Racket: Modules, Contracts, Languages

Advanced Functional Programming

Kostis Sagonas

November 2018

(original set of slides by Jean-Noël Monette)

Content

Modules are the way to structure larger programs in smaller pieces.

Modules can import and export bindings.

Contracts define conditions on the exported bindings.

Modules can be used to define new languages.

Modules

Each file defines a separate module.

A module is usually defined in its own file.

In that case, the name of the module is the name of the file.

Modules

Each file defines a separate module.

A module is usually defined in its own file.

In that case, the name of the module is the name of the file.

Bindings are exported with `(provide funcname)`.

Bindings are imported with `(require "filename")`.

Example

```
#lang racket ; In file fact.rkt
(provide fact)
(define (fact x) (fact-help x 1))
(define (fact-help x acc)
  (case x
    [(0) acc]
    [else (fact-help (- x 1) (* x acc))]))
```

Non-exported functions are private.

```
#lang racket ; In file use-fact.rkt
(require "fact.rkt")
(fact 10)
(fact-help 10 1) ; Error
```

Provide

`provide` takes a list of elements to provide. Those elements can take various forms.

- `all-defined-out` exports all bindings defined in the module.
- `(rename-out [orig-name export-name] ...)` changes the name of some exported elements.
- `(prefix-out prefix spec)` adds a prefix to all bindings exported by `spec`.
- `(except-out spec id ...)` exports all bindings from `spec` except `id`.
- `(all-from-out module-path)` re-exports all the bindings from another module.
- ...

Provide: Example

```
(provide (except-out (all-from-out racket) lambda)
         (rename-out [lambda function]))
```

This exports all bindings provided by racket but **`lambda`**. **`lambda`** is replaced by **`function`**.

Require

`require` takes a list of elements to import. Again, they can be of various forms.

- A module name will simply import all bindings from the module.
- `(except-in spec id ...)` imports all bindings from `spec` except `id`.
- `(rename-in spec [orig-id new-id] ...)` renames some of the bindings.
- `(only-in spec id [o-id n-id] ...)` imports only some bindings, optionally renaming them.
- `(prefix-in prefix spec)` adds a prefix to all bindings.
- `(for-syntax spec)` imports at the syntax level (see macros).
- ...

Require: Example

```
(require (prefix-in racket: (except-in racket lambda))
         (only-in racket [lambda function]))
```

This imports all bindings from racket by prefixing them with "racket:", except `lambda` that is renamed to `function`.

Submodules

Submodules can be defined using `(module mname language decls ...)`, where

- `mname` is the name of the module,
- `language` defines the initially available bindings (it is usually `racket`),
- `decls ...` is the body of the module.

Submodules are not automatically evaluated along their parent. They need to be imported as any module.

Such a submodule can be imported by `(require 'mname)` (in the parent module) or by `(require (submod "fname" mname))` (in another module).

Submodules can be nested arbitrarily.

Submodule: Example

```
(module my-sum racket
  (provide plus)
  (define (plus x y) (+ x y)))

(require 'my-sum)
(plus 3 4)

(module test-sum racket
  (require (submod ".." my-sum))
  (equal? (plus 3 4) (plus 4 3)))

(require 'test-sum)
```

Submodule: Example

```racket
(module my-sum racket
  (provide plus)
  (define (plus x y) (+ x y)))

(require 'my-sum)
(plus 3 4)

(module test-sum racket
  (require (submod ".." my-sum))
  (equal? (plus 3 4) (plus 4 3)))

(require 'test-sum)
```

Note the order of evaluation: `require`'s, then `define`'s, then the rest.

Submodules (2)

Submodules cannot access the bindings of their parent module.

It is possible to define "inverted" submodules that can access the parent's bindings, but cannot be required inside it.

`(module* mname language-or-#f decls)` declares such a submodule.

The language can also be `#f`, in which case the submodule accesses all of the parent module bindings.

`(module+ mname decls)` is a shortcut when the language is `#f`. Using this form, the same submodule can also be declared in several parts.

Submodules: Example

```
#lang racket ; In file fact.rkt
(provide fact)
; ... Code of fact-help as in slide 5
(module* extra #f
  (provide fact fact-help))
```

Importing the module **extra** allows to have access to **fact-help** as well.

```
#lang racket ; In file use-fact.rkt
(require (submod "fact.rkt" extra))
(fact-help 10 1) ; Now it works
```

Special Submodules

```
(module+ main
  (display "In the main"))

(module+ test
  (when (not (= (fact 10) (fact2 10))) (raise "Problem")))
```

Those module names have a special meaning.

- `main` modules are run when the module is directly executed (not required from another module).
- `test` modules are run by the `raco` executable as `raco test`.

Modules

Modules are useful to structure larger programs.

We will see that they provide the basis for two interesting concepts:

- Contracts define the boundary between modules.
- New languages can be created transparently.

Side Note: Include

`(include "filename")` *inlines* the content of `"filename"` in the current file.

This is not to be confused with `require`!

Contracts

Contracts are used to define pre- and post-conditions on provided procedures.

They are defined as part of the `provide` instruction.

The contract system ensures that contracts are always respected.

A first contract

```
(provide (contract-out [fact (-> natural-number/c
                                 natural-number/c)]))
```

The contract stipulates that `fact` is a function taking a natural and returning a natural.

A first contract

```
(provide (contract-out [fact (-> natural-number/c
                                  natural-number/c)]))
```

The contract stipulates that `fact` is a function taking a natural and returning a natural.

`natural-number/c` is provided by the system but we can write our own test:

```
(define (nat? x)
  (and (number? x) (integer? x) (exact? x) (>= x 0)))
(provide (contract-out [fact (-> nat? nat?)]))
```

A basic contract just needs to be a procedure taking one argument. Its return value is interpreted as a "boolean".

Contract language (examples)

`(and/c number? integer?)` ensures the two conditions.

`(or/c number? string?)` ensures one of the conditions.

`any` accepts any argument(s).

`any/c` accepts one argument of any type.

`(listof string?)` accepts a list of strings.

Literals (e.g., symbols, numbers) are evaluated as a contract accepting only themselves. Example: `(or/c 'bold 'italic #f)`

Contract language (examples)

`(and/c number? integer?)` ensures the two conditions.

`(or/c number? string?)` ensures one of the conditions.

`any` accepts any argument(s).

`any/c` accepts one argument of any type.

`(listof string?)` accepts a list of strings.

Literals (e.g., symbols, numbers) are evaluated as a contract accepting only themselves. Example: `(or/c 'bold 'italic #f)`

Side Question: how would you implement e.g. `or/c`?

Contract language (examples)

`(and/c number? integer?)` ensures the two conditions.

`(or/c number? string?)` ensures one of the conditions.

`any` accepts any argument(s).

`any/c` accepts one argument of any type.

`(listof string?)` accepts a list of strings.

Literals (e.g., symbols, numbers) are evaluated as a contract accepting only themselves. Example: `(or/c 'bold 'italic #f)`

Side Question: how would you implement e.g. `or/c`?

```
(define ((or/c . tests) x)
  (ormap (lambda (test) (cond [(procedure? test) (test x)]
                              [else (equal? test x)]))
         tests))
```

Functions with optional and rest arguments

To declare a contract for a function with a variable number of arguments, use the `->*` combinator that takes three arguments (mandatory, optional, return).

```
(define (silly x y [z 1]) (* (+ x y) z))

(provide (contract-out [silly (->* (integer? integer?)
                                   (integer?)
                                   integer?)]))
```

Functions with optional and rest arguments

To declare a contract for a function with a variable number of arguments, use the **->*** combinator that takes three arguments (mandatory, optional, return).

```
(define (silly x y [z 1]) (* (+ x y) z))

(provide (contract-out [silly (->* (integer? integer?)
                                   (integer?)
                                   integer?)]))
```

One can also put a contract on the rest.

```
(provide (contract-out [max (->* ()
                                 ()
                                 #:rest (listof real?)
                                 real?)]))
```

Dependencies

```
(struct counter (cnt))
(define (count) (counter 0))
(define (inc cnt) (counter (add1 (counter-cnt cnt))))
(define (val cnt) (counter-cnt cnt))
```

How to express that the value stored in the result of `inc` is the value of the argument plus 1?

Dependencies

```
(struct counter (cnt))
(define (count) (counter 0))
(define (inc cnt) (counter (add1 (counter-cnt cnt))))
(define (val cnt) (counter-cnt cnt))
```

How to express that the value stored in the result of `inc` is the value of the argument plus 1?

With what we have seen so far, it is not possible.

We need to use yet another combinator: `->i`

Specifying dependencies

`->i` looks like `->*` but all arguments are given names so that they can be reused.

```
(define (minus a b) (- a b))
(provide
 (contract-out
  [minus (->i ([a natural-number/c]
               [b (a) (and/c natural-number/c (<=/c a))])
              [result (a) (and/c natural-number/c
                                 (<=/c a))])]))
```

Example: counter

```racket
(module counter racket
  (struct counter (cnt))
  (define (count) (counter 0))
  (define (inc cnt) (counter (add1 (counter-cnt cnt))))
  (define (val cnt) (counter-cnt cnt))

  (provide
   (contract-out
    [count (-> (and/c counter? (lambda (res) (= 0 (val res)))))]
    [val (-> counter? natural-number/c)]
    [inc (->i ([cnt counter?])
              [res (cnt)
                   (and/c counter?
                          (lambda (res)
                            (= (val res) (add1 (val cnt)))))])])))

(require 'counter)
(define x (count))
(val x)
(define y (inc (inc x)))
(val y)
```

Example: stack

```
(provide make-stack ; empty stack
         list->stack ; already filled stack
         stack->list ; The content of the stack in a list
         empty-stack? ; Is the stack empty?
         size ; The number of elements in the stack
         top ; The first element of the stack
         pop ; The stack without its top element
         push ; The stack with a new element on top
         stack?) ; Is this object a stack?
```

Stack: implementation

```
(struct stack (list))

(define (make-stack) (stack '()))
(define list->stack stack)
(define stack->list stack-list)
(define empty-stack? (compose null? stack-list))
(define size (compose length stack-list))
(define top  (compose car stack-list))
(define pop (compose stack cdr stack-list))
(define (push s el) (stack (cons el (stack-list s))))
; stack? is provided by the struct.
```

top and **pop** need a non-empty stack. This is not handled here but in the contracts.

Stack: simple contracts

First, we put contracts on types and pre-conditions.

Stack: simple contracts

First, we put contracts on types and pre-conditions.

```
(provide
 (contract-out
  [make-stack (-> stack?)]
  [list->stack (-> (listof any/c) stack?)]
  [stack->list (-> stack? (listof any/c))]
  [empty-stack? (-> stack? boolean?)]
  [size (-> stack? natural-number/c)]
  [top (-> (and/c stack? (not/c empty-stack?)) any/c)]
  [pop (-> (and/c stack? (not/c empty-stack?)) stack?)]
  [push (-> stack? any/c stack?)]
  [stack? (-> any/c boolean?)]))
```

Stack: what is a stack?

The previous contracts would still be valid if our implementation was actually a queue.

What is the main characteristic of a stack?

Stack: what is a stack?

The previous contracts would still be valid if our implementation was actually a queue.

What is the main characteristic of a stack?

LIFO: Last-In-First-Out

The elements are popped in the opposite order than they were pushed.

Condition on `push`: the pushed element is now on `top`.

Contract on push

```
(...
 [push (->i ([st stack?]
             [elem any/c])
            [res (elem) (and/c stack?
                               (lambda (res)
                                 (equal? (top res) elem)))])]
 ...)
```

Contract on push

```
(...
 [push (->i ([st stack?]
             [elem any/c])
            [res (elem) (and/c stack?
                              (lambda (res)
                                (equal? (top res) elem)))])]
 ...)
```

Exercise: What about the property that the rest of the stack is preserved?

Contracts for more complex cases

Many more complex contracts are available.

See the guide and reference.

Contracts for more complex cases

Many more complex contracts are available.

See the guide and reference.

Contracts are checked at runtime. Hence they incur an overhead.

Module boundary is a good place to check contracts.

One should avoid writing contracts that are too costly to check.

New languages

Macros can only extend the language, and do it inside the syntactic convention of the language.

If one wants to create a new language, it may be necessary to restrict or alter the language, or to change the syntax.

We will see how to do that, but not the syntax part.

New languages (2)

The "language" of a module can be defined arbitrarily.

In `(module name language body ...)`, the argument `language` can be any module.

The bindings provided by the `language` module define what is available to the new module.

One can define what is available in a language in the `provide` instruction.

A first language

```
(module racketf racket ; Like racket but "lambda" is "function"
(provide (except-out (all-from-out racket) lambda)
         (rename-out [lambda function])))

(module mymodule (submod ".." racketf) ; Use the new language
  (define sqr (function (x) (* x x)))
  ; (define sqr (lambda (x) (* x x))) ; would be an error
  (sqr 5))

(require 'mymodule)
```

Minimal export

```
(module minimal racket
  (provide #%app ; Implicit form for procedure application
           #%module-begin ; Implicit for module declaration
           #%datum ; Implicit for literals and data
           #%top ; Implicit for unbound identifiers
           lambda)) ; Just because we want to do something

(module test  (submod ".." minimal)
  ; ok
  ((lambda (x) x) 10)
  ; not ok
  ((lambda (x) (+ x 1)) 10))
```

Redefining implicit forms

```
#lang racket
(module verbose racket
  (provide (except-out (all-from-out racket)
                       #%module-begin
                       #%app
                       #%top
                       #%datum)
           (rename-out [module-begin #%module-begin]
                       [app #%app]
                       [top #%top]
                       [datum #%datum]))

  (define-syntax-rule (module-begin expr ...)
    (#%module-begin
     (displayln "Entering Module Verbose")
     expr ...
     (displayln "Leaving Module Verbose")))

  [...])
```

Redefining implicit forms

```
(define-syntax-rule (app f arg ...)
  (begin (display "Applying: ")
         (displayln '(f arg ...))
         (let ([res (#%app f arg ...)])
           (display "  res: ")
           (displayln res)
           res)))

(define-syntax-rule (top . arg)
  (begin (display "Not found ")
         (displayln 'arg)
         'arg))

(define-syntax-rule (datum . arg)
  (begin (display "Value: ")
         (displayln 'arg)
         (#%datum . arg)))
```

Redefining implicit forms

```
(module client (submod ".." verbose)
  (define x 5)
  (+ x 10 x)
  (display y))

(require 'client)
```

Result:

```
Entering Module Verbose
Value: 5
Applying: (+ x 10 x)
Value: 10
res: 20
20
Applying: (display y)
Not found y
y res: #<void>
Leaving Module Verbose
```

A new #lang

```
#lang s-exp "fname.rkt"
...
```

is equivalent to

```
(module name "fname.rkt"
  ...)
```

A new #lang

```
#lang s-exp "fname.rkt"
...
```

is equivalent to

```
(module name "fname.rkt"
  ...)
```

To change the syntax, or to get rid of the "s-exp", see the guide.

Example: Half-Life

Everytime an identifier is used, its value is divided by two.

```
#lang s-exp "Language-half-life.rkt"

(define x 10)
(define y 10)
(+ (* x y) (* x y))
(define z (* 50 y (- x x)))
(+ (if (= z 100) z 100) y)
z
```

Example: Half-Life

Everytime an identifier is used, its value is divided by two.

```
#lang s-exp "Language-half-life.rkt"

(define x 10)
(define y 10)
(+ (* x y) (* x y))
(define z (* 50 y (- x x)))
(+ (if (= z 100) z 100) y)
z
```

Produces...

```
125
51
25
```

Half-Life: Implementation

```
#lang racket

(provide + - / * = > < >= <= if
         #%app #%datum
         #%module-begin
         #%top-interaction
         (rename-out [my-define define]
                     [lookup #%top]))

(define env (make-hash))

(define-syntax-rule (my-define id expr)
  (let ([val expr])
    (hash-set! env 'id val)
    (void)))
```

Half-Life: Implementation

```
(define-syntax-rule (lookup . id)
  (if (hash-has-key? env 'id)
      (let* ([val (hash-ref env 'id)]
             [new-val (floor (/ val 2))])
        (if (not (= new-val 0))
            (hash-set! env 'id new-val)
            (hash-remove! env 'id))
        val)
      (error "This id does not exist (anymore)")))
```

Summary

- Modules are used to structure large programs in smaller pieces.
- It is possible to define contracts on the provided procedures.
- Creating a new language amounts to using macros and the module system.

General summary

Racket is a modern functional programming language.

`{if {you get sick of '()} {use {'[] or '{}}}}` :-)

It is more than one language:

- several existing languages
- the infrastructure to create your own.