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 programming languages provide features for creating modular programs. This may be as simple as separating code into different files or as complex as specifying separate interfaces and implementations for modules. When code is separated into modules, it is possible to compile each module separately provided enough information is known about calls to other modules. Separate compilation allows programmers to work on one part of their program without needing to recompile the whole program, but this convenience comes at a cost. The compiler does not have much information about functionality from other modules as it compiles a single module, making optimizations more difficult.

A separately compiled program is turned into an executable through linking all of the modules together, which can happen either statically or dynamically. Static linking creates a full executable program by combining all of the modules of the program before running the program. Dynamic linking creates references to other modules in the program and loads them as needed when the program runs. Linking is just one step in the process of taking a program from source code to running code, and programming languages often have more steps that happen in that transformation process. This research looks at the impact macro systems have on optimizing separately compiled programs.

Some programming languages provide macro systems, which run before compiling,

that enable programmers to add features to the programming language through manipulation of the syntax of the program. Simple macro systems allow textual replacement, like the preprocessor in C/C++, while more advanced systems provide access to the syntax as objects with rich lexical information and a full programming language to manipulate the syntax. Some of these more advanced macro systems even allow programmers to write macros in the same programming language they use to write their programs.

These advanced macro systems give programmers the ability to extend a programming language in arbitrary ways, even to the point of being able to create domain specific languages as collections of macros. When creating such programs, the details of individual macros are often hidden in modules, and, much like a separately compiled program, these modules need to be linked into an executable. Unlike a separately compiled program though, macros generate code, so they need to run before the program can be compiled.

Macros written in the same language as the program need to use the same compiler that the language uses, but the compilation needs to happen before normal program compilation. Macros can also use other macros in their implementation, which means that those other macros must be ready to run (compiled) before their use. This leads to an interleaving of compiling and running code that goes back and forth to ultimately produce the final program.

Separate compilation is more difficult in the presence of an advanced macro system. Macros are defined side-by-side with program definitions, which means that macro definitions can be spread across modules and can also use functionality provided in separate modules. 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 solution to the problems of separate compilation with advanced macros is described by Flatt and implemented for the Racket programming language [11]. He explains a method of compiling modules in phases, where each phase corresponds to which part of the program is running and which part of the program is compiling. In this case, running means executing

code, which includes executing macros, because macros are just code as well. Compiling means translating code from source code to executable code, perhaps with macros doing part of the translation. At phase 0, the main program is running and nothing is compiling. At phase 1, the main program is compiling and macros that it uses are running. At phase 2, the phase 1 macros are compiling and any macros used in the compilation of phase 1 macros are running, and so on. The compiler starts compiling at the highest phase of the program (which can be upwards of phase 70 in normal Racket programs) and compiles each phase downward until it produces phase 0 code. By separating compilation into phases, it is possible to have a both a module system and a macro system coexist.

Separate compilation makes compiler optimization more difficult and less effective. Good abstractions, provided by splitting a program into modules, 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. A single module compiled alone is difficult to optimize because the compiler has little to no information about values that come from other modules when compiling the module. Existing optimizations have even less information when modules can extend the compiler, as modules can in languages with advanced macro systems.

Language implementations have solved the problem of optimizing separately compiled programs in various ways. Some implementations of languages avoid the problems associated with separate compilation by not allowing it. The whole-program optimizing compilers Stalin [16] for Scheme and MLton [5] for ML take this approach. Other implementations do optimizations while statically linking the program. There are options in gcc [1] and clang [4] to do this type of optimization. This type of optimization is usually done on machine code, so it can be too low level to do certain optimizations. Some implementations perform cross-module inlining by selectively choosing functions to inline between modules. Inlining must be heuristic-based, and good heuristics are hard to develop. Just-In-Time (JIT) compilers attack the problem in a different way by deferring optimizations until the program is running,

where it has more information on the actual use of the program.

Our solution for optimizing modular programs in the presence of an advanced macro system, called demodularization, is to transform a modular program into a non-modular program by combining all phase 0 (runtime) code and data in the program into a single module. Conceptually, this is similar to what static linkers do in programming languages with separate compilation, but more complicated because finding all phase 0 code from the set of modules that comprise a program is not trivial. Modules can include other modules at different phase offsets, creating a directed acyclic graph of module relationships. In finding and combining all of the phase 0 code, the demodularization algorithm removes code for all other phases because they were only needed to compile the program, not run it.

After combining, the single module is then re-run through the existing optimizer, but this time the optimizer has information about the whole program. This part is similar to doing link-time optimization for programming languages with separate compilation, but is at a higher level than most link-time optimizers. Demodularization is meant to be done when producing the final production version of a program, so that during development, a programmer can still take advantage of separate compilation.

Thesis *Demodularizing programs in a language with a module system and an expressive macro system is feasible and yields more optimization opportunity for the compiler, resulting in more efficient code when compared to phased compilation and cross-module inlining.*

The approach we take to validating the thesis is to first describe the Racket module and macro systems in Chapter 2 as an example of a language with a module system and an expressive macro system that is open source. Then, in Chapter 3, we explain demodularization at a high level with a detailed example to show how the algorithm works on a Racket program. Next, we use an operational semantics model of the demodularization process on a simple language to show that demodularization preserves program behavior (Chapter 4). The operational semantics model removes the unnecessary details of the full implementa-

tion so the demodularization process is easier to understand and verify. We then describe an actual implementation for Racket (Chapter 5). The implementation presents interesting difficulties in using the Racket bytecode format and integrating with the existing compiler framework written in C. Following that are experimental results of demodularizing and optimizing Racket programs and comparing them to phased compilation and cross-module inlining (Chapter 6). The experimental results show that demodularization improves performance, especially when a program is highly modular. Finally, we discuss related work on whole program optimization (Chapter 7) and conclude with a summary and discussion of future work (Chapter 8).

Chapter 2

The Racket Module and Macro Systems

Demodularization is made to run on a program written in a language that combines macros and modules. For this research, we chose to implement demodularization for the Racket programming language because of its advanced macro and module systems. This chapter gives background information on Racket through a detailed example program that demonstrates macro and module usage.

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 [18] and Scheme [17], 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 [20] that are suitable for specific tasks.

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 [19], Object-Oriented Racket [10], Logic Programming [14], 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

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

Listing 2.1: a macro implementation of a `while` loop

things like function application syntax.

2.2 Macros

Macros are the main way programmers add new features to Racket. Macros are written alongside normal functions in a program, but they are used during the compilation of the program. Essentially, 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 function 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 2.1 shows how to write a macro that adds `while` loop to the Racket language. The macro turns a `while` loop into a recursive function and a call to that recursive function. The `#'` creates a syntax object out of the following parenthesized expression. The `test` and `body ...` forms are pattern variables that will be substituted by **`syntax-case`** when a

```
(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 2.2: use and expansion of a `while` loop

programmer uses the macro, with `test` matching one expression and `body ...` matching one or more expressions. Listing 2.2 shows a use of the while macro and what it expands into after running the macro. Whatever syntax is in the place of the pattern variables at the use of the macro will be put into the output of the macro.

Racket's macros are hygienic [13], which means that identifiers created from programs will not clash with identifiers created from macros. This means that macros are protected from the surrounding program changing their identifiers, and also that the program is protected from the macro changing its identifiers. In this example that means that the Racket program using a while loop macro will not be able to call the resulting `while-loop` function that the macro creates, and that if the program happens to have another definition named `while-loop`, that definition 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. Modules also allow control over imports and exports, using the Racket forms **`require`** and **`provide`** re-

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

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

Listing 2.3: `counter.rkt`: A simple Racket module implementing a counter

spectively. Imports refer to other modules that have definitions for functionality that the importing module needs by specifying the location of the other modules. Exports created by **`provide`** list all of the identifiers within a module that will be visible by other modules when imported. By default, everything is hidden and a programmer must specify what becomes visible.

Listings 2.3-2.5 form an example program that highlights the use of modules and macros together. The main module of the program, `while-test.rkt` (Listing 2.4), uses a while loop (provided by `while-lang.rkt` in Listing 2.5) and a counter (provided by `counter.rkt` in Listing 2.3) to print out counter values. The program also uses the `counter` module at compile-time to print out information about the `while` loop (in this case, how many expressions are in the `while` loop).

Listing 2.3 shows a simple Racket module and some of the features of the Racket module system. The `counter` module contains a variable definition and functions 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 2.4.

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 functionality from the `counter` module inside the `while` definition as seen in Listing 2.5. The definition for the `while` loop is the same as in Listing **??**, but now there is extra code that runs before the macro returns a syntax object. The extra code

```
#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 2.4: `while-test.rkt`: A Racket module that uses other modules

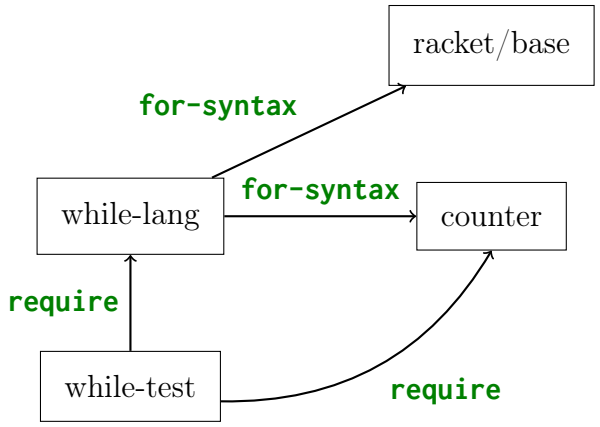

Figure 2.1: Require graph for the `while-test` program

uses the `counter` module to count how many expressions are in the body of the while loop and then prints that answer. 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.

Because the macro runs at compile-time and uses the `counter` module at compile-time, the programmer must indicate that fact to the compiler. This is done by using the **`for-syntax`** form. The macro also uses `printf`, which comes from the `racket/base` module, so that must be imported with **`for-syntax`** as well. 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.

Figure 2.1 shows the require graph of the program, which details the relationship between the different modules in the program. The require graph shows the order in which the imports are traversed when compiling and running the program. It also shows whether or not a module is required with **`for-syntax`**, which determines what phase the module is

```
#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 2.5: `while-lang.rkt`: A Racket module implementing a language with `while` loops

available at in relation to the rest of the program.

2.4 Phases

The way the Racket module system allows for reliable separate compilation 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. A module can contain both phase 0 and phase 1 code; regular function definitions are phase 0, and macro definitions are phase 1.

Higher phases occur when macros use other macros in their implementation. Another way higher phases occur is by importing modules for use inside a macro. In the example program, the `while-lang` module imports the `counter` module using **`for-syntax`**. The **`for-syntax`** form means that the module will be included for use at phase 1 relative to the

including module, so that its code is available to use inside macro definitions. There is an analogous form, `for-template`, that will include code at phase -1 relative to the including module, so that the included module's code will be available for use inside the output of a macro. By allowing references to relative phases going in both directions, a Racket program can have a complex structure of imports and exports.

It is possible for a single module to be used in multiple phases, and each phase will have a separate instantiation of the module. In the example program above, the `counter` module will be instantiated twice, once for use while macros are running at phase 1, and once for use while the main program is running at phase 0.

Even though compilation of a program can go through many phases and interleave compiling and running code many times, side effects that occur during this process are discarded. 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" [2].

2.5 Compiling and Running Programs

Compiling a Racket program interleaves traditional compiling (turning source code into bytecode) and running macros. This process can then trigger compilation at higher phases if the macros use other modules in their implementation. We will use compiling and running the example program in Listings 2.3-2.5 to illustrate the process.

Compilation starts with the main module of the program, in this case `while-test.rkt`. In terms of phases, compilation starts at phase 1. Anytime the compiler encounters a `require` form, it will switch to compiling the imported module, which would be `while-lang.rkt` in the example. Compiling `while-lang.rkt` will trigger compilation of its imports as well, but this time its imports are required with `for-syntax`. This means that compilation switches to phase 2, and compiles the imports. We can assume that `racket/base` is already compiled, so the compiler moves on to compile `counter.rkt`. Because `counter.rkt` has no imports,

it can be compiled completely into bytecode. Now, the compiler can finish the macro in `while-lang.rkt` and produce bytecode for the macro. Next, the compiler switches back to phase 1 and continues compiling `while-test.rkt`. The compiler runs into the import for `counter.rkt`, but it has already been compiled, so it moves on to compiling the body of the module. The body contains a use of the `while` macro, so it runs the code for the macro, which includes printing out the number of expressions in the `while` loop. Finally, the compiler turns the output of the macro into bytecode and finishes compilation.

To run the program, the Racket runtime loads the compiled bytecode for the `while-test` module and runs it, and when the runtime encounters imports, it runs them as well. With our example program, that means that nothing will happen on the import of `while-lang` because it just contains a macro, and on the import of `counter` the runtime will load the function definitions and create the `x` variable. This is a separate instantiation of the `counter` module than the one that was used at compile time, so the counter starts fresh from 0.

2.6 Preparing for Demodularization

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 there will be phase 0 code to run from an imported module where the import path to that module includes phase shifts. Module phases are relative to one another, so if one module imports a module at phase 1, which then imports a different module at phase -1, the code in the third module will be at phase 0 relative to the first module and must be run in the final program.

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 3.1 shows an example of using such a library. The library is implemented as a macro, shown in Listing 3.2, 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 3.1: `main.rkt` module with queue usage

implementatations depending on the length of the initial queue.

The macro `with-queue` uses `syntax-case` for its pattern-matching capabilities, and creates a syntax object that will use the long or short implementation depending on the length of `(v ...)`. The macro uses a feature from the `racket/syntax` module called `format-id` which operates like `printf` but instead of creating a string and printing it, it creates a

```
#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 3.2: `queue.rkt` module

syntax object for an identifier. So, `make-queue` is a syntax object that will refer to either `make-long-queue` or `make-short-queue`. Similarly, `enqueue` and `dequeue` will refer to syntax objects for the correct kind of queue. The final syntax object is created using `#`` which creates a syntax object but allows for escaping with `#,` to insert the actual syntax objects created above. All of this work with syntax objects makes it so that the correct queue will be used at compile-time and there will be no overhead of choosing a queue implementation at runtime, and it will all be invisible to the user of the `with-queue` macro.

Listing 3.3 and Listing 3.4 show the two queue implementations, with the `long-queue` using vectors and the `short-queue` using lists. The actual implementations are not important

for this example, so most of the code is elided. The `...` notation has meaning in Racket, so `---` is used to indicate elided code.

```
#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 3.3: `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 3.4: `short-queue.rkt` module

The require graph for this program is shown in Figure 3.1. The `main` module only includes the `queue` module, and the `queue` module includes two modules using **`for-syntax`** and two using **`require`**.

Compiling this program involves expanding the `with-queue` macro, which will turn the program into Listing 3.5.

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

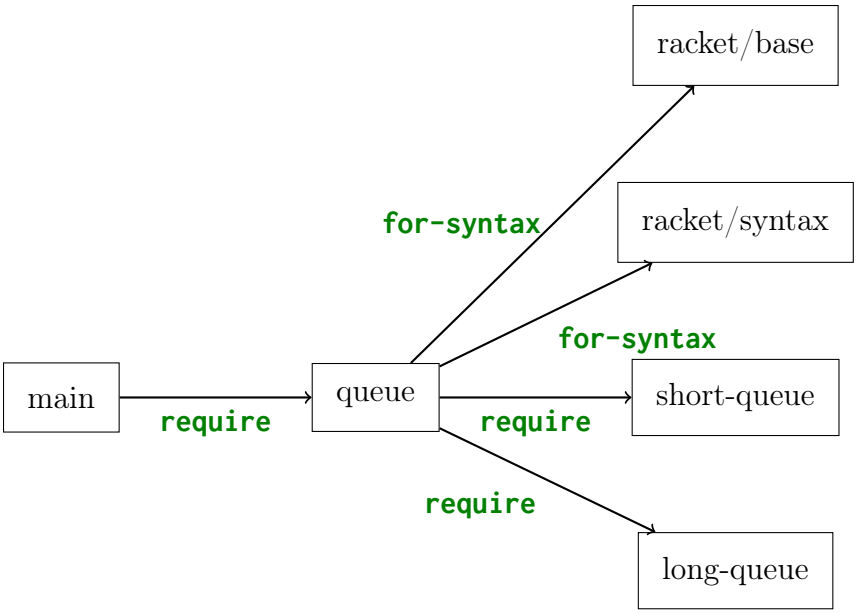

Figure 3.1: Require graph for the `main` program

```
(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 3.5: `main.rkt` module after macro expansion

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

3.2 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 a macro definition, which is phase 1 code, so it is not put into the main module. The `queue` module requires `racket/base` and `racket/syntax` using **`for-syntax`**, which means that all of their definitions are imported at phase 1 and do not need to be included in the output (it is possible that they have definitions for phase -1 which would need to be included, but in this simple example we will ignore that). Next, demodularization reaches the imports for the `short-queue` and `long-queue` modules and includes all of their definitions (which are all phase 0) in the order they appear in their original modules. The demodularization algorithm then removes all imports from the main module of the program because all code has been included directly in the main module. Listing 3.6 shows what the demodularized example program would 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.

3.3 Optimization

After a program has been demodularized, the optimizer now has all the information it needs to do optimizations on the whole program. Listing 3.7 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

```
(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 3.6: `main.rkt` module after demodularization

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
   (let ((q (--- make-vector --- 1 2 3 4 5 6 ---)))
     (--- vector-set! --- 4 ---)
     (displayln (--- vector-ref ---))
     (displayln (--- vector-ref ---)))))
```

Listing 3.7: `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. We also define a compiled version of the language that mirrors the compiled version of Racket. 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 a body of code. 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

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

that model the interaction of macros with the module system. A `quote` expression is like a 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. As an example of the compilation process, consider Listing 4.2. It shows a program in the *mod* language. Listing 4.2 shows what this program would look like after compilation.

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

The code no longer contains definitions, and all code has been sorted into the appropriate phases.

```
((module foo ((require bar @ 1)) ((define x @ 1 as (+ y 8))) ((set! x 3)))
 (module bar () ((define x @ -1 as 9) (define y @ 0 as 4)) ()))
```

Listing 4.1: Example program in the *mod* language

```
((module foo ((require bar @ 1)) ((0 ((set! (bar x) 3))) (1 ((set! (foo x) (+ (bar y) 8
 (module bar () ((-1 ((set! (bar x) 9))) (0 ((set! (bar y) 4))))))
```

Listing 4.2: Compiled version of example program

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 in which 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, σ 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.

$$
\begin{aligned}
E &::= [] \\
&\quad \mid (+\ E\ expr) \\
&\quad \mid (+\ val\ E) \\
&\quad \mid (\mathsf{set!}\ var\ E) \\
&\quad \mid (\mathsf{begin}\ E\ expr) \\
\sigma &::= ([var\ val]\ ...) \\
state &::= (program\ /\ (inst\ ...)\ /\ (inst\ ...)) \\
inst &::= (id\ phase) \\
st &::= (\sigma\ /\ state)
\end{aligned}
$$

Figure 4.3: Extensions to compiled language grammar

The reduction rules in Equations 4.1–4.7 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. Evaluation proceeds by adding required modules to the evaluation stack and executing code of modules in the order they appear on the evaluation stack when there are no more requires left.

$$
\begin{aligned}
&(\sigma\ /\ ((mod_0\ ...\ (\mathsf{module}\ id_0\ ((\mathsf{require}\ id_{new}\ @\ phase_{new})\ req\ ...)\ (code\ ...))\ mod_n\ ...)\ /\ ((id_0\ phase_0)\ inst_n\ ...)\ /\ (inst_d\ ...))) \longrightarrow \\
&(\sigma\ /\ ((mod_0\ ...\ (\mathsf{module}\ id_0\ (req\ ...)\ (code\ ...))\ mod_n\ ...)\ /\ ((id_{new}\ (\ominus\ phase_0\ \ phase_{new}))\ \ (id_0\ phase_0)\ inst_n\ ...)\ /\ (inst_d\ ...)))
\end{aligned}
\tag{4.1}
$$

The 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 by the difference between the current module's relative phase and the required phase. The $\ominus$ is not part of the language and represents the prefix version of the math function minus. 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.

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

The 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.

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

The rule in Equation 4.3 replaces an addition expression of numbers with the result of computing their sum. The $\oplus$ represents the prefix version of the math function plus.

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

The 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 `begin` with a value in the first position, the rule in Equation 4.5 removes the value and replaces the `begin` expression with the expression in the second position.

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

When there are no more expressions left in a module, the 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 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.

The combination of all of these rules forms a definition for how modular programs in the compiled *mod* language are evaluated.

4.4 Demodularization

Figures 4.4–4.7 show 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.

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

Figure 4.4: Demodularization algorithm

The 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 rules in Figure 4.5 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 first rule handles the case

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

Figure 4.5: Demodularization algorithm

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 rules in Figure 4.6 apply

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

Figure 4.6: Demodularization algorithm

when handling the working list of required modules. The first rule is similar to the first rule of Figure 4.5 because it extracts code that will be in phase 0 of the main module and inserts it into the main module. The second rule 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.

$$\begin{aligned}
&\text{demod}[\![id, (), (mod_0 \ldots\ (\text{module } id\ (req \ldots)\ (code \ldots))\ mod_n \ldots)]\!] \\
&\quad = \text{demod}[\![id, (), ((\text{module } id\ (req \ldots)\ (code \ldots))\ mod_0 \ldots mod_n \ldots)]\!] \\
&\text{demod}[\![id, (), ((\text{module } id\ ((\text{require } id_r\ @\ phase_r)\ req_m \ldots)\ (code_m \ldots))\ mod_i \ldots\ (\text{module } id_r\ (req_r \ldots)\ (code_r \ldots))\ mod_n \ldots)]\!] \\
&\quad = \text{demod}[\![id, (), ((\text{module } id\ ((\text{require } id_r\ @\ phase_r)\ req_m \ldots)\ (code_m \ldots))\ (\text{module } id_r\ (req_r \ldots)\ (code_r \ldots))\ mod_i \ldots mod_n \ldots)]\!] \\
&\text{demod}[\![id, ((id_c\ phase_c)\ (id_n\ phase_n) \ldots), ((\text{module } id\ (req_m \ldots)\ (code_m \ldots))\ mod_i \ldots\ (\text{module } id_c\ (req_c \ldots)\ (code_c \ldots))\ mod_n \ldots)]\!] \\
&\quad = \text{demod}[\![id, ((id_c\ phase_c)\ (id_n\ phase_n) \ldots), ((\text{module } id\ (req_m \ldots)\ (code_m \ldots))\ (\text{module } id_c\ (req_c \ldots)\ (code_c \ldots))\ mod_i \ldots\ mod_n \ldots)]\!] \\
&\text{demod}[\![id, ((id_c\ phase_c)\ (id_n\ phase_n) \ldots), ((\text{module } id\ (req_m \ldots)\ (code_m \ldots))\ (\text{module } id_c\ ((\text{require } id_r\ @\ phase_r)\ req_c \ldots)\ (code_c \ldots))\ mod_i \ldots\ (\text{module } id_r\ (req_r \ldots)\ (code_r \ldots))\ mod_n \ldots)]\!] \\
&\quad = \text{demod}[\![id, ((id_c\ phase_c)\ (id_n\ phase_n) \ldots), ((\text{module } id\ (req_m \ldots)\ (code_m \ldots))\ (\text{module } id_c\ ((\text{require } id_r\ @\ phase_r)\ req_c \ldots)\ (code_c \ldots))\ (\text{module } id_r\ (req_r \ldots)\ (code_r \ldots))\ mod_i \ldots\ mod_n \ldots)]\!]
\end{aligned}$$

Figure 4.7: Demodularization algorithm

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.

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 Sketch. 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 run the same module code in the same order and result in the same stores. □

The full proof of the theorem would be irrelevant to the implementation because graph traversal is not the difficult part of the demodularization algorithm, instead it is the complexities of the bytecode representation which are not modeled in the *mod* language.

Chapter 5

Implementation

The implementation of demodularization for the Racket module system integrates with the existing compilation process. The goal of the implementation was to write the actual algorithm in Racket, but to use as much of the existing infrastructure as possible. We did not want to duplicate work done by the compiler, so the demodularization algorithm operates on Racket bytecode. We also did not want to duplicate work done by the optimizer, so we pass the demodularized program through the existing optimizer. All of the pieces of the demodularization process are contained within a command line tool.

5.1 Racket Compilation Process

Figure 5.1 shows the compilation process that a Racket module undergoes and the various intermediate forms the code takes. In principle it would be possible to write the demodularization algorithm for modules at any point in this process, but we chose to write it at the bytecode level because it is a well defined file type and phases are clearly deliniated.

The compilation process for Racket is all written in C. The first step of compiling a Racket module is to expand all of the macros it uses into what is known as the Racket kernel language or fully expanded code. The next step is compilation to an Intermediate Representation (IR) made up of C data structures. The optimizer works on this IR to produce an optmized version of it. Then, the compiler finishes turning the IR into bytecode.

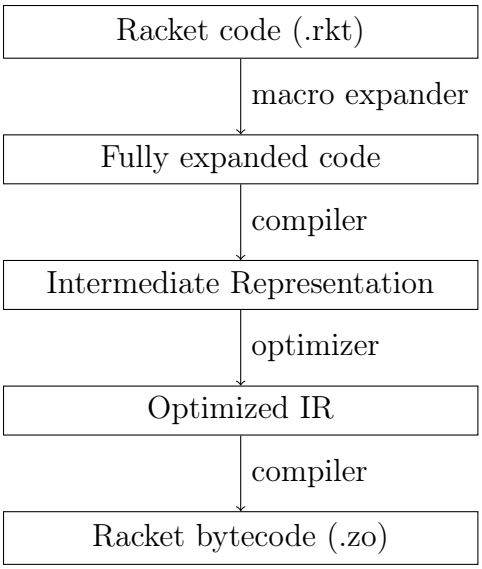

Figure 5.1: Racket compilation process

5.2 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 "The Racket virtual machine and randomized testing" [12].

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 5.1 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 5.2.

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

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

Listing 5.1: 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 5.2: Bytecode representation of program from Listing 5.1

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.

5.3 Demodularization

The implementation of demodularization for the Racket module system is written in Racket and consumes and produces Racket bytecode. Figure 5.3 shows the demodularization process and how it interacts with existing Racket components. 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

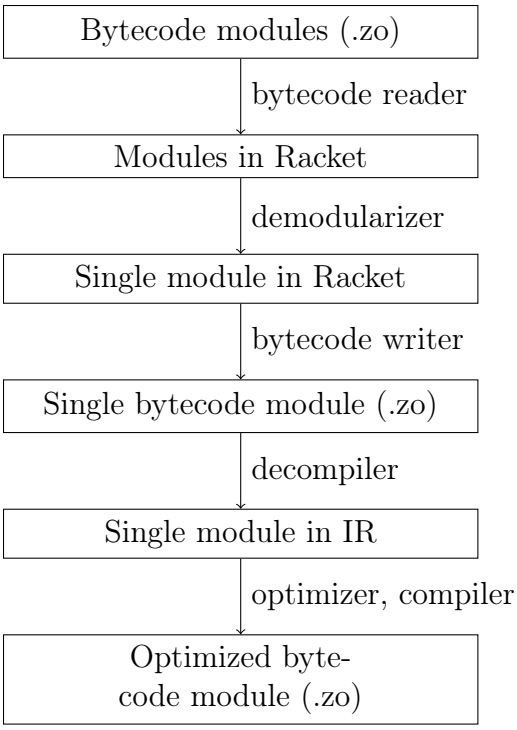

Figure 5.2: Demodularization process

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 5.2, 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.

5.4 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.

5.5 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 5.2 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.

5.6 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 such programs are fundamentally incomplete programs whose full meaning isn't known until runtime.

5.7 Usage

The implementation of demodularization 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. For example, to demodularize a program whose main module was `while-test.rkt`, a user would type the following command:

```
raco demod -O while-test.rkt
```

Chapter 6

Evaluation

For demodularization to be useful, it should improve performance of modular programs. To test whether or not this is the case, we created benchmarks to test the speed of demodularized programs versus their modular counterparts. We also compare the results with the cross-module inliner included in Racket. We show improvements in the size of programs through the use of a dead-code elimination algorithm. Finally, we discuss further improvements that demodularization can enable.

6.1 Racket Optimizer

The Racket optimizer included a cross-module inliner in version 5.2. This inliner accomplishes many of the same improvements that the demodularizer does. The inliner does have limitations on the size of functions that it will inline, so with large functions demodularization is still preferable. We compare the results with the inliner turned off, with the inliner turned on, and with demodularization. The demodularized programs are also run through the existing Racket optimizer.

6.2 Testing Setup

The tests were run on a MacBook Pro (2GHz Intel i7 processor, 8GB Memory) in OS X Yosemite, using Racket v6.2. Each test was run 5 times and an average of the results was taken.

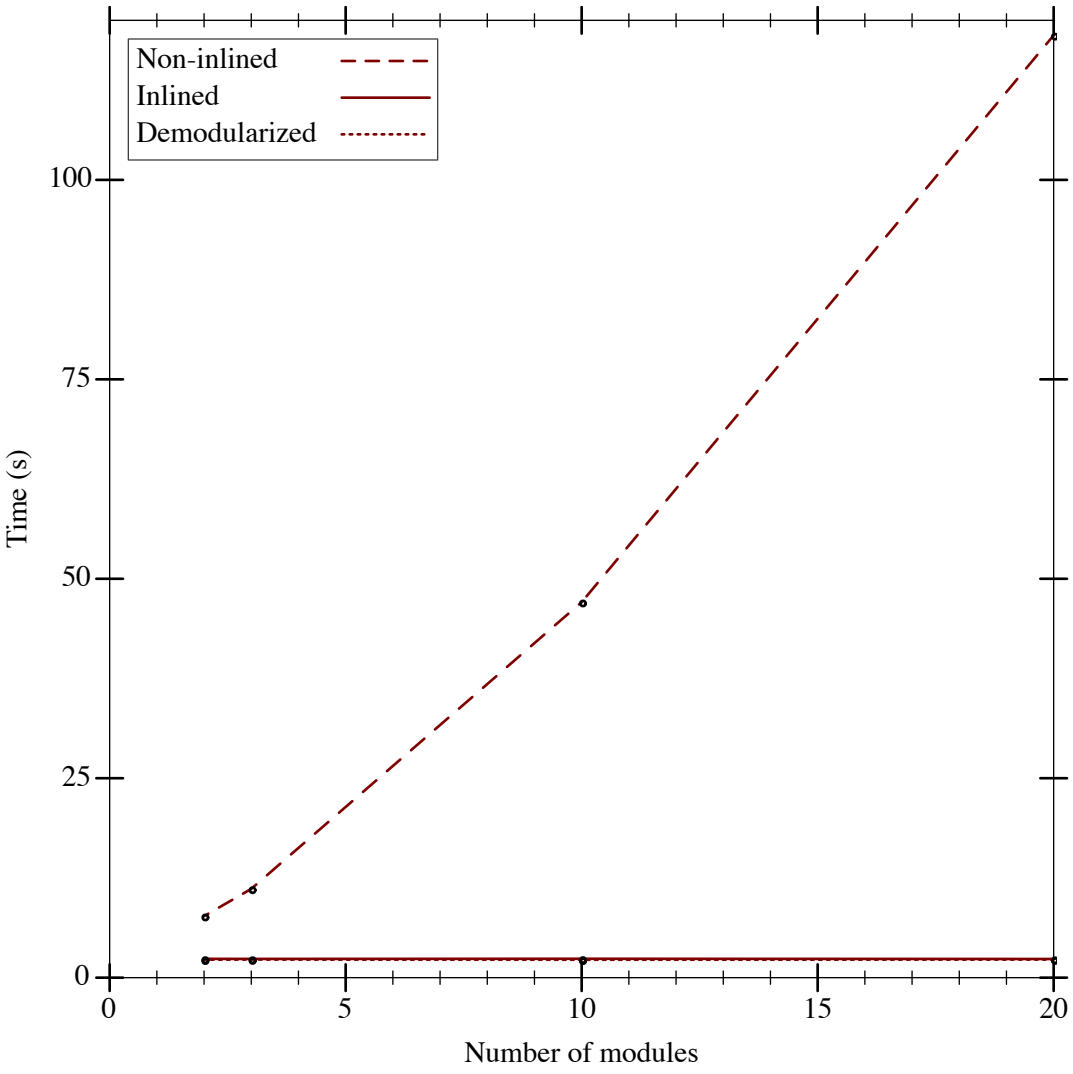

Figure 6.1: Results from micro benchmarks

6.3 Micro benchmarks

The micro benchmarks for demodularization involve a module that calls a function from another module, which calls a function from another module, and so on for the number of modules in the test. The function call happens in a loop to get times that are significant. The full test generation code can be found in Appendix A. Figure 6.1 shows the results of running these benchmarks with different numbers of modules. The X-axis represents the number of modules in a program. The Y-axis represents the time in seconds it takes for the program to run, where lower times are better. Without inlining, the micro benchmarks

increase in a linear fashion with an increase in the number of modules. With inlining, the micro benchmarks stay at a constant speed. With demodularization, the micro benchmarks also stay at a constant speed, with a slightly better speed than inlining. Both inlining and demodularization result in similar final bytecode, where the only difference is how many times the loop is unrolled. The cross-module inlining algorithm is more complicated than the demodularization algorithm, so there are cases where inlining would fail and demodularization would not. The inlining algorithm creates annotations on the bytecode of functions, which it must track and update. If a function is too large, the algorithm will not inline it (even if it is only used once). Demodularization does not have these limitations because it every function in the program is available to the optimizer as if it were a local function. Therefore, it would be possible to create a benchmark that differentiates between cross-module inlining and demodularization by creating functions that are too large to inline, but would still be optimized in the demodularized program.

6.4 Macro Benchmarks

For macro benchmarks we selected programs that would terminate deterministically and that used multiple modules. One of the programs we tested was a program that uses the Racket XML library to read large XML files. The program can be found in Appendix A. For test data, we used astronomical data from NASA [3]. The second program we tested is a benchmark for the Redex tool. Redex is a library that allows users to build and test semantic models. We also tested uses of the math and plot libraries, which are written in a modular way. Figure 6.2 shows the results of running demodularization on larger programs.

6.5 Dead Code Elimination

As an experiment, we implemented an unsound dead code elimination algorithm for demodularized programs. It identifies all uses of toplevel definitions in the body of the demodularized

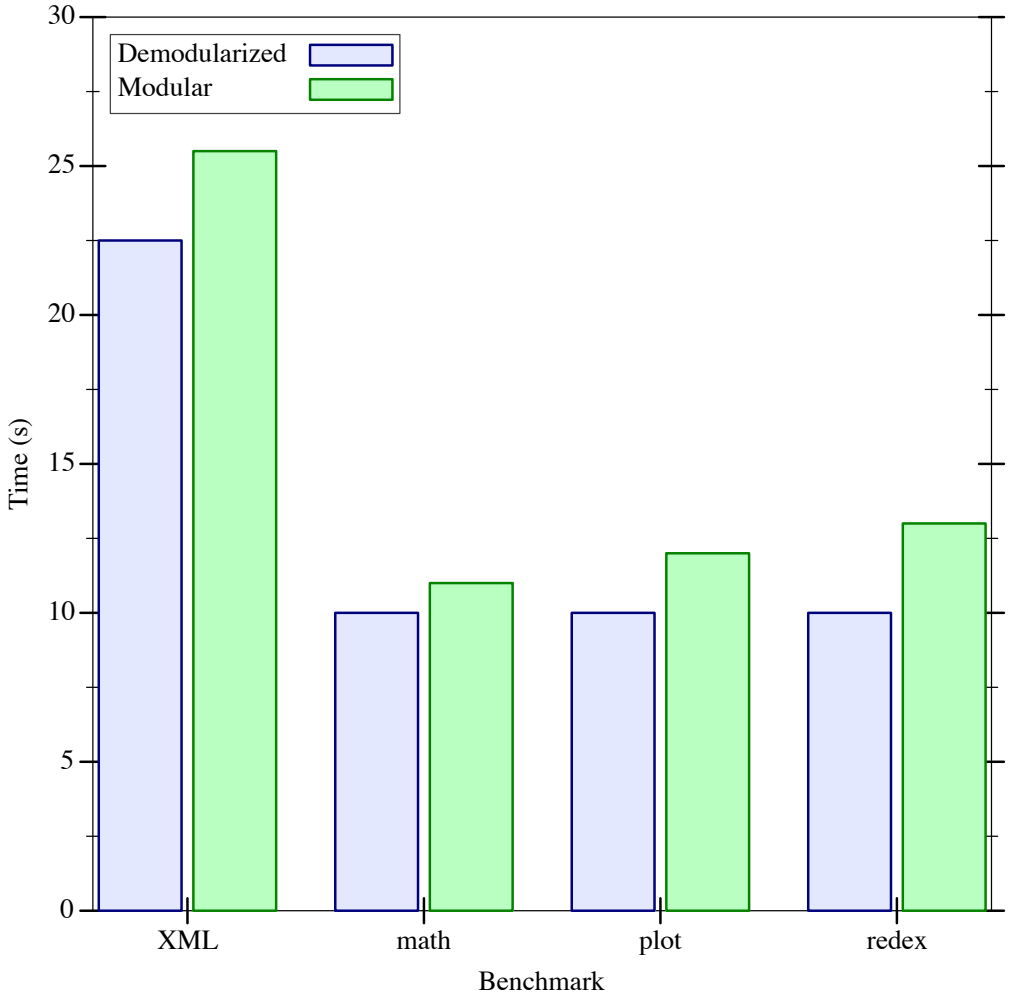

Figure 6.2: Results from macro benchmarks

program and eliminates all other toplevels. The reason it is unsound is because although some toplevels may not be referenced directly in the program, they may be needed for side-effects that they have. These side-effects may include setting up global objects or tables that will be referenced by the program, or I/O operations. In order to make the dead code elimination algorithm sound, we would need to identify primitives that can have side-effects and include any toplevels that use those primitives. This would be possible to do with some engineering effort, but it is beyond the scope of this work. The experiment gives a lower-bound on the performance of this optimization, so it is a useful result. Figure 6.3 shows how much smaller programs become after running the dead code elimination algorithm on them. When dead code is eliminated it opens up opportunities for other optimizations because the code is smaller. It opens up opportunities for inlining functions that are used a small number of times or determining if the values of arguments are the same. Also, smaller code size can be a benefit by itself in space constrained or networked systems.

6.6 Further Optimizations

Having access to the whole-program at once enables optmizations that currently are not implemented by the Racket optimizer. For example, any optimizations that rely on Control-Flow Analysis (CFA) [15] require access to the whole program. These include type test eliminations and inlining inside function definitions based on arguments. Demodularization enables these sorts of optmiziations to be performed on modular programs.

Demodularization is a simple solution to the optimization problems that arise from modular programming. Even just using existing optimizations that were not implemented with whole programs in mind, we see some benefits in performance. When new optimizations are developed that take advantage of whole programs, we should see even bigger improvements in performance.

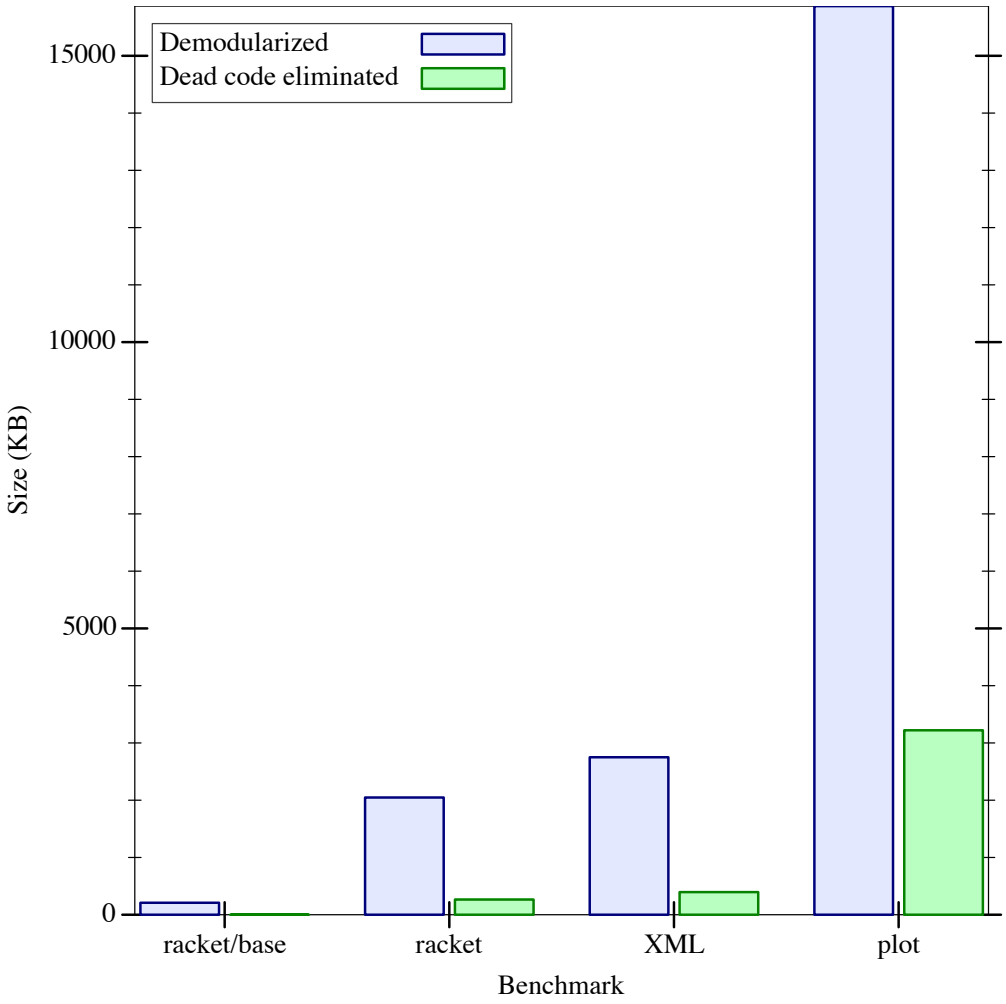

Figure 6.3: Dead code elimination results

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 [6, 7]. 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. [7] 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 [6] 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 [8, 9]. 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. [8] 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. [9] 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. We would like to see implementations of Control-Flow Analysis for Racket programs now that whole programs are accessible through demodularization.

References

[1] Link-time optimization in gcc: Requirements and high-level design. https://gcc.gnu.org/projects/lto/lto.pdf, 2005. [Online; accessed 18-January-2016].

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

[3] Xml data repository. http://www.cs.washington.edu/research/xmldatasets/www/repository.html#nasa, 2015. [Online; accessed 16-December-2015].

[4] Llvm link time optimization: Design and implementation. http://llvm.org/docs/LinkTimeOptimization.html, 2016. [Online; accessed 18-January-2016].

[5] Mlton. http://www.mlton.org, 2016. [Online; accessed 18-January-2016].

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Casey Klein, Matthew Flatt, and RobertBruce Findler. The racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, pages 1–45, 2013.

[13] 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.

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

[15] Olin Grigsby Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

[16] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical report, 1999.

[17] 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.

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

[19] 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.

[20] 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.

Appendix A

Benchmarks

A.1 Micro-benchmark generator

```
#lang racket

(define (compile-without-optimizations filename)
  (parameterize ([current-namespace (make-base-namespace)]
                 [compile-context-preservation-enabled #t])
    (with-output-to-file (format "compiled/~a" (path-add-suffix filename ".zo"))
      #:exists 'truncate
      (lambda ()
        (write (compile (read (open-input-file filename))))))))

(define (write-module n)
  (with-output-to-file (format "m~a.rkt" n)
    #:exists 'truncate
    (lambda ()
      (printf
"(module m~a '#%kernel
  (#%require \"m~a.rkt\")
  (#%provide f~a)
  (define-values (f~a) (lambda (x)
                         (f~a x))))
"
n (add1 n) n n (add1 n)))))

(define (write-first-module)
  (with-output-to-file "m0.rkt"
    #:exists 'truncate
```

```
    (lambda ()
      (printf
"(module m0 '#%kernel
  (#%require \"m1.rkt\")
  (letrec-values ([(loop)
                   (values (lambda (x)
                             (if (= x 0)
                                 'done
                                 (begin
                                   (f1 x)
                                   (loop (sub1 x))))))])
    (loop 1000000000)))
")))

(define (write-last-module n)
  (with-output-to-file (format "m~a.rkt" n)
    #:exists 'truncate
    (lambda ()
      (printf
"(module m~a '#%kernel
  (#%provide f~a)
  (define-values (f~a) (lambda (x)
                         (add1 x))))
"
n n n))))

(define num-modules
  (sub1 (string->number (vector-ref (current-command-line-arguments) 0))))
(write-first-module)
(for ([i (in-range 1 num-modules)])
  (write-module i))
(write-last-module num-modules)

(for ([i (in-range (add1 num-modules))])
  (compile-without-optimizations (format "m~a.rkt" i)))
```

A.2 XML benchmark

test-data.sh

```
if [ ! -f "nasa.xml" ];
then
  curl -O http://www.cs.washington.edu/research/xmldatasets/data/nasa/nasa.xml
fi
racket xml.rkt < nasa.xml
raco demod -O xml.rkt
racket xml_rkt_merged.zo < nasa.xml
```

xml.rkt

```
#lang racket/base
(require xml)
(time (void (read-xml)))
```