A Module System for a Racket-like Language

Abhishek Mishra
University of Maryland, College Park
amishra6@terpmail.umd.edu

ABSTRACT

For any reasonably sized program, a module system is a necessity to organize code and minimize code reuse. This paper describes the implementation of a module system for a functional language syntactically and semantically analogous to Racket. The implementation utilizes syntactic desugaring to link multiple modules together into a single program which is finally compiled. The module system is straightforward and lightweight, meant to aid with future experimentation with the Racket sublanguage.

1 INTRODUCTION

Module systems are an integral part of any programming language as they allow for increased expressiveness and extensibility. Racket's module system is fairly straightforward; module declarations (not submodules) are only permitted in the top level, modules share a global namespace, and dependencies/exports are statically declared [1]. These are three qualities that should be preserved during the implementation of modules for the smaller Racket-like language used in this paper.

This language, which will be referred to as Bracket (suB-Racket), is a subset of Racket with less functionality, meant for implementing and experimenting with new language features. A well-formed program in the language appears like so:

```
#lang racket
(begin
 (define ...)
 (define ...)
 ...
 body)
```

where `body` is the expression evaluated and returned as the program's final result. With the use of modules, one could write programs that reduce code reuse and allow for separation of code based on functionality. For example:

File 1: list.rkt

```
#lang racket
(begin
 (provide fold map)
 (define (fold ...) ...)
 (define (map ...) ...)
 ...)
```

File 2: main.rkt

```
#lang racket
(begin
 (require "list.rkt")
 (define ...)
 ...
 (... (fold ... (map ...))))
```

A program should be able to `provide` the functions it wishes to make visible to other programs, and `require` any specific files it may need. When a program is `required`, it should only be able to access functions that are provided by the imported program. It's important to note that based on the definition of Bracket, a program may only produce a single result. Based on the previous example, any side effects from the body expression of `list.rkt` would be ignored during the evaluation of `main.rkt`. This is differs from Racket's own module system which runs the body of a module when it is imported [1].

2 IMPLEMENTATION

The final goal of the implementation is to provide a single executable that has gathered all the required dependencies, and can run the program without any issues. To accomplish this, the compiler must traverse the program's dependencies, identify all the provided functions, and make them accessible from the program. This means any program's imports/exports should be easily available to read, and the compiler should be able to link the main program to the provided definitions.

2.1 Programs to Modules

The first goal is to make imports/exports accessible to the compiler when it is loading dependencies and looking for definitions. A program in Bracket is syntactically formed as `` `(begin ,ds ... ,e) `` where `ds` is a list of definitions preceding the main body. Also, `provide` and `require` definitions can appear within `ds` in any order, which must be accounted for. For example, the following is valid:

```
#lang racket
(begin
 (define (f x) (g x))
 (provide f)
 (require "g.rkt")
 ...)
```

This is handled in the implementation by transforming the program into a module in a single pass before any dependency resolution is done. A module's syntactic form is `` `((,imp . ,exp) . ,p) `` where `imp` is the list of filenames or modules this module requires, `exp` is the list of provided definitions in this module, and `p` is a program in the form `` `(begin ,ds ... ,e) `` where `ds` has no `provide` or `require` blocks. This transformation, `prog->mod`, is applied to the program being compiled, as well as any dependencies that are loaded.

2.2 Linking

Once the file being compiled has been transformed via `prog->mod`, its list of dependencies are readily available. Each file in the list has its raw text loaded, parsed, and checked for syntax. Afterwards, it is transformed via `prog->mod` as well, but dependencies go through a final transformation: all definitions that are not in a `provide` block are removed from the imported module's main body. This is to ensure any functions or values intended to be private are not incorrectly passed into the main program.

Once the dependencies are loaded and transformed, the linking process begins. Linking is quite straightforward and can be demonstrated via the following simple example with two programs:

File 1: p1.rkt

```
#lang racket
(begin
 (provide f)
 (define (f x) ...)
 (define ...)
 ...
 b1)
```

File 2: p2.rkt

```
#lang racket
(begin
 (require "p1.rkt")
 (define (g y) ...)
 b2)
```

If compiling `p2.rkt`, the side effects of expression `b1` in `p1.rkt` would be ignored due to the semantics of Bracket. The programs are combined using nested `(begin ...)` expressions, which is semantically identical to creating a flattened list of `(define ...)` statements. However, the resulting code with nested `begin` blocks would allow the programmer to easily identify where modules are separated from each other if needed. The program that results from the linking process would look like the following:

```
#lang racket
(begin
 (define (f x) ...)
 (begin
  (define (g y) ...)
  b2))
```

The result would then be compiled like any other Bracket program, and it would run and give the desired output. Both `f` and `g` are accessible from the scope of `b2`, the resulting code is easily digestible, and the Bracket compiler requires no serious modifications to handle this module implementation since changes are primarily syntax-oriented.

3 DISCUSSION & RELATED WORK

This module system provides the core features that are present in Racket's module system: modules share a global namespace and the imports/exports are statically declared [1]. Additionally, since Bracket modules also function as standalone programs, they can be separately compiled and manually linked to other compiled programs. However, this implementation doesn't provide some of the advanced features present in other module systems, or even Racket's own unit system. For example, recursive modules where two modules rely on each other's imports/exports would not be supported in this implementation of modules, since it would lead to an infinite nesting of `(begin ...)` statements.

Another core feature missing from this implementation is the ability to add nested submodules. Submodules would allow for even more extensible programs, allowing users to introduce their own "phases" that complement run-time and compile-time. For example, using Racket's submodule system, one could define a `test` submodule that would only run when the main program is run; the submodule could be entirely ignored when the outer module is imported [1].

Ths implementation of module systems is somewhat aligned with those commonly used in some dynamic programming languages like Ruby and Python. These provide limited to no encapsulation since their implementations are relatively straightforward syntactic sugar. However, these languages still succeed because their use case does not typically warrant advanced modular systems. Meanwhile, JavaScript is used for dynamic code execution across various machines with different versions of the language, resulting in a more complex module system with features such as dynamic loading and evaluation of modules based on URLs or constantly changing code [2]. JavaScript's module loaders are inspired from Java's class loaders, Java objects which dynamically load code based on class names provided by the programmer [3].

Module systems from ML were also explored prior to developing this implementation because of the prevalence of ML in programming language literature [4][5]. Although the module systems were robust and feature-rich, their applications did not apply to this implementation. For example, the module systems discussed heavily relied on ML's notion of signatures and structures, which are paradigms not utilized often in Racket and not even implemented in Bracket. However, the work is worth revisiting if a module system is being devised for a typed variant of Racket, since the proposed solutions for common static type-checking problems are addressed in these papers.

4 CONCLUSION

Overall, the Bracket module system provides a straightforward, lightweight solution to fit the use case of the language. Primarily used for experimenting with language design, it provides the barebone essentials for developers to produce relatively small modules that can be reused as needed. There are plenty of features to be added as Bracket grows, such as submodules and units, but the current implementation provides a means for more advanced experimentation with the language and future development of a more intricate module system.

REFERENCES

[1] Matthew Flatt. 2014. You Want it When, Again? - School of Computing. https://dl.acm.org/doi/pdf/10.1145/2637365.2517211

[2] David Herman and Sam Tobin-Hochstadt. 2011. Modules for JavaScript. (May 2011).

[3] Sheng Liang and Gilad Bracha. 1998. Dynamic class loading in the Java virtual machine. *ACM SIGPLAN Notices* 33, 10 (1998), 36–44. https://doi.org/10.1145/286942.286945

[4] Andreas Rossberg. 2015. 1ML – core and modules united (F-ing first-class modules). *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015* (Aug 2015), 35–47. https://doi.org/10.1145/2784731.2784738

[5] Andreas Rossberg and Derek Dreyer. 2013. Mixin' Up the ML Module System. *ACM Transactions on Programming Languages and Systems* 35, 1 (2013), 1–84. https://doi.org/10.1145/2450136.2450137