eq?` to itself, and copying the constant would break this property.[1] Other Scheme objects, such as symbols, numbers, characters, or booleans can be copied.[2] A procedure is considered *inlinable* when it contains no free local variables,[3] only copyable constants, and fits within the *size limit* when the library is compiled.

The *size limit* controls the amount of code expansion that the source inliner will allow when inlining occurs. It is normally based on the size of the code after inlining but must be based on the size before inlining here, as we are operating without knowledge of the call site. The size limit is a parameter that can be set in the Scheme system to allow the compiler to keep larger or smaller procedures around for inlining. Limiting inlinable procedures to those without free variables both avoids potential problems with referencing an identifier that is not bound in the context where the source is copied and any copying of shared mutable state. At this point in

[1] Our compiler and linker preserve structure sharing within a single compilation unit, allowing constants to be propagated freely. It does not presently have any mechanism for preserving structure sharing across compilation units.

[2] A symbol can be copied because our linker preserves sharing of symbols, a number can be copied because the Scheme standard does not require numbers to be comparable with `eq?`, and characters and booleans can be copied because neither is a heap allocated object.

[3] All variables defined in a library are treated as local to allow inlining within the library, but variables imported from other libraries, including primitives, are not.

the compiler, variables are specifically local variables, so references to primitives or globals are not considered free. The one exception to this is references to imported library identifiers. Allowing references to library identifiers would require that library dependencies be updated in libraries that import the one being compiled. There is currently no facility for doing this.

3.1 Compiling Libraries

The implementation of automatic cross-library optimization requires that both the expander and the source optimizer be aware of the optimization. As described in Section 2.1, definitions in a library are bound by a `letrec*` form, and exported identifiers are then set in the top-level environment to the value of the corresponding local variable. To enable automatic cross-library optimization, the internal representation of a library global is extended to contain a mutable field that stores the optimization information for the identifier. For instance, in our `(factorial)` library example, the library global binding set in the compile-time environment is modified to the following:

```
($sc-put-cte
  'fact.7
  '(library-global factorial.5 fact.7 . #f)
  #f)
```

the final field in the library global is where cross-library optimization information is stored. During expansion of the library, this field is set to `#f`, which indicates that no optimization information is available.

The expander also generates a node in the internal representation of the output of the library to indicate to the source optimizer that the contained expression is related to the given library identifier. This changes the assignment of the top-level identifier from our `(factorial)` library to the following:

```
($set-top-level-value! 'fact.7
  (cte-optimization-loc '(fact.7 . #f) fact))
```

where `(fact.7 . #f)` is the same pair as that included in the library global binding. This breadcrumb is the only explicit piece of library information that remains after the expander finishes expanding the library.

When the source optimizer encounters a cross-library optimization node, it first performs source optimization on the expression contained in the node and then inspects the result. If the result is a copyable constant or an inlinable procedure, the optimization field of the associated library export is modified to contain the internal representation of the expression. In our example, the `cdr` of this pair, `'(fact.7 . #f)`, would be updated to contain the cross-library optimization information. This sets up the cross-library optimization, as the internal representation of a library global is written to the binary output file for use when the library is imported. Unfortunately, because the reference to `fact` is free within the body of the `fact` procedure, the `fact` procedure is not inlinable. However, we can transform the example, by moving the loop within the `fact` procedure as follows:

```
(library (factorial)
  (export fact)
  (import (rnrs))
  (define fact
    (lambda (n)
      (let f ([n n])
        (if (zero? n)
            1
            (* n (f (- n 1))))))))
```

In this case, the library global binding would look something like the following:

```
($sc-put-cte
  'fact.7
  '(library-global factorial.5 fact.7 .
     (lambda (n)
       ((letrec ([f (lambda (n)
                      (if (zero? n)
                          1
                          (* n (f (- n 1)))))])
          f)
        n)))
  #f)
```

3.2 Importing Libraries

When a library is imported, the expander and source optimizer are responsible for completing the cross-library optimization by replacing references to library globals with the optimization information, when it is available. Performing the replacement in the expander, when possible, allows library dependencies to be more tightly computed, as there is no dependence on the library if all of the library identifiers used from the library are optimized away. For instance, when the `(factorial)` library is imported into the example program from Section 2, the reference to `fact` would be replaced with the `fact` procedure code, if we use the inlinable version of the `fact` function from the previous section. The procedure code would be inlined as follows:

```
(import (rnrs) (factorial))
(display "factorial of 5: ")
(write ((lambda (n)
          ((letrec ([f (lambda (n)
                         (if (zero? n)
                             1
                             (* n (f (- n 1)))))])
             f)
           n))
        5))
(newline)
```

Since there is no longer any reference to any exported identifiers from the `(factorial)` library, the expander can discard it as a dependency. The source optimizer can also (slightly) improve the code by β-reducing the direct application of the `lambda` expression, as follows:

```
(write ((letrec ([f (lambda (n)
                      (if (zero? n)
                          1
                          (* n (f (- n 1)))))])
          f)
        5))
```

When a library global is not in call position in the library source but is moved into call or test position through inlining, the source optimizer replaces the library global with the optimization information. This allows inlining that would not be possible using only the expander but does not allow library dependencies, which are fixed by the time source optimization occurs, to be dropped.

For example, if we contrive a program like the following, with the `fact` procedure in test position:

```
(import (rnrs) (factorial))
(if fact
    (display "factorial")
    (display "no factorial"))
(newline)
```

The source optimizer would produce:

```
(import (rnrs) (factorial))
(display "factorial")
(newline)
```

Since the reference to `fact` is in test position, and a procedure is a non-false value. The import of the `(factorial)` library, however, cannot be discarded in this case, since the source optimizer is performing the inlining instead of the expander.

3.3 Constants and correctness

The standard requires that if a library export is referenced in another library or top-level program at run time, the library must be invoked before the reference is made. We do not violate the letter of this requirement, since the expanded output does not in fact contain a reference to a propagated or inlined variable. While this is just a technicality, based on an extended and thus nonstandard semantics for the expander, nor do we violate the spirit of the requirement. Given the possibly multiple times at which libraries might or might not be invoked at both compile time and run time due to phasing, library init expressions should not, in our opinion, have externally visible effects.[4] Instead, library init expressions should be used exclusively to create internal structures needed by exported procedures. So, when all of the references to the variable exports of a library can be propagated or inlined away (remembering that inlining does not happen for procedures with post-optimization free variables), the imported bindings do not depend on the internal structures created by the init expressions, and the init code need not be run.

The automatic cross-library optimization is inherently limited. Constructed constants cannot be copied because, if any two libraries with the same constructed constant were imported, the constants would no longer be `eq?` to each other. Only procedure expressions exported from the library that contain no free variables and no references to library globals are eligible for external inlining, as they will be lifted out of their lexical context and do not carry the set of dependencies needed for the library globals. These procedures are further limited by the size of the expression, as the inlining process would normally count only the size of the procedure after inlining and other optimizations it enables occur. The size after inlining can be small enough even when the original expression is too large.

[4] If a library needs to perform externally visible effects, this should be accomplished via an exported init procedure.

4. Empirical Evaluation

The goal of cross-library optimization is to enable constant propagation and procedure inlining across library boundaries, with all of the additional optimization this often enables, including constant folding, useless code elimination, and more constant propagation and procedure inlining [24]. This section illustrates when cross-library optimization is important and compares the automatic cross-library optimizations with the optimizations performed by the `library-group` form.

Applications with frequent calls across library boundaries will get the most benefit from both of these optimizations. One of the particular use cases for the automatic cross-library optimization is to ensure that record accessors and mutators can be inlined across library boundaries. This extends procedural record optimizations [15] across library boundaries. When a library exports larger procedures or procedures that contain free variables or library references, the automatic cross-library optimization will not be able to perform inlining, where the `library-group` form will. While both of these approaches perform similar tasks, the two are not mutually exclusive. In particular, even applications that use the `library-group` form can benefit from automatic cross-library optimization, when a pre-compiled library is also used as part of the application.

To illustrate when performance gains are expected, we present three example libraries, two written by Eduardo Cavazos and one written specifically to illustrate the differences between automatic cross-library optimization and library groups. All three programs are tested with Chez Scheme Version 8.9.6 [10], targeting both 32- and 64-bit in-

	32-bit			64-bit		
Program	**LG**	**ACLO**	**Both**	**LG**	**ACLO**	**Both**
MPL	27%	22%	25%	19%	16%	19%
Matrix	-7%	0%	-3%	3%	0%	3%
Maze	18%	7%	18%	13%	7%	14%

Table 1. Speedups when using a library group (LG), automatic cross-library optimization (ACLO), or both (Both) when compiling with type-checking enabled.

	32-bit			64-bit		
Program	**LG**	**ACLO**	**Both**	**LG**	**ACLO**	**Both**
MPL	15%	16%	18%	17%	14%	19%
Matrix	4%	0%	4%	-6%	0%	0%
Maze	22%	8%	23%	15%	6%	15%

Table 2. Speedups when using a library group (LG), automatic cross-library optimization (ACLO), or both (Both) when compiling without type-checks.

struction sets on an Intel Core i7 3960X 3.30 GHz 12-core, dual processor machine with 64 GB of RAM. Each benchmark is run nine times. Before each run, two maximum-generation garbage collections are performed to help stabilize the timing measurements. The first run is a warm up and is not timed. The following eight runs are measured, and the standard deviation is computed to determine if speedups are significant.

The first program [6] implements a set of tests for the "Mathematical Pseudo Language" [8, 9] (MPL), a symbolic math library. The second uses a library for indexable sequences [5] to implement a matrix multiply algorithm [11]. The final application is an adaptation of the maze benchmark from the R6RS Benchmarks [7]. In the R6RS benchmarks, this is a single top-level program, originally composed of several files (according to the comments). We separated the maze program into a set of libraries, based on the original file demarcations indicated in the comments. A few function definitions were moved in order to avoid cycles in the dependency graph.

Many small libraries comprise the MPL library. Each basic mathematical function, such as `+`, `/`, and `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 [12], avoids cross-library calls, as it is implemented entirely as a macro. The mathematical libraries, however, make use of operations defined as procedures in other libraries, which cannot be inlined across library boundaries without one of the optimizations described in this paper. The test library also makes many cross-library calls to both the math libraries and the testing support library. Thus, there are many cross-library calls that can be eliminated by using a library group or automatic cross-library optimization. Using a library group alone results in between a 15% and 27% speed-ups over the separately compiled version, depending on machine type and optimization level, see Tables 1 and 2. Because many procedures used across library boundaries are inlinable, the automatic cross-library optimization also optimizes the MPL tests, though not as well, with speed-ups between 14% and 22%, depending on machine type and optimization level, see Tables 1 and 2.

The matrix-multiply example uses a `vector-for-each` form that provides 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 operations it creates, however, are implemented as macros, so operations defined by these libraries are expanded into basic operations at their use sites. Three nested calls to `vector-for-each-with-index` are used in `matrix-multiply`, which expand into inline loops. A test program calls matrix-multiply on 50 x 50, 100 x 100, and 500 x 500 matrices. The calls to the multiply operation are cross-library calls, but the majority of the work of performing matrix multiple occurs entirely in the matrix multiple library, so we do not expect much benefit from cross-library optimizations. Using a library group sometimes results in a small slowdown (between 7% and 3% slower) or a small speed-up (at 3%), depending on machine type and optimize level. The standard deviation for this benchmark is high, however, at around 5%, so the use of the library group form has little significant impact. Enabling automatic cross-library optimization, whether by itself or with a library group, has no impact on this benchmark's performance. Tables 1 and 2 provide the speed-up or slowdown for each machine type.

The maze example is composed of seven libraries used to specify and solve mazes. Similar to the MPL library, there are many calls across library boundaries. Several of the procedures in these libraries are too large to fit within the default size limit, so we do not expect them to be automatically inlinable across library boundaries. Automatic cross-library optimization shows some benefit in this example, but not as much as the library group. It is a good example of an application where the automatic cross-library optimization can get us part of the way to our optimization goals, but cannot provide all of the benefits of a library group. Using a library group results in between a 13% and 22% speed-ups, while automatic cross-library optimization results in only between a 6% and 8% speed-ups, depending on machine type and optimize level. Tables 1 and 2 provide the speed-ups for each machine type and optimize level.

For all three benchmarks, using both a library group and automatic cross-library optimization together results in no significant improvement over a library group alone.

In our example programs, the difference in time between compiling the program as a set of individual libraries with automatic cross-library optimization enabled and as a single library group is negligible.

5. Related Work

The Glasgow Haskell Compiler (GHC) [1] provides support for aggressive cross-module optimization [21], with higher levels of optimization enabling more extensive cross-library optimization. Similar to our automatic cross-library optimization, these optimizations happen automatically. Unlike our automatic cross-library optimization, which stores the library object code and optimization information in a single file, GHC uses separate files to store the module object code and the optimization information. Distributing only the module object code is easier in GHC's model, since it is already separate from the optimization information, however, if the two files are not kept in sync, it can lead to problems. We provide a mechanism for stripping optimization information from a binary library, if there is a desire to distribute only the library object code.

The Standard ML of New Jersey (SML/NJ) compiler supports both separate compilation and cross-module analysis and optimization through two different approaches [3, 19]. The first approach [3] splits functors and higher-order functions into two parts, an *expansive* part that cannot be inlined and an *inlinable* part that contains only constants and side-effect free code. Determining the inlinable code, is similar to our Scheme compiler's determination of copyable constants and externally inlinable procedures. The second approach [19] utilizes type-directed compilation techniques to optimize across higher-order modules. Both approaches have been fully integrated with the SML/NJ compiler.

The OCaml Compiler [2] allows modules to be compiled separately and includes extra information with the compiled modules to allow for cross unit optimizations [18]. Similar to GHC, the OCaml compiler uses separate files to store the module object code and the optimization information.

The automatic cross-library optimization uses information encoded in the binary of a compiled library when compiling a new library. Link-time optimization is a related approach, where optimization takes place when binary libraries are linked. In this approach, compilation has already happened for all of the source code, and it is only the binaries that are used in the optimization. Several different approaches to this technique exist and are beginning to be used in compilers such as GCC [22] and compiler frameworks such as LLVM [17]. 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 [4] link-time optimizer applies similar techniques to optimize cross-module calls when compiling Gambit-C Scheme code into C.

Dynamically linked libraries can also be optimized at link time [13]. Similar to other link-time optimizations, extra information in the shared object file is used to perform some inlining optimizations. Currently, our system does not support any link-time optimizations, however, it is possible we could use similar information to that used for our automatic cross-library optimization.

6. Conclusion

The automatic cross-library optimizations described in this paper provide a reasonable compromise between the complete cross-library optimization possible when using a library group and getting no cross-library optimization when compiling libraries separately. Automatic cross-library optimization does not require any intervention from the programmer, yet can lead to significant improvements in performance, particularly when an application has many calls across library boundaries.

Interprocedural optimizations performed after source optimization, such as closure sharing and elimination [16] are enabled when library groups are used. At present, they are not enabled by automatic cross-library optimization, except in effect, when inlining causes two or more procedures to be combined into one. It would be useful to extend automatic cross-library optimization to support such optimizations.

The requirement to maintain pointer equivalence for non-immediate constants, even though they are immutable, currently prevents propagation of such constants, including strings, pairs, and vectors, etc., across library boundaries, and currently also prevents automatic cross-library inlining of procedures whose bodies, post-optimization, contain such constants. To remedy this, compiler could associate a globally unique identifier with each constant that might be propagated, and the linker could use this to maintain shared structure across compilation units. The complexity and overhead of doing so could be fairly high, considering that each piece of a complex constant, including each pair in the backbone of a list constant, would require its own unique identifier.

A better long-term solution is to change the language to relax the pointer-equivalence requirement for constants to give the implementation more latitude, as it is already given for procedures. In our experience, programmers rarely count on pointer-equivalence of constants, and a programmer always has the option of explicitly allocating a (mutable) object for which maintenance of pointer-equivalence is inherently required.

Acknowledgments

Different parts of Keep's effort on this work were partially supported by the DARPA programs APAC and CRASH. Comments from the reviewers lead to several improvements in the presentation.

References

[1] The Glasgow Haskell Compiler. URL `http://www.haskell.org/ghc/`.

[2] The Caml Language. URL `http://caml.inria.fr/`.

[3] M. Blume and A. W. Appel. Lambda-splitting: a higher-order approach to cross-module optimizations. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 112–124, New York, NY, USA, 1997. ACM. ISBN 0-89791-918-1. . URL `http://doi.acm.org/10.1145/258948.258960`.

[4] D. Boucher. GOld: a link-time optimizer for Scheme. In *Proceedings of the 2000 Workshop on Scheme and Functional Programming*, Scheme '00, 2000.

[5] E. Cavazos. Dharmalab git repository, . URL `http://github.com/dharmatech/dharmalab/tree/master/indexable-sequence/`.

[6] E. Cavazos. MPL git repository, . URL `http://github.com/dharmatech/mpl`.

[7] W. D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

[8] J. S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811586.

[9] J. S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A. K. Peters, Ltd., Natick, MA, USA, 2002. ISBN 1568811594.

[10] R. K. Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

[11] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[12] D. Eddington. Xitomatl bazaar repository. URL `https://code.launchpad.net/~derick-eddington/scheme-libraries/xitomatl`.

[13] W. W. Ho, W.-C. Chang, and L. H. Leung. Optimizing the performance of dynamically-linked programs. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 19–19, Berkeley, CA, USA, 1995. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267411.1267430`.

[14] A. W. Keep and R. K. Dybvig. Enabling cross-library optimization and compile-time error checking in the presence of procedural macros. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming*, Scheme '10, pages 66–76, 2010.

[15] A. W. Keep and R. K. Dybvig. A sufficiently smart compiler for procedural records. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

[16] A. W. Keep, A. Hearn, and R. K. Dybvig. Optimizing closures in O(0) time. In *Proceedings of the 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

[17] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[18] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml System release 4.00: Documenation and User's Guide*. July 2012. URL `http://caml.inria.fr/pub/docs/manual-ocaml`.

[19] Z. Shao. Typed cross-module compilation. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 141–152, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. . URL `http://doi.acm.org/10.1145/289423.289436`.

[20] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19 (Supplement S1):1–301, 2009. . URL `http://www.r6rs.org/`.

[21] The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, version 6.12.1. URL `http://www.haskell.org/ghc/docs/latest/html/users_guide/`.

[22] The GNU Project. Link-Time Optimization in GCC: Requirements and high-level design, November 2005.

[23] The SML/NJ Fellowship. Standard ML of New Jersey. URL `http://www.smlnj.org/`.

[24] O. Waddell and R. K. Dybig. Fast and effective procedure inlining. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 35–52, London, UK, 1997. Springer-Verlag. ISBN 3-540-63468-1.