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 running the algorithm and existing optimizations on Racket programs.

Keywords: macros, Racket, modules, optimization

ACKNOWLEDGMENTS

Jay McCarthy, Matthew Flatt, Eric Mercer

Contents

List of Figures

List of Tables

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 [5] for Racket, 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.

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. Link time optimizations can be too low level to do useful optimizations. Inlining must be heuristic-based, and good heuristics are hard to develop.

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.

We provide an operational semantics for a simple language with a phased module system, and argue that the demodularization process preserves program meaning. We also provide an implementation of demodularization for the Racket programming language, and verify experimentally that programs perform better after demodularization.

We explain demodularization at a high level with a detailed example (Chapter 2). Next, we use the operational semantics model of the demodularization process to explain why demodularization is correct (Chapter 3), then describe an actual implementation for Racket (Chapter 4), followed by experimental results of demodularizing and optimizing real-world Racket programs (Chapter 5). 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 descended from LISP **??** and Scheme **??**. Therefore, Racket's syntax is based on s-expressions (although it is possible to change this). Racket provides many language features, such as closures, first-class continuations, macros, contracts, and garbage collection. The main implementation of Racket is a virtual machine with Just-In-Time compilation. The real power of Racket comes from the idea of Languages as Libraries **??**, where a programmer adds features to the language by writing Racket code that extends the compiler.

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 simplest way to write a macro is to use `define-syntax-rule`. `define-syntax-rule` has two arguments, the first is a pattern of syntax to match and the second is what that syntax should transform into. Listing 1 shows how to use one of these macros to write a

```
(define-syntax-rule (while test body ...)
  (let loop ()
    (if test
        (begin
          body ...
          (loop))
        (void))))
```

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

`while` loop. The first argument models what a while loop will look like in use, which will be a `while` identifier, followed by a `test` expression that will be tested each time through the loop, followed by zero or more expressions to run in the body of the while loop. This transforms into a **`let`** combined with an **`if`**, with whatever syntax the programmer put in for `test` and `body ...` being substituted into the output. Racket's macros are hygenic, which in this case means that even if the code passed in as the body contains a `loop` variable, it will not clash with the value from the macro.

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 2 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 2: `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 3.

```
#lang s-exp "while-lang.rkt"
(require "counter.rkt")

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

Listing 3: `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 4. The `while` macro has to change to use **syntax-case** so that it is possible to

```
#lang racket/base
(require (for-syntax racket/base
                     "counter.rkt"))
(provide (all-from-out racket/base)
         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))
       #'(let loop ()
           (if test
             (begin
               body ...
               (loop))
             (void))))]))
```

Listing 4: `while-lang.rkt`: A Racket module implementing a language with `while` loops

run arbitrary code alongside the pattern-matching functionality. 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` module for `printf`). The `while-lang` module also has an interesting **`provide`** form that exports everything from `racket/base`. This export allows programmers to use the `while-lang` module as a language for other modules, like was done in the `while-test` module.

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 **??**, Object-Oriented Racket **??**, Logic Programming **??**, 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.

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. In the program with the main module `while-test` above, 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 should not be 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 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.

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. This is a guarantee of the Racket module system, that "Any effects of the instantiation of the module's phase 1 due to compilation on the Racket runtime system are discarded," which is known as the "Separate Compilation Guarantee" **??**.

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 they were 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, we can see 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

Motivation and Intuition

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

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

Listing 5: `main.rkt` module with queue usage

```
#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 6: `queue.rkt` module

```
#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 7: `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 8: `short-queue.rkt` module

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

```
(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 10: `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 11: `main.rkt` module after optimization

Chapter 4

Implementation

The demodularization algorithm for the Racket module system operates on Racket bytecode. Racket's bytecode format is one step removed from the fully-expanded kernel language: instead of identifiers for bindings, it uses locations. For toplevel bindings, these locations point to memory allocated for each module known as the module's prefix. So, in `long-queue.rkt`, `make-long-queue` would be in prefix location 0 and `long-enqueue` would be in prefix location 1, and all the references to `make-long-queue` and `long-enqueue` are replaced with references to 0 and 1. Like in the model, the algorithm combines all phase 0 code into a single module, but since the references are locations instead of identifiers, the locations of different modules overlap. We solve this by extending the prefix of the main module to have locations for the required module's toplevel identifiers, and then adjusting the toplevel references in the required module to those new locations.

After combining all the code for a program into a single module, we want to optimize it. The existing optimizations for Racket operate on an intermediate form that is part way between fully-expanded code and bytecode. Therefore, to hook into the existing optimizations, we decompile the bytecode of the demodularized program into the intermediate form and then run it through the optimizer to produce bytecode once more.

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`. These features can work with demodularization, but the onus is on the programmer to make sure to use the features in particular ways. The main restriction is that the program cannot

share a module that is part of the demodularized program and also part of a dynamically required module. This restriction may seem easy to follow in theory, but in practice it is hard because most modules rely on built-in Racket libraries that will be in both the static and dynamic parts of the program.

Chapter 5

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 6

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 [1, 2]. 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. [2] 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 [1] 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 [3, 4]. 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. [3] 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. [4] 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 7

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] 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.

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

[3] 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.

[4] 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.

[5] 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.