Enabling Optimizations through Demodularization

Blake Johnson

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Christophe Giraud-Carrier
Quinn Snell

Department of Computer Science

Brigham Young University

December 2015

ABSTRACT

Enabling Optimizations through Demodularization

Blake Johnson
Department of Computer Science, BYU
Master of Science

Programmers want to write modular programs to increase maintainability and create abstractions, but modularity hampers optimizations, especially when modules are compiled separately or written in different languages. In languages with syntactic extension capabilities, each module in a program can be written in a separate language, and the module system must ensure that the modules interoperate correctly. In Racket, the module system ensures this by separating module code into phases for runtime and compile-time and allowing phased imports and exports inside modules. We present an algorithm, called demodularization, that combines all executable code from a phased modular program into a single module that can then be optimized as a whole program. The demodularized programs have the same behavior as their modular counterparts but are easier to optimize. We show that programs maintain their meaning through an operational semantics of the demodularization process and verify that performance increases by comparing modular Racket programs to the equivalent demodularized and optimized programs. We use the existing Racket optimizer to optimize the demodularized programs by decompiling them into an intermediate form that the optimizer uses. We also demonstrate a dead code elimination optimization that dramatically reduces the file size of demodularized Racket programs.

Keywords: macros, Racket, modules, optimization

ACKNOWLEDGMENTS

Jay McCarthy, Matthew Flatt, Eric Mercer

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Programmers should not have to sacrifice the software engineering goals of modular design and good abstractions for performance. Instead, their tools should make running a well-designed program as efficient as possible.

Many languages provide features for creating modular programs which enable separate compilation and module reuse. Some languages provide expressive macro systems, which enable programmers to extend the compiler in arbitrary ways. Combining module systems with expressive macro systems allow programmers to write modular programs with each module written in its own domain-specific language. A compiler for such a language must ensure that modular programs have the same meaning independent of the order in which the modules are compiled. A phased module system, like the one described by Flatt for Racket [7], is a way to allow both separately compiled modules and expressive macros in a language.

Modular programs are difficult to optimize because the compiler has little to no information about values that come from other modules when compiling a single module. Existing optimizations have even less information when modules can extend the compiler. Good abstractions are meant to obscure internal implementations so that it is easier for programmers to reason about their programs, but this obscurity also limits information available for optimizations. In contrast, non-modular programs are simpler to optimize because the compiler has information about every value in the program. For example, inlining a function in a non-modular program is trivial because every use of the function is known at compile-time.

Some languages avoid the problem of optimizing modular programs by not allowing modules, while others do optimizations at link time, and others use inlining. Not allowing modules defeats the benefits of modular design. Languages such as C and Scheme that do not have well developed module systems make it difficult to manage large programs written in them. Link time optimizations can be too low level to do useful optimizations. Linking is usually performed on already compiled modules, so any optimizations are performed on machine code and are missing source-level information. Inlining must be heuristic-based, and good heuristics are hard to develop. Some work has been done on these heuristics for the ML programming language [2].

Our solution for optimizing modular programs, called demodularization, is to transform a modular program into a non-modular program by combining all runtime code and data in the program into a single module. In a phased module system, finding all of the runtime values is not trivial. Phased module systems allow programmers to refer to the same module while writing compiler extensions and while writing normal programs. A demodularized program does not need to include modules that are only needed during compile-time, but whether or not the module is needed only at compile-time is not obvious from just examining the module in isolation.

A program with a single module is effectively a non-modular program. After demodularization, a program becomes a single module, so existing optimizers have more information. Also, demodularization enables new optimizations that need whole program information. Demodularizing programs in a language with an expressive macro system is feasible and useful.

We explain the Racket module system in Chapter 2. In Chapter 3, we explain demodularization at a high level with a detailed example. Next, we use the operational semantics model of the demodularization process to explain why demodularization is correct (Chapter 4), and we then describe an actual implementation for Racket (Chapter 5), followed by experimental results of demodularizing and optimizing real-world Racket programs (Chap-

ter 6). The operational semantics model removes the unnecessary details of the full implementation so the demodularization process is easier to understand and verify. The actual implementation presents interesting difficulties that the model does not. The experimental results show that demodularization improves performance, especially when a program is highly modular.

Chapter 2

The Racket Module System

2.1 The Racket Programming Language

The Racket Programming Language is a platform for creating powerful abstractions. These abstractions are created through the use of Racket's macro and module systems. The macro system, a heritage from LISP [11] and Scheme [10], allows programmers to add new features to their programs in an integrated way. The module system allows programmers to separate their programs into logical parts and hide implementation details. Together, they allow programmers to create Languages as Libraries [13] that are suitable for specific tasks.

2.2 Macros

Macros are the main way programmers add new features to Racket. Esentially, macros are functions (known as transformers) whose domain and range are syntax objects. Syntax objects are data structures that contain the raw syntax of a program, along with lexical information and other properties associated with the syntax. If a programmer needs the power, they can write transformers using all of the features of Racket, as long as the function takes in and produces syntax objects. Racket provides pattern-matching for syntax objects that makes writing macros simpler when that is all that a programmer needs.

The `define-syntax` form is used to identify a macro definition that is a funtion from syntax to syntax. The helper function, `syntax-case` matches syntax patterns so that it is easy to use them in the output of the macro.

Listing 1 shows how to use one of these macros to write a `while` loop. The macro

```
(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     #'(begin
         (define (while-loop)
           (when test
             body ...
             (while-loop)))
         (while-loop))]))
```

Listing 1: a macro implementation of a `while` loop

turns a `while` loop into a recursive function and a call to that recursive function. Whatever syntax used by the programmer for the `test` and `body ...` forms will be substituted into the output of the macro. Listing 2 shows a use of the while macro and what it expands into after running the macro. Racket's macros are hygienic [8], which means that identifiers created

```
(while (< x 10)
  (printf "x is ~a\n" x)
  (set! x (add1 x)))

=>

(begin
  (define (while-loop)
    (when (< x 10)
      (printf "x is ~a\n" x)
      (set! x (add1 x)
      (while-loop))))
  (while-loop))
```

Listing 2: use and expansion of a `while` loop

from programs will not clash with identifiers created from macros. In this example that means that the Racket program using a while loop will not be able to call the `while-loop` function that the macro creates, and that if the program has another definition for `while-loop`, it will be independent from the definition introduced by the macro. Also, each use of the macro will have distinct definitions for `while-loop` so that the macro can be nested and used multiple times.

2.3 Modules

Racket modules are a way of grouping definitions, expressions, and macros into groups. Modules also allow control over imports and exports, using the Racket forms **require** and **provide** respectively. Listing 3 shows a simple Racket module and some of the features of the Racket module system. The `counter` module contains a variable definition and functions

```
#lang racket/base
(provide get-counter-val count-up)

(define x 0)
(define (get-counter-val) x)
(define (count-up . v) (set! x (add1 x)))
```

Listing 3: `counter.rkt`: A simple Racket module implementing a counter

for getting and incrementing that variable. The module only exports the functions, so the module encapsulates the variable. If a programmer wanted to use this module, they would import it using **require** as shown in Listing 4.

```
#lang racket/base
(require "while-lang.rkt")
(require "counter.rkt")

(while (< (get-counter-val) 4)
  (printf "loop runtime val is ~a\n" (get-counter-val))
  (count-up))
```

Listing 4: `while-test.rkt`: A Racket module that uses other modules

It is also possible to use modules to help write macros. For example, if a programmer wanted to change the `while` macro so that it reported how many expressions were in each while loop, they could use the `counter` module inside the `while` definition as seen in Listing 5. It is possible to add code to the `while` macro because it is just a function from syntax to syntax that runs at compile-time. Also, the **require** form includes a **for-syntax** form to indicate that the `counter` module is needed at compile-time (along with the `racket/base`

```
#lang racket/base
(require (for-syntax racket/base
                     "counter.rkt"))
(provide while)

(define-syntax (while stx)
  (syntax-case stx ()
    [(while test body ...)
     (begin
       (for-each count-up
                 (syntax->list #'(body ...)))
       (printf "while loop contains ~a expressions\n"
               (get-counter-val))
       #'(begin
           (define (while-loop)
             (when test
               body ...
               (while-loop)))
           (while-loop)))]))
```

Listing 5: `while-lang.rkt`: A Racket module implementing a language with `while` loops

module for `printf`). Now this module can be used to extend the language of other programs by adding a `while` loop, although it is just a library that is included like any other library.

2.4 Languages

The ability to use modules as languages allows programmers to write specialized domain-specific languages (DSLs) to better express solutions to domain problems. Using this ability, programmers have added languages for Typed Racket [12], Object-Oriented Racket [6], Logic Programming [9], and more. Also, because they are modules, it is possible to use them all in the same program and use the right language for each specific problem. The language creating facilities of Racket extend to more than just collections of functions and macros; they also allow changing the parser and changing the meaning of things like function application syntax.

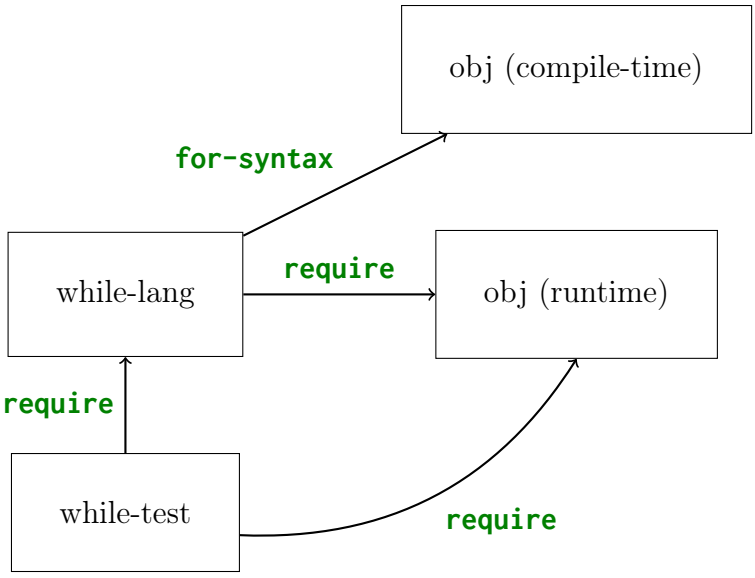

Figure 2.1: Module dependencies for the `while-test` program

2.5 Separate Compiliation

In the presence of macros that can run arbitrary code, compilation of a single module becomes complicated. Compilation and execution of code has to be interleaved in order to produce the correct compiled program. All of the listings in the previous section form a program with the `while-test` module as the main module. Figure 2.1 shows the relationship between the different modules in the program. To compile the program, both the `while-lang` and `counter` need to be compiled before `while-test`, but also the `counter` module needs to be compiled and ready to use while compiling `while-test` because the counter functions are used inside the `while` macro. While compiling `while-test`, the `while` macro will be expanded and code from the `counter` module will run. The value of the counter is not shared between compiling `while-test` and running it.

2.6 Phases

The way the Racket module system allows for reliable separate compiliation is by separating compilation and execution into phases. Phase 0 is the execution phase of the main module of a program, what could be considered running the program. Phase 1 is the compilation

phase of the main module, which could include execution of code inside macros. Phase 2 is the compiliation phase of any modules that need to execute during phase 1, and higher phases are used as needed to compile and run code used in lower phases. It is possible for a single module to be used in multiple phases. Each phase will have a separate instantiation of the module. It is possible for modules to reference modules at higher phases by using the `for-syntax` form as seen above, and it is also possible to reference modules at lower phases by using the `for-template` form. The `for-template` form is used when a programmer wants to include code from an imported module in the output of a macro.

In the example above, the `while-test` module is compiled at phase 1, which triggers compilation of the `counter` module at phase 2, so that the `counter` module can be instantiated at phase 1 to count how many expressions are in the `while` loop. The phase 1 instantiation of the `counter` module is separate from the phase 0 instantiation of the module. The separate compilation guarantee of the Racket module system is that "Any effects of the instantiation of the module's phase 1 due to compilation on the Racket runtime system are discarded" [1].

2.7 Program Evaluation

Running a separately compiled Racket program involves following all of the `require` forms and running all phase 0 code in the order in which the code is imported. It is possible that phase 0 code will come through a module that is required by using `for-syntax` if somewhere along the line code is then required using `for-template`.

By understanding how Racket programs are compiled and evaluated, it is apparent that only phase 0 code is necessary to run the program. This is the basis for how demodularization can recover whole programs from separately compiled Racket modules.

Chapter 3

Intuition

Demodularization is the process of collecting all phase 0 code required by a program into a single module. This is done by tracing through the `require` graph starting at a program's main module. The following example program illustrates the need for demodularization and how it is done.

3.1 Example Program

A programmer wants to use a queue library where the library uses different backing structures depending on the length of the queue. Listing 6 shows an example of using such a library. The library is implemented as a macro, shown in Listing 7, that switches between two

```
#lang racket/base
(require "queue.rkt")

(with-queue (1 2 3 4 5 6)
  (enqueue 4)
  (displayln (dequeue))
  (displayln (dequeue)))
```

Listing 6: `main.rkt` module with queue usage

implementatations depending on the length of the initial queue. Listing 8 and Listing 9 show the two queue implementations, with the `long-queue` using vectors and the `short-queue` using lists. The `...` notation has meaning in Racket, so `....` is used to indicate elided code.

```
#lang racket/base
(require (for-syntax racket/base
                     racket/syntax)
         "short-queue.rkt"
         "long-queue.rkt")

(define-syntax (with-queue stx)
  (syntax-case stx ()
    [(with-queue (v ...) e ...)
     (begin
       (define type
         (if (> (length (syntax->list #'(v ...))) 5)
             'long
             'short))
       (define make-queue
         (format-id #'stx "make-~a-queue" type))
       (define enqueue (format-id #'stx "~a-enqueue" type))
       (define dequeue (format-id #'stx "~a-dequeue" type))
       #`(let ([q (#,make-queue v ...)])
           (define (#,(datum->syntax stx 'dequeue))
             (#,dequeue q))
           (define (#,(datum->syntax stx 'enqueue) x)
             (#,enqueue q x))
           e ...))]))

(provide with-queue)
```

Listing 7: `queue.rkt` module

3.2 Compilation

Compiling this program involves expanding the `with-queue` macro, which will turn the program into Listing 10.

The expanded program contains references to `long-queue` operations because the length of the initial queue is over 5. Although the expanded program is not fully compiled to bytecode, it can stand in for a fully compiled program with regard to optimization and demodularization.

With the expanded program, it is possible to see how cross-module optimizations would be difficult. The program has calls to functions in the `long-queue` module, so the op-

```
#lang racket/base
(define (make-long-queue . vs)
  .... make-vector ....)

(define (long-enqueue q v)
  .... vector-set! ....)

(define (long-dequeue q)
  .... vector-ref ....)

(provide (all-defined-out))
```

Listing 8: `long-queue.rkt` module

```
#lang racket/base
(define (make-short-queue . vs)
  .... list ....)

(define (short-enqueue q v)
  .... cons ....)

(define (short-dequeue q)
  .... list-ref ....)

(provide (all-defined-out))
```

Listing 9: `short-queue.rkt` module

timizer does not have access to the definitions of the functions while finishing the compilation of the main module.

3.3 Demodularization

Demodularization of the program proceeds by following the **`require`** graph and including all phase 0 code from the required modules into the main module of the program. The `main` module requires the `queue` module, but the `queue` module only has phase 1 code, so it is not included in the final module. The `queue` module requires the `long-queue` and `short-queue` modules, which have various phase 0 definitions that get included in the order they appear in their original modules. Listing 11 shows what the demodularized example program would

```
(module main racket/base
  (#%module-begin
   (require "queue.rkt")
   (let ((q (make-long-queue 1 2 3 4 5 6)))
     (define (dequeue) (long-dequeue q))
     (define (enqueue x) (long-enqueue q x))
     (enqueue 4)
     (displayln (dequeue))
     (displayln (dequeue)))))
```

Listing 10: `main.rkt` module after macro expansion

look like.

The demodularized program should have the exact same behavior at runtime as the modular program. At this point, the only thing that changes is that the Racket runtime does not need to follow the **`require`** graph because all phase 0 definitions and expressions are in a single module. Also, both queue implementations appear in the demodularized program even though only one of them is used.

3.4 Optimization

After a program has been demodularized, the optimizer now has all the information it needs to do optimizations on the whole program. Listing 12 shows what the example program would look like after optimization.

The existing optimizer is able to inline the various functions because it has access to all of the definitions within the module. The optimizer also removes the extra queue implementation because it is dead code. Therefore, demodularization enables optimization by providing the optimizer access to the whole program in one place.

```
(module main racket/base
  (#%module-begin
   (define (make-short-queue . vs)
    .... list ....)

   (define (short-enqueue q v)
    .... cons ....)

   (define (short-dequeue q)
    .... list-ref ....)

   (define (make-long-queue . vs)
    .... make-vector ....)

   (define (long-enqueue q v)
    .... vector-set! ....)

   (define (long-dequeue q)
    .... vector-ref ....)

   (let ((q (make-long-queue 1 2 3 4 5 6)))
     (define (dequeue) (long-dequeue q))
     (define (enqueue x) (long-enqueue q x))
     (enqueue 4)
     (displayln (dequeue))
     (displayln (dequeue)))))
```

Listing 11: `main.rkt` module after demodularization

```
(module main racket/base
  (#%module-begin
   (let ((q (.... make-vector .... 1 2 3 4 5 6 ....)))
     (.... vector-set! .... 4 ....)
     (displayln (.... vector-ref ....))
     (displayln (.... vector-ref ....)))))
```

Listing 12: `main.rkt` module after optimization

Chapter 4

Model

We can understand the specifics of demodularization by describing it as an algorithm for a simple language with a well defined semantics. The simple language models the existing Racket module system. We describe a small-step operational semantics (a series of syntactic rules) that define how the language works. Next, we describe how the demodularization algorithm works as a metafunction over the language. Finally, we show that programs before and after demodularization produce the same results.

4.1 A Module Language

The *mod* language (Figure 4.1) contains only the features necessary to write modular programs where it is possible to observe the effects of module evaluation order.

A program in *mod* consists of a list of modules that can refer to each other. Each module has a name, any number of imports, any number of definitions, and sequenced code expressions. All definitions in a module are exposed as exports to other modules, but to use definitions from another module, the program must import it through a `require` expression. Both `require` and `define` expressions have phase annotations; this simulates the interactions between modules in a language with macros without requiring a model of macro expansion. The language includes variable references, numbers, addition, and mutation. Mutation makes module evaluation order observable, and addition represents the work that a module does. In addition to numbers and variables, there are two special forms of values and references that model the interaction of macros with the module system. A `quote` expression is like a

```
program ::= (mod ...)
    mod ::= (module id (req ...) (def ...) expr)
    req ::= (require id @ phase)
   expr ::= var
          | val
          | (+ expr expr)
          | (set! var expr)
          | (begin expr expr)
    val ::= number
          | (quote id @ phase)
    var ::= id
          | (ref id @ phase)
     id ::= variable-not-otherwise-mentioned
  phase ::= number
    def ::= (define id @ phase as expr)
```

Figure 4.1: *mod* language grammar

reference to syntax at runtime. A `ref` expression is like a macro that can only do one thing: refer to a variable at a phase.

4.2 Compilation

We have to compile *mod* programs before demodularizing them, just like in the Racket implementation. In Racket, compiling expands all macros in a program and changes definitions and variable references to refer to memory locations. In *mod*, compiling eliminates `ref` expressions, turns definitions into `set!` expressions, changes variable references to include module information, and sorts code into phases. Compilation in both cases still leaves behind a relatively high-level language, but the language is free of syntactic extensions. This is important for demodularization because otherwise macro expansion would have to be part of the algorithm, which would complicate it and possibly duplicate work. The grammar in Figure 4.2 specifies the compiled language for *mod*.

The grammar no longer has definitions, all variables now include module references, and code is sorted into phases. The actual compilation function is not relevant to demodularization.

```
program ::= (mod ...)
    mod ::= (module id (req ...) (code ...))
   code ::= (phase expr)
    req ::= (require id @ phase)
   expr ::= var
          | val
          | (+ expr expr)
          | (set! var expr)
          | (begin expr expr)
    val ::= number
          | (quote id @ phase)
    var ::= (id id)
     id ::= variable-not-otherwise-mentioned
  phase ::= number
```

Figure 4.2: Compiled language grammar

4.3 Evaluation

We evaluate the compiled language using a small-step reduction semantics. Because the reduction rules are syntactic, we extend the compiled language further with evaluation contexts, a heap representation, and a stack representation to keep track of the order to instantiate modules. These extensions are in Figure 4.3. An expression of the form:

$$(\sigma \ / \ program \ / \ ((id \ phase) \ \ldots) \ / \ ((id \ phase) \ \ldots))$$

represents the state of the machine during evaluation. σ represents the heap of the program, and when evaluation finishes represents the output of the program. The list of modules is the code of program in the compiled language. The first list of *(id phase)* pairs is the list of modules to evaluate, and the second list is the modules that have already been evaluated.

```
     E ::= []
         | (+ E expr)
         | (+ val E)
         | (set! var E)
         | (begin E expr)
     σ ::= ([var val] ...)
 state ::= (program / (inst ...) / (inst ...))
  inst ::= (id phase)
    st ::= (σ / state)
```

Figure 4.3: Extensions to compiled language grammar

The reduction rules in Figures **??**–**??** evaluate a compiled program that starts with an empty heap, the program code, a stack that contains the identifier of the main module at phase 0, and an empty completed module list.

$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ ((\textsf{require}\ id_{new}\ @\ phase_{new})\ req \ldots)\ (code \ldots))\ mod_n \ldots) \mathbin{/} ((id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))) \longrightarrow$$
$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ (req \ldots)\ (code \ldots))\ mod_n \ldots) \mathbin{/} ((id_{new}\ (\ominus\ phase_0\ \ phase_{new}))\ \ (id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))) \quad (4.1)$$

The *module require* rule in Equation 4.1 matches a program with a `require` expression in the module at the top of the evaluation stack and evaluates it by removing the `require` expression from the module and pushing the required module onto the evaluation stack with the phase shifted appropriately. The current module is still on the stack and will continue evaluating after the required module is done evaluating. The subsequent rules all apply only when the phase relative to the main module is zero.

$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \mathbin{/} ((id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))[var]) \longrightarrow$$
$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \mathbin{/} ((id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))[val])$$
$$\text{where } val = \textsf{lookup}\ [\![\sigma, var]\!] \quad (4.2)$$

The *var ref* rule in Equation 4.2 looks up a variable in the heap and replaces the variable with its current value. The lookup function is a simple list lookup function that matches the variable name with its occurence in the heap.

$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \mathbin{/} ((id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))[(+\ number_0\ number_1)]) \longrightarrow$$
$$(\sigma \mathbin{/} ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \mathbin{/} ((id_0\ phase_0)\ inst_n \ldots) \mathbin{/} (inst_d \ldots))[(\oplus\ number_0\ \ number_1)]) \quad (4.3)$$

The *add* rule in Equation 4.3 replaces an addition expression of numbers with the result of computing their sum. The rule uses the standard Racket implementation of plus to compute the sum.

$$(\sigma_0 \,/\, ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \,/\, ((id_0\ phase_0)\ inst_n \ldots) \,/\, (inst_d \ldots))[(\textsf{set!}\ var\ val)]) \longrightarrow$$
$$(\sigma_1 \,/\, ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \,/\, ((id_0\ phase_0)\ inst_n \ldots) \,/\, (inst_d \ldots))[val])$$
$$\text{where } \sigma_1 = \textsf{assign}\ [\![\sigma_0, var, val]\!] \tag{4.4}$$

The *set!* rule in Equation 4.4 installs a value for a variable into the heap and reduces to the value. The assign function either replaces the existing value for a variable in the heap or installs a new entry in the heap if it does not exist already.

$$(\sigma \,/\, ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \,/\, ((id_0\ phase_0)\ inst_n \ldots) \,/\, (inst_d \ldots))[(\textsf{begin}\ val\ expr)]) \longrightarrow$$
$$(\sigma \,/\, ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ E)\ code_n \ldots))\ mod_n \ldots) \,/\, ((id_0\ phase_0)\ inst_n \ldots) \,/\, (inst_d \ldots))[expr]) \tag{4.5}$$

When an expression is a value, the *expression done* rule in Equation 4.5 matches and removes the expression from the module.

$$(\sigma \,/\, ((mod_0 \ldots (\textsf{module}\ id_0\ ()\ (code_0 \ldots (phase_0\ val)\ code_n \ldots))\ mod_n \ldots) \,/\, ((id_0\ phase_0)\ inst_n \ldots) \,/\, (inst_d \ldots))) \longrightarrow$$
$$(\sigma \,/\, ((mod_0 \ldots\ mod_n \ldots) \,/\, (inst_n \ldots) \,/\, ((id_0\ phase_0)\ inst_d \ldots))) \tag{4.6}$$

When there are no more expressions left in a module, the *module done* rule in Equation 4.6 applies by removing the module from the program and placing a reference to it in the list of finished modules.

$$(\sigma \,/\, ((mod \ldots) \,/\, (inst_0\ inst_n \ldots) \,/\, (inst_{d0} \ldots\ inst_0\ inst_{dn} \ldots))) \longrightarrow$$
$$(\sigma \,/\, ((mod \ldots) \,/\, (inst_n \ldots) \,/\, (inst_{d0} \ldots\ inst_0\ inst_{dn} \ldots))) \tag{4.7}$$

The *module done already* rule in Equation 4.7 applies when the current module on the stack is in the finished list, so that modules are not evaluated multiple times.

4.4 Demodularization

Figures 4.4–4.7 shows the demodularization algorithm for the compiled language. The algorithm takes as input an *id* specifying the main module of the program, a working list of phased requires, and a list of modules that make up the program.

$$\mathsf{demod}\,[\![\, id, (), (mod_0 \ldots (\mathsf{module}\ id\ ()\ (code_0 \ldots (0\ expr)\ code_n \ldots))\ mod_n \ldots)]\!] = ((\mathsf{module}\ id\ ()\ ((0\ expr))))$$

Figure 4.4: Demodularization algorithm

The first rule in Figure 4.4 applies when the main module has no requires left and there are no requires in the working list, meaning the algorithm can terminate with just the phase 0 code of the main module remaining. The second and third rules in Figure 4.5

$$\begin{aligned}
&\mathsf{demod}\,[\![\, id, (), ((\mathsf{module}\ id\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_m \ldots)\ (code_{m0} \ldots (0\ expr_m)\ code_{mn} \ldots)) \\
&\qquad (\mathsf{module}\ id_r\ (req_r \ldots)\ (code_{r0} \ldots (phase_{nr}\ expr_r)\ code_{rn} \ldots)) \\
&\qquad\quad mod_n \ldots)]\!] \\
&\quad = \mathsf{demod}\,[\![\, id, ((id_r\ (\text{-}\ phase_r))), \\
&\qquad ((\mathsf{module}\ id\ (req_m \ldots)\ (code_{m0} \ldots (0\ (\mathsf{begin}\ expr_r\ expr_m))\ code_{mn} \ldots)) \\
&\qquad\ (\mathsf{module}\ id_r\ (req_r \ldots)\ (code_{r0} \ldots (phase_{nr}\ expr_r)\ code_{rn} \ldots)) \\
&\qquad\quad mod_n \ldots)]\!]
\end{aligned}$$

where $phase_{nr} = (\text{-}\ phase_r)$

$$\begin{aligned}
&\mathsf{demod}\,[\![\, id, (), ((\mathsf{module}\ id\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_m \ldots)\ (code_m \ldots)) \\
&\qquad (\mathsf{module}\ id_r\ (req_r \ldots)\ (code_r \ldots)) \\
&\qquad mod_n \ldots)]\!] \\
&\quad = \mathsf{demod}\,[\![\, id, ((id_r\ (\text{-}\ phase_r))), \\
&\qquad ((\mathsf{module}\ id\ (req_m \ldots)\ (code_m \ldots)) \\
&\qquad\ (\mathsf{module}\ id_r\ (req_r \ldots)\ (code_r \ldots)) \\
&\qquad\quad mod_n \ldots)]\!]
\end{aligned}$$

Figure 4.5: Demodularization algorithm

apply when the main module requires the next module in the module list. Both rules add the required module's *id* to the working list of required modules so that the algorithm will follow the complete require graph. The second rule handles the case where the required module has code that will be at phase 0 for the main module and puts that code before the existing phase 0 code of the main module. The next three rules in Figure 4.6 apply when handling the working list of required modules. The first of the three is similar to the second rule because it extracts code that will be in phase 0 of the main module and inserts it into the main module. The second rule of the three working list rules handles the case where there is no matching code for phase 0. The third rule removes an entry from the working list when the module has no more requires.

The final four rules in Figure 4.7 rearrange the program's module list so that modules that require each other are adjacent in the list and the other rules can apply.

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_{m0}\ ...\ (0\ expr_m)\ code_{mn}\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_c\ ...)\ (code_c\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_{r0}\ ...\ (phase_c\ expr_r)\ code_{rn}\ ...))\\
&\quad\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, ((id_r\ (\text{-}\ phase_c\ \ phase_r))\ \ (id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_{m0}\ ...\ (0\ (\mathsf{begin}\ expr_r\ expr_m))\ code_{mn}\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ (req_c\ ...)\ (code_c\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_{r0}\ ...\ (phase_c\ expr_r)\ code_{rn}\ ...))\\
&\quad\ mod_n\ ...)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_c\ ...)\ (code_c\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\\
&\quad\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, ((id_r\ (\text{-}\ phase_c\ \ phase_r))\ \ (id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ (req_c\ ...)\ (code_c\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\\
&\quad\ mod_n\ ...)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ()\ (code_c\ ...))\\
&\quad\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, ((id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ()\ (code_c\ ...))\\
&\quad\ mod_n\ ...)]\!]
\end{aligned}
$$

Figure 4.6: Demodularization algorithm

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, (), (mod_0\ ...\ (\mathsf{module}\ id\ (req\ ...)\ (code\ ...))\ mod_n\ ...)]\!] \\
&\quad = \mathsf{demod}[\![\, id, (), ((\mathsf{module}\ id\ (req\ ...)\ (code\ ...))\ mod_0\ ...\ mod_n\ ...)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, (), ((\mathsf{module}\ id\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_m\ ...)\ (code_m\ ...))\\
&\quad\ mod_i\ ...\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, (), ((\mathsf{module}\ id\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\ mod_i\ ...\ mod_n\ ...)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ mod_i\ ...\\
&\quad\ (\mathsf{module}\ id_c\ (req_c\ ...)\ (code_c\ ...))\\
&\quad\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ (req_c\ ...)\ (code_c\ ...))\\
&\quad\ mod_i\ ...\\
&\quad\ mod_n\ ...)]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_c\ ...)\ (code_c\ ...))\\
&\quad\ mod_i\ ...\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\\
&\quad\ mod_n\ ...)]\!] \\
&= \mathsf{demod}[\![\, id, ((id_c\ phase_c)\ (id_n\ phase_n)\ ...),\\
&\quad ((\mathsf{module}\ id\ (req_m\ ...)\ (code_m\ ...))\\
&\quad\ (\mathsf{module}\ id_c\ ((\mathsf{require}\ id_r\ @\ phase_r)\ req_c\ ...)\ (code_c\ ...))\\
&\quad\ (\mathsf{module}\ id_r\ (req_r\ ...)\ (code_r\ ...))\\
&\quad\ mod_i\ ...\\
&\quad\ mod_n\ ...)]\!]
\end{aligned}
$$

Figure 4.7: Demodularization algorithm

4.5 Equivalence

We claim that the programs will evaluate to the same answers before and after demodularization.

Theorem. *Evaluating a program P a number of steps n and the demodularized program P' a number of steps m will be bisimilar with respect to the stores of the programs as shown in Figure 4.8.*

$$
\begin{array}{ccc}
P & \longleftrightarrow & P' \\
{\scriptstyle n}\downarrow & & \downarrow{\scriptstyle m} \\
\hat{P} & \longleftrightarrow & \hat{P'}
\end{array}
$$

Figure 4.8: Bisimulation of a program before and after demodularization

Proof. The proof is by induction on the DAG of required modules in a program. For each shape of possible requires in a program, we would show the evaluation before and after demodularization end up running the same module code in the same order and result in the same stores. □

The full proof of the theorem would not be instructive because of the complexities of the implementation of the model and the demodularization algorithm.

Chapter 5

Implementation

The implementation of demodularization for the Racket module system is included in the Racket distribution as part of the build tool `raco`. Users can pass in a module they would like to demodularize and optmize to the tool and it will compile all the necessary modules, run the demodularization algorithm on them, decompile the resulting module, and run optimizations on it to produce the final module. The demodularization algorithm runs on compiled modules, which are stored in Racket bytecode files.

5.1 Racket bytecode

Racket bytecode is different from most other forms of bytecode in that it mostly maintains the structure of the abstract syntax tree from the original program. The main aspect that changes between fully-expanded Racket programs and Racket bytecode is the use of stack positions instead of variables and the addition of stack-manipulating instructions. A full explanation of Racket bytecode and what it means can be found in **??**.

A table is created in the bytecode for the top level bindings in a module, which is known as the prefix. All top level bindings have an entry in the prefix, including bindings that come from other modules, and any use of a top level binding in a program is replaced with a numeric reference to the entry. Listing 13 shows a simple program written in the most basic language that Racket supports: the `kernel` language. When this code is compiled, it turns into the bytecode in Listing 14.

In the bytecode, all references to the variable `y` have been replaced with references

```
(module hello '#%kernel
  (#%require racket/private/misc)
  (define-values (y) (random))
  (let-values ([(x) #f])
    (set! x 5)
    (displayln x)
    (displayln (+ x y))))
```

Listing 13: Example program written in `kernel` language

```
(module hello
  (prefix (y (module-variable racket/private/misc displayln 7)))
  (requires #%kernel racket/private/misc)
  (def-values (toplevel 0)
    (app (primval 273)))
  (let-one
   (box-env
    (install-value 0 5
      (seq
       (app (toplevel 1) (local 1))
       (app (toplevel 1)
            (app (primval 247) (local 3) (toplevel 0))))))))
```

Listing 14: Bytecode representation of program from Listing 13

to the prefix in the form of `(toplevel 0)`. All references to `displayln` have also been replaced with `(toplevel 1)` references, which in turn references element `7` of the prefix of the `racket/private/misc` module. The references to `x` are now all different because the stack changes between each usage of `x`. The references to `random` and `+` are replaced with references to primitive implementations in the Racket runtime.

5.2 Demodularization

The implementation of demodularization for the Racket module system is written in Racket and consumes and produces Racket bytecode. The library for reading and writing Racket bytecode in Racket was mainly used for debugging purposes and was incomplete, so the first task was to fully implement the Racket bytecode library.

The actual algorithm for demodularization is similar to the model in the previous chapter in that it traces requires and includes all phase 0 code in the final module, but it also has to deal with the module prefix and references to the prefix. When the algorithm includes a required module, it also includes that module's prefix appended to the end of the main module's prefix. Then, it must adjust all references in that module's code to point at the new combined prefix. Also, the algorithm tracks cross-module references and rewrites them to point to the new prefix as well. In the example in Listing 14, when the algorithm includes the module `racket/private/misc` it must rewrite the reference to `(toplevel 1)` to whatever the new position for `displayln` is in the combined prefix.

5.3 Optimization

For demodularization to be useful, the program needs to be optimized after demodularizing it. To optmize the demodularized bytecode, we wanted the existing Racket optimizer built into the Racket compiler so that we get all existing optimizations "for free" and any new optimizations in the future will work on both regularly compiled and demodularized programs. The existing optimizer works on an intermediate form between fully-expanded Racket code and Racket bytecode. This intermediate form only exists as C data structures in the implementation of Racket. Therefore, to use the existing optimizer, Racket bytecode needed to be decompiled into this intermediate form.

5.4 Decompilation

The decompiler was written in C, so that it could interoperate with the optimizer. There are three main differences between Racket bytecode and the intermediate form: stack positions, cyclic closures, and reference arguments. In Racket bytecode and in the intermediate form, all references to bindings are numeric, but in bytecode the references are stack positions and in the intermediate form the references are lexical. For example, the bytecode in Listing **??** has three references to `x`, in the form of `(install-value 0 5)`, `(local 1)`, and `(local 3)`.

In the intermediate form, all of these references to `x` should be `0` because lexically they all refer to the same variable that is the closest binding.

Racket bytecode allows for cyclic closures, or closures which contain a reference to themselves. Nothing like this exists in the intermediate form, so to decompile cyclic closures, the decompiler creates a new top level definition for the closure and replaces references to the closure (including the self-references) with references to the top level definition.

Finally, Racket bytecode allows reference arguments (like C++ reference arguments) in functions, but the intermediate form doesn't allow them. The decompiler turns reference arguments into `case-lamdba` closures over the reference arguments with one case for getting the argument value and one case for setting the argument value.

The decompilation algorithm is not the identity function when composed with compilation because of the transformations performed on cyclic closures and reference arguments. Sometimes the optimizer will remove the **`case-lambda`** arguments and turn them back into reference arguments, but this is not guaranteed. Racket has robust bytecode verification built in to the module loading system, so errors that were introduced by the decompiler were mostly caught by bytecode verification.

5.5 Limitations

Racket provides features that treat modules as first-class objects during runtime. For example, programs can load and evaluate modules at runtime through `dynamic-require`. Because the demodularizer cannot know ahead of time what modules might be loaded at runtime, it disallows programs that use `dynamic-require`. If the modules loaded through `dynamic-require` were completely separate (meaning they do not share any required modules) from the main program, it would be possible to demodularize the program, but in practice most modules required at runtime will share with the main program.

The restriction of not allowing `dynamic-require` means that programs that need to load modules on the fly, such as REPLs, sandboxed evaluation environments, and scripting

environments cannot be demodularized. This is okay because it is unclear what whole program compilation means for such programs.

Chapter 6

Evaluation

We tested our implementation of demodularization by selecting existing Racket programs and measuring their execution time before and after demodularization. We also measured the memory usage and compiled bytecode size of the programs. We ran the benchmarks on an Intel Core 2 Quad machine running Ubuntu and ran each program X times. We expect programs to perform better based on how modular the program is, which we measure by counting the number of modules in a program's require graph and how many cross module references occur in the program.

Figure XXX shows the results of running this experiment on XXX Racket programs. On one end of the spectrum, there are programs like XXX which are already basically single module programs, so demodularization does little besides rerun the optimizer on the program. Running the optimizer again may have positive or negative effects on performance, it may unroll loops and inline definitions more aggressively the second time, but some of these "optimizations" may hurt performance. On the other end of the spectrum, highly modular programs like XXX perform much better after demodularization. We expect performance to increase at a linear or even superlinear pace as modularity increases because of the extra information available to the optimizer.

This experiment uses only the existing Racket optimizations, which are intra-module optimizations. Certain optimizations that are not worthwhile to do at the intra-module level have larger payoffs when applied to whole programs. With demodularization, we anticipate that new whole-program optimizations enabled by demodularization will increase

performance even more.

Chapter 7

Related Work

Prior work on whole-program optimization has come in two flavors, depending on how much access to the source code the optimizer has. The first approach assumes full access to the source code and is based on inlining. The second approach only has access to compiled modules and is based on combining modules.

The first approach is based on selectively inlining code across module boundaries because it has full access to the source code of the program [2, 3]. Most of the focus of this approach is finding appropriate heuristics to inline certain functions without ballooning the size of the program and making sure the program still produces the same results. Resulting programs are not completely demodularized; they still have some calls to other modules. Specifically, Chambers et al. [3] show how this approach applies to object-oriented languages like C++ and Java, where they are able to exploit properties of the class systems to choose what to inline. Blume and Appel [2] showed how to deal with inlining in the presence of higher order functions, to make sure the semantics of the program didn't change due to inlining. Their approach led to performance increases of around 8%.

The second approach is taking already compiled modules, combining them into a single module, and optimizing the single module at link time [4, 5]. Most of the work done with this approach optimized at the assembly code level, but because they were able to view the whole program, the performance increases were still valuable. The link-time optimization system by Sutter et al. [4] achieves a 19% speedup on C programs. One of the reasons for starting with compiled modules is so that programs using multiple languages

can be optimized in a common language, like the work done by Debray et al. [5] to combine a program written in both Scheme and Fortran. The main problem with this approach is that the common language has less information for optimization than the source code had. These approaches are similar to demodularization, but the operate at a lower level and work on languages without phased module systems.

Chapter 8

Conclusion

Demodularization is a useful optimization for deploying modular programs. A programmer can write a modular program and get the benefits of separate compilation while devloping the program, and then get additional speedups by running the demodularizer on the completed program. Demodularization also enables new optimizations that are not feasible to implement for modular programs. Without module boundaries, inter-procedural analysis is much easier and worthwhile. Also, dead code elmination works much better because the whole program is visible, while in a modular program, only dead code that is private to the module can be eliminated.

In the future, we would like to implement an aggressive dead code elimination algorithm for Racket. We implemented a naive one that does not respect side effects, but shows the potential gains from this optimization; it is able to shrink Racket binaries down from about 2MB to about 100KB. This promising result implies that other low-hanging optimizations should be possible on demodularized programs that can increase performance.

References

[1] Separate compiliation guarantee. `http://docs.racket-lang.org/reference/eval-model.html#%28part._separate-compilation%29`, 2015. [Online; accessed 4-December-2015].

[2] Matthias Blume and Andrew 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.

[3] Craig Chambers, Jeffrey Dean, and David Grove. Whole-program optimization of object-oriented languages. Technical report, 1996.

[4] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.*, 27(5):882–945, September 2005.

[5] Saumya K. Debray, Robert Muth, and Scott A. Watterson. Link-time improvement of scheme programs. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 76–90, London, UK, UK, 1999. Springer-Verlag.

[6] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 94–104, New York, NY, USA, 1998. ACM.

[7] Matthew Flatt. Composable and compilable macros: you want it when? In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 72–83, New York, NY, USA, 2002. ACM.

[8] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[9] Jay McCarthy. Datalog: Deductive database programming. `http://docs.racket-lang.org/datalog`, 2015. [Online; accessed 4-December-2015].

[10] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19:1–301, 2009.

[11] Guy L. Steele, Jr. *Common LISP: The Language (2nd Ed.)*. Digital Press, Newton, MA, USA, 1990.

[12] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA, 2008. ACM.

[13] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.