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

Andrew W. Keep          R. Kent Dybvig

INDIANA UNIVERSITY
SCHOOL OF INFORMATICS AND COMPUTING
Bloomington
Library Groups

Andrew W. Keep           R. Kent Dybvig
Goals

• Cross-library optimizations
• Type checking across library boundaries
• Single binary for multiple libraries
• Unchanged development process
Library Groups

- Explicitly combine libraries
- Optionally add a top-level program
- A new form: library-group
Example

(library (tree)
 (export make-tree ---)
 (import (rnrs))
 (define make-tree ---)
 ---)
Example

(library (tree constants)
 (export quote-tree t0 ---)
 (import (rnrs) (tree))
 (define-syntax quote-tree
  --- (make-tree ---) ---)
 (define t0 (quote-tree))
 ---)
Example

(import (rnrs) (tree) (tree constants))
(define tree->list ---)
(tree->list t₀)
(tree-value (tree-children t2))
(tree->list (quote-tree 5 (7 9)))
Example

(library-group
  (library (tree)
    (export make-tree)
    (import (rnrs))
    (define make-tree)
  )
  (library (tree constants)
    (export quote-tree t0)
    (import (rnrs) (tree))
    (define-syntax quote-tree
      (make-tree)
    )
    (define t0 (quote-tree))
  )
  (import (rnrs) (tree) (tree constants))
  (define tree->list)
  (tree->list t0)
  (tree-value (tree-children t2))
  (tree->list (quote-tree 5 (7 9))))
Example

(library-group
 (include "tree.sls")
 (include "tree/constants.sls")
 (include "app.sps"))
Library Group Syntax

library-group  ->  (library-group  lglib*  lgprog)  
               |  (library-group  lglib*)

lglib  ->  library  |  (include  filename)
lgprog  ->  program  |  (include  filename)
Challenges

- Achieving proper phasing
- Handling cyclic dependencies
- Enabling cross-library optimization/checking
Implementation:
Libraries

• Visit code, invoke code, metadata
• Import dependencies form a DAG
• Invoke code body uses letrec* semantics
Example

(letrec* ([make-tree ---]
   
   (set-top-level! $make-tree make-tree)
   
   ---)

---)
Example

(letrec* ([t0 tree-constant] ___) (set-top-level! $t0 t0) ___)
Example

(letrec* ([tree->list ---])
  (tree->list $t0)
  ($tree-value ($tree-children $t2))
  (tree->list tree-constant))
Implementation: Library Groups

- Combine letrec* expressions
- Preserve existing library exports
- Invoke libraries needed during expansion
(lambda (uid)
  (case uid
    [(tree) (letrec* ([make-tree ___]
                        ___)
            ___)]
    [(constants) (letrec* ([t0 ___]
                            ___)
                  ___)]
    [else (letrec* ([tree->list ___])
               (tree->list $t0)
               ($tree-value
                (car ($tree-children $t2)))
               (tree->list ___))))}
Library Group 1

• Advantages:
  • Single output binary
  • Matches existing library semantics

• Disadvantages:
  • Hinders cross-library optimizations
Library Group 2

(letrec* ([make-tree ----] ----)  
(set-top-level! $make-tree make-tree)  
---  
(letrec* ([t0 tree-constant] ----)  
(set-top-level! $t0 t0)  
---  
(letrec* ([tree->list ----])  
(tree->list $t0)  
($tree-value  
  (car ($tree-children $t2))))  
(tree->list tree-constant)))
Library Group 2

• Advantages:
  • Creates a single invoke code
  • Allows optimizations and checking

• Disadvantage:
  • Causes dependency problems
Dependency Problems

(A) ➔ (B) ➔ (C)
Dependency Problems

(A) → (B) → (C)
Dependency Problems

(A) → (B) → (C)
(lambda (uid)
  (letrec* ([make-tree ---] ----)
    ----
    (mark-invoked! 'tree)
    (let ([nested-lib
           (lambda (uid)
             (letrec* ([t0 ----] ----)
               ----
               (mark-invoked! 'constants)
               (let ([nested-lib program code])
                 (if (eq? uid 'constants)
                   nested-lib
                   (nested-lib uid)))]))]
      (if (eq? uid 'tree)
        nested-lib
        (nested-lib uid)))))

Library Group 3

- Advantages:
  - Avoids synthetic cycles
  - Allows optimization and checking
  - Single output binary
Caveat: Dynamic Dependencies

- Arises from use of `eval` in init expressions
- Library groups allow explicit ordering
- Work arounds
  - Transform into import dependency
  - Move into initialization function
Fixing Dynamic Dependencies

• Start with case-based library–group
• Lift “simple” letrec* bindings
• Requires letrec* style optimization
Library Phasing

- Retain phasing between libraries in group
- Cannot simply recompile from source
- Relatively straightforward solution
Summary

- Library groups meet our goals:
  - Cross-library optimization
  - Type checking across library boundaries
  - Single output binary
  - Maintains proper phasing order
  - Avoids synthetic import dependency cycles
Thanks

Questions?