

Option Contracts

Version 6.10.1

September 12, 2017

```
(require racket/contract/option)
package: option-contract-lib
```

This module introduces *option contracts*, a flavor of behavioral software contracts. With option contracts developers control in a programmatic manner whether, when, and how often contracts are checked. Using this flavor of contracts, Racketeers can mimic any compiler flag system but also create run-time informed checking systems.

```
(option/c c
  [#:with-contract with
   #:tester tester
   #:invariant invariant
   #:immutable immutable
   #:flat? flat?
   #:struct struct-id]) → contract?
c : contract?
with : boolean? = #f
tester : (or/c (-> any boolean?) 'dont-care) = 'dont-care
invariant : (or/c (-> any boolean?) 'dont-care) = 'dont-care
immutable : (or/c #t #f 'dont-care) = 'dont-care
flat? : boolean? = #f
struct-id : (or/c identifier? 'none) = 'none
```

Returns a contract that recognizes vectors or hashes or instances of struct *struct-id*. The data structure must match *c* and pass the *tester*.

When an *option/c* contract is attached to a value, the value is checked against the *tester*, if *tester* is a predicate. After that, contract checking is disabled for the value, if *with* is *#f*. If *with* is *#t* contract checking for the value remains enabled for *c*.

If *waive-option* is applied to a value guarded by an *option/c* contract, then *waive-option* returns the value after removing the *option/c* guard. If *exercise-option* is

applied to a value guarded by an `option/c` contract, then `exercise-option` returns the value with contract checking enabled for `c`. If the `invariant` argument is a predicate, then `exercise-option` returns the value with contract checking enabled for `(invariant/c c invariant #:immutable immutable #:flat? flat? #:struct struct-id)`.

The arguments `flat?` and `immutable` should be provided only if `invariant` is a predicate. In any other case, the result is a contract error.

Examples:

```
> (module server0 racket
  (require racket/contract/option)
  (provide
    (contract-out
      [vec (option/c (vectorof number?))]))
  (define vec (vector 1 2 3 4)))
> (require 'server0)
> (vector-set! vec 1 'foo)
> (vector-ref vec 1)
'foo
> (module server1 racket
  (require racket/contract/option)
  (provide
    (contract-out
      [vec (option/c (vectorof number?) #:with-contract #t)]))
  (define vec (vector 1 2 3 4)))
> (require 'server1)
> (vector-set! vec 1 'foo)
vec: contract violation
  expected: number?
  given: 'foo
  in: an element of
      the option of
      (option/c
        (vectorof number?)
        #:with-contract
        #t)
  contract from: server1
  blaming: top-level
    (assuming the contract is correct)
  at: eval:6.0
> (module server2 racket
  (require racket/contract/option)
  (provide
    (contract-out
      [vec (option/c (vectorof number?) #:tester sorted?)]))
```

```

(define vec (vector 1 42 3 4))
(define (sorted? vec)
  (for/and ([el vec]
            [cel (vector-drop vec 1)])
    (<= el cel)))
> (require 'server2)
vec: broke its own contract;

```

```

in: option contract tester #<procedure:sorted?> of
  (option/c
   (vectorof number?)
   #:tester
   #<procedure:sorted?>)
contract from: server2
blaming: server2
  (assuming the contract is correct)
at: eval:9.0

```

```

(exercise-option x) → any/c
x : any/c

```

Returns `x` with contract checking enabled if an `option/c` guards `x`. In any other case it returns `x`. The result of `exercise-option` loses the guard related to `option/c`, if it has one to begin with, and thus its contract checking status cannot change further.

Examples:

```

> (module server3 racket
  (require racket/contract/option)
  (provide (contract-out [foo (option/c (-> number? symbol?))])))
  (define foo (λ (x) x))
> (require 'server3 racket/contract/option)
(define e-foo (exercise-option foo))
> (foo 42)
42
> (e-foo 'wrong)
foo: contract violation
  expected: number?
  given: 'wrong
in: the 1st argument of
  the option of
  (option/c (-> number? symbol?))
contract from: server3
blaming: top-level
  (assuming the contract is correct)
at: eval:11.0

```

```

> ((exercise-option e-foo) 'wrong)
foo: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
      the option of
        (option/c (-> number? symbol?))
  contract from: server3
  blaming: top-level
    (assuming the contract is correct)
  at: eval:11.0

```

transfer/c : contract?

A contract that accepts any value. If the value is guarded with an `option/c` contract, `transfer/c` modifies the blame information for the `option/c` contract by adding the providing module and its client to the positive and negative blame parties respectively. If the value is not a value guarded with an `option/c` contract, then `transfer/c` is equivalent to `any/c`.

Examples:

```

> (module server4 racket
  (require racket/contract/option)
  (provide (contract-out [foo (option/c (-> number? symbol?))])))
  (define foo (λ (x) x))
> (module middleman racket
  (require racket/contract/option 'server4)
  (provide (contract-out [foo transfer/c])))
> (require 'middleman racket/contract/option)
(define e-foo (exercise-option foo))
> (e-foo 1)
foo: broke its own contract
  promised: symbol?
  produced: 1
  in: the range of
      the option of
        (option/c (-> number? symbol?))
  contract from: server4
  blaming multiple parties:
    middleman
    server4
    (assuming the contract is correct)
  at: eval:17.0
> (module server5 racket
  (require racket/contract/option)
  (provide (contract-out [boo transfer/c])))

```

```

      (define (boo x) x))
> (require 'server5)
> (boo 42)
42

```

```

| (waive-option x) → any/c
  x : any/c

```

If an `option/c` guards `x`, then `waive-option` returns `x` without the `option/c` guard. In any other case it returns `x`. The result of `waive-option` loses the guard related to `option/c`, if it had one to begin with, and thus its contract checking status cannot change further.

Examples:

```

> (module server6 racket
  (require racket/contract/option)
  (provide (contract-out [bar (option/c (-> number? symbol?))]))
  (define bar (λ (x) x)))
> (require 'server6 racket/contract/option)
(define e-bar (waive-option bar))
> (e-bar 'wrong)
'wrong
> ((waive-option e-bar) 'wrong)
'wrong

```

```

| (tweak-option x) → any/c
  x : any/c

```

If an `option/c` guards `x` and contract checking for `x` is enabled, then `tweak-option` returns `x` with contract checking for `x` disabled. If an `option/c` guards `x` and contract checking for `x` is disabled, then `tweak-option` returns `x` with contract checking for `x` enabled. In any other case it returns `x`. The result of `tweak-option` retains the guard related to `option/c` if it has one to begin with and thus its contract checking status can change further using `tweak-option`, `exercise-option` or `waive-option`.

Examples:

```

> (module server7 racket
  (require racket/contract/option)
  (provide (contract-out [bar (option/c (-> number? symbol?))]))
  (define bar (λ (x) x)))
> (require 'server7 racket/contract/option)
(define t-bar (tweak-option bar))

```

```

> (t-bar 'wrong)
bar: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
       the option of
         (option/c (-> number? symbol?))
  contract from: server7
  blaming: top-level
    (assuming the contract is correct)
  at: eval:30.0
> ((tweak-option t-bar) 'wrong)
'wrong
> ((waive-option t-bar) 'wrong)
'wrong
> ((exercise-option t-bar) 'wrong)
bar: contract violation
  expected: number?
  given: 'wrong
  in: the 1st argument of
       the option of
         (option/c (-> number? symbol?))
  contract from: server7
  blaming: top-level
    (assuming the contract is correct)
  at: eval:30.0

```

```

| (has-option? v) → boolean?
  v : any/c

```

Returns #t if `v` has an option contract.

```

| (has-option-with-contract? v) → boolean?
  v : any/c

```

Returns #t if `v` has an option contract with contract checking enabled.

```

| (invariant/c c
   invariant
   [#:immutable immutable
    #:flat? flat?
    #:struct struct-id]) → contract?
c : contract?
invariant : (-> any boolean?)
immutable : (or/c #t #f 'dont-care) = 'dont-care

```

```
flat? : boolean? = #f
struct-id : (or/c identifier? 'none) = 'none
```

Returns a contract that recognizes vectors or hashes or instances of struct `struct-id`. The data structure must match `c` and satisfy the `invariant` argument.

If the `flat?` argument is `#t`, then the resulting contract is a flat contract, and the `c` arguments must also be flat contracts. Such flat contracts will be unsound if applied to a mutable data structure, as they will not check future operations on the vector.

If the `immutable` argument is `#t` and the `c` arguments are flat contracts, the result will be a flat contract. If the `c` arguments are chaperone contracts, then the result will be a chaperone contract.

Examples:

```
> (module server8 racket
  (require racket/contract/option)
  (provide
   change
   (contract-out
    [vec (invariant/c
          any/c
          sorted?)]))
  (define vec (vector 1 2 3 4 5))
  (define (change) (vector-set! vec 2 42))
  (define (sorted? vec)
    (for/and ([el vec]
              [cel (vector-drop vec 1)])
      (<= el cel))))
> (require 'server8)
> (vector-set! vec 2 42)
vec: contract violation
  expected vector that satisfies #<procedure:sorted?> given:
  #(1 2 42 4 5)
  in: (invariant/c any/c #<procedure:sorted?>)
  contract from: server8
  blaming: top-level
  (assuming the contract is correct)
  at: eval:37.0
> (change)
> (vector-ref vec 2)
vec: broke its own contract
  expected vector that satisfies #<procedure:sorted?> given:
  #(1 2 42 4 5)
  in: (invariant/c any/c #<procedure:sorted?>)
```

contract from: server8
blaming: server8
(assuming the contract is correct)
at: eval:37.0