Racketfest 2019 - Introduction to Contracts

Version 7.2

Paulo Matos <pmatos@linki.tools>

March 22, 2019

The contract system is one of the most mature and feature-rich parts of the Racket Programming Language. This is an introductory text to the contract system and as such it will only, in some cases, provide half the truth about some details of the system. Some definitions shown might be simplified but the whole truth and all the nitty gritty details are available in the Racket Reference manual.

After this tutorial, you should be able to:

- understand what and where the contract boundaries are;
- add contracts to your code;
- and analyze the performance impact contracts have on the overall application performance.

It doesn't sound much but there is a lot of ground to cover, so let's get started.

1 Contracts - Why?

Let us start with an example. For simplicity's sake, we will assume there are no complex numbers already in Racket and instead try to implement them from scratch.

```
#lang racket

(struct complex (r i)
  #:transparent)
```

We have defined a structure `complex` to represent our complex number, which provides a function of the same name to construct them.

Examples:

```
> (complex 2 2)
(complex  2 2)
> (complex 2 1)
(complex  2 1)
> (complex 2 #false)
(complex  2 #f)
> (complex (list 2) 'broken)
(complex  '(2) 'broken)
```

Something is definitely not right because, `(complex (list 2) 'broken)` does not represent a complex number. A complex number, if you recall, has two components: one is the real part and the other is the imaginary part here denoted by the fields `r` and `i` respectively. Both of which are real numbers.

You are, of course, thinking that what we need here are type checks. And you would be right! Here is one way to do it.

```
#lang racket

(struct complex (r i)
  #:transparent)

(define (make-complex r i)
  (unless (and (real? r) (real? i))
    (error 'complex "needs real? arguments"))
  (complex r i))
```

Lets give these a try.

```
> (make-complex 2 2)
(complex  2 2)
```

```
> (make-complex 2 #false)
complex: needs real? arguments
```

This worked but we can do better... with contracts.

```
#lang racket

(struct complex (r i)
  #:transparent)

(define/contract (make-complex r i)
  (real? real? . -> . complex?)
  (complex r i))
```

Same interactions as above but using contracts instead.

```
> (make-complex 2 2)
(complex  2 2)
> (make-complex 2 #false)
make-complex: contract violation
   expected: real?
   given: #f
   in: the 2nd argument of
         (-> real? real? complex?)
   contract from: (function make-complex)
   blaming: anonymous-module
    (assuming the contract is correct)
   at: string:6.18
```

Racket contracts in case of an error, not only checks arguments and return values but it provides a much better error message. However, contracts allow to check for relation between arguments and results, specify concisely these contraints and perform runtime pre- and post-checks.

2 Contracts and Boundaries

In the previous example, by defining the function with `define/contract`, we have created a contract boundary that encapsulates the function. This means that each time the function `complex` is called or returns the contract is checked.

The contract boundary should, in general, be at the module level. There are a couple reasons for this. The first one I would like to mention is performance.

```
(define/contract (binary-complex+ c1 c2)
  (complex? complex? . -> . complex?)
  (complex (+ (complex-r c1) (complex-r c2))
           (+ (complex-i c1) (complex-i c2))))

(define/contract (complex+ . cs)
  (() () #:rest (listof complex?) . ->* . complex?)
  (for/fold ([r (complex 0 0)])
            ([c (in-list cs)])
    (binary-complex+ r c)))
```

Note how the repeated call to `binary-complex+` causes its contract to be checked repeatedly for each element passed as argument to `complex+` and yet this is redundant since it was already checked that all the elements are complex numbers.

The other reason is surprising. To understand it take a look at the definitions of `f` and `g`.

```
#lang racket

(define/contract (f x)
   (integer? . -> . integer?)
   x)
 (define/contract (g)
   (-> string?)
   (f "not an integer"))
```

Function `f`'s contract is broken when `g` calls `f` with `"not an integer"`, which is a string - not an integer. And it would be understandable and even expected to assume that the blame here should be pointed at `g`. After all, it is `g` who calls `f` with an incorrect value, therefore breaking the contract.

Example:

```
> (g)
f: contract violation
   expected: integer?
   given: "not an integer"
   in: the 1st argument of
```

```
      (-> integer? integer?)
  contract from: (function f)
  blaming: anonymous-module
   (assuming the contract is correct)
  at: string:3.18
```

The problem is that each of the function's contract are not with each other, but instead the contract is between the function itself and the top-level who mediates the calls. Therefore when `f` is called, the blame is on the top-level who is responsible for `f`'s call in the body of `g`. Using `define/contract` can potentially create confusing blames, which if you are not aware can send you on the wrong path towards debugging a problem. For these reasons, as a rule of thumb avoid using `define/contract`, use `contract-out` instead and design your modules accordingly.

```
#lang racket

(provide
  (contract-out
    [complex (real? real? . -> . complex?)]
    [complex? (any/c . -> . boolean?)]
    [complex+ (complex? ... . -> . complex?)]))

(struct complex (r i)
  #:transparent)

(define (binary-complex+ c1 c2)
  (complex (+ (complex-r c1) (complex-r c2))
           (+ (complex-i c1) (complex-i c2))))

(define (complex+ . cs)
  (for/fold ([r (complex 0 0)])
            ([c (in-list cs)])
    (binary-complex+ r c)))
```

Now the contracts for all the provided functions are defined in the header of the module and only checked when a function is evaluated from outside the module. Since we have set the contract boundary to be at the module level by using `contract-out`, in-module interactions will not be checked.

```
(define (add1 c)
  (define one (complex 1 #f))
  (complex+ c one))

> (add1 (complex 1 2))
+: contract violation
  expected: number?
```

given: #f
argument position: 2nd
other arguments...:
2

Note that the contract complains, not about the construction of the broken complex number `(complex 1 #f)` nor about the call to `complex+`. Instead, it only triggers a contract violation inside racket addition operator `+`. Again, this is to be expected since we have moved our contract boundary to the module level.

3 Function and Data Structure Contracts

In this section we will introduce some types of function and data structure contracts available in Racket.

Contracts come in two forms, those that are constructed using the operations we wills soon introduce and those builtin to Racket values, i.e. some Racket values double as contracts. For example, symbols and booleans are contracts that recognize themselves using `eq?`. Strings and characters also recognize themselves but using `equal?` instead. Numbers recognize themselves using `=`. Lastly any procedure of arity 1 is treated as a predicate and during contract checking, it is applied to the values that appear and should return `#false` if the contract fails, and anything else otherwise. For more information on racket values as contracts, please refer to the Reference on Contracts.

The rest of this section will discuss contract combinators, which take contracts and produce other contracts. There are three categories of contracts: flat contracts, chaperone contracts and impersonator contracts here we will only discuss flat contracts.

A flat contract can be fully checked for a given value, so they are essentially predicate functions.

```
#lang racket

(provide
  (contract-out
     [balance real?]))

(define balance 0)
```

This module shows that the contract for `balance` uses the predicate `real?` as a contract. You can check if a function can be used as a flat contract with `flat-contract?`.

```
> (flat-contract? real?)
#t
```

Flat contracts are therefore the atoms of the contract system. Once you know that predicates are flat contracts, you can build on them using contract combinators. The first contract combinator we will be looking at is `->`. We have seen above a simple use of this combinator: `(-> real? real? complex?)`. It creates a contract for a function that accepts two real numbers and returns a complex number.

Here are a few examples of the use of `->`.

```
(complex? ... . -> . complex?)
(complex? ... real? . -> . complex?)
(complex? . -> . (values real? real?))
```

```
(complex? . -> . void?)
```

The arrow combinator is the simplest combinator for function contracts. When using the infix notation, the function's domain comes to the left of the arrow and the function's range to the right. Ellipsis denote one or more arguments matching the last names contract. In the first example shown above the function takes as arguments one or more complex numbers and returns a complex number.

Next up in the complexity chain we have the `->*` contract combinator which allows us to specify optional keyword or positional arguments in addition to what `->` allowed us to do.

For example:

```
(->* (integer?) (boolean? #:x string?) #:rest (listof integer?)
     boolean?)
```

For the `->*` combinator, the simple case is having: mandatory arguments domain, optional arguments domain, rest argument and range. That is what our example shows. A function matching this argument would accept an integer as first argument followed by an optional boolean positional argument, an optional string keyword argument (with keyword `#:x`) and arbitrarily many integers, returning a boolean value.

If there are no optional arguments or rest arguments then we do not need to write them.

```
(->* (integer?) integer?)
(integer? . -> . integer?)
```

Both of these contracts specify a function that receives a mandatory integer and returns an integer. It is possible to specify using `->*` pre and post conditions that check the environment at the time the call is evaluated and the function returns. For more information, refer to the reference manual.

The last contract combinator on functions we will look at is the `->i` combinator. With `->i` each argument and result is named and these can be used to express dependencies between the arguments and the results.

Let's say that we have a function that adds two complex number vectors of the same length and returns a complex number vector, which has that same length. You could express it with `->i` as follows.

```
(->i ([v1 (vector/c complex?)]
      [v2 (v1) (and/c (vector/c complex?)
                      (= (vector-length v2)
                         (vector-length v1)))])
     [r (v1 v2) (and/c (vector/c complex?)
                       (= (vector-length v1)
                          (vector-lenght r)))])
```

For structure contracts, whose boundary is the module in which they were defined there are two possibilities. Either the module exports the `struct` definitions or a specific structure. There are different contract combinators for each of these options.

To export specific structure values you can use `struct/c` which imposes a contract on this specific value.

```
#lang racket

(provide
  (contract-out
    [zero-complex (struct/c complex zero? zero?)]))

(struct complex (r i)
  #:transparent)

(define zero-complex (complex 0 0))
```

4 Breaking Contracts

There is a feature in Racket that allows us to exercise a contract by automatically generating random values, pass them to a function covered by the contract and check if we get the function to break its own contract.

```
#lang racket

(define/contract (sum-list xs)
  ((listof integer?) . -> . integer?)
  (define total
    (for/fold ([s 0])
              ([x (in-list xs)])
      (+ s x)))
  (if (> total 100)
      'buh
      total))
```

We have clearly introduced a bug in `sum-list` by returning a symbol, instead of an integer when the sum of the list is longer than 100. Let's attempt to get `sum-list` to break its own contract.

```
> (contract-exercise sum-list)
sum-list: broke its own contract
  promised: integer?
  produced: 'buh
  in: the range of
        (-> (listof integer?) integer?)
  contract from: (function sum-list)
  blaming: (function sum-list)
   (assuming the contract is correct)
  at: string:3.18
```

5 Contracts Performance Analysis

Contracts, as mentioned, perform runtime checks on values. These checks can, depending on the contract, be very expensive. As such when developing a performance bound application, it is important to understand how much do contracts affect performance.

We start this section by showing how to profile contracts. We then follow with a discussion of cases where you might disable contract checks and how to do this based on a compile time value.

5.1 Profiling

Racket provides an excellent contract profiler which helps you investigate how much of your code is spent checking contracts.

The straightforward way to run the profiler is by using the command `raco contract-profile`. However, depending on how complex your application is, you might need more flexibility in which case you need the function `contract-profile-thunk`.

5.2 Disabling Contracts

The contract system is exceptional. However, sometimes the performance hit taken by using contracts is too high. Although we would like to believe that the safety of contracts *always* makes up for the any performance hit taken, that is not the case.

As a developer you decide to make the call that the performance hit by the contracts is too high and you would like to experiment with disabling them.

Disabling contracts requires compile-time variable to make the choice between disabling contracts or not and a macro to either emit the contracts or not. The code shown assumes you are using `contract-out` to define your contracts. A similar macro could be used to disable `define/contract` contracts.

```
#lang racket/base

(require (for-syntax racket/base racket/string)
         racket/contract
         racket/require-syntax racket/provide-syntax
         racket/match
         racket/list
         syntax/parse/define
         racket/struct-info)
```

```
(provide provide/opt-contract
         compiled-with-contracts?
         (all-from-out racket/contract))

(define-for-syntax enable-contracts?
  (and (getenv "ENABLE_CONTRACTS") #true))

(begin-for-syntax
  (define-syntax-class clause
    #:attributes (i)
    (pattern [(~datum struct) (~or nm:id (nm:id super:id)) (flds ...)]
             #:with i #'(struct-out nm))
    (pattern [(~datum rename) out:id in:id cnt:expr]
             #:with i #'(rename-out [out in]))
    (pattern [i:id cnt:expr])))

(define-syntax provide/opt-contract
  (if enable-contracts?
      (lambda (stx)
        (syntax-parse stx
          [(_ c:clause ...)
           #'(provide (contract-out c ...))]))
      (lambda (stx)
        (syntax-parse stx
          [(_ c:clause ...)
           #'(provide c.i ...)]))))

(define-syntax (compiled-with-contracts? stx)
  (datum->syntax stx enable-contracts?))
```

This code was based on code from Typed Racket to optionally disable contracts. If at expansion time, `ENABLE_CONTRACTS` environment variable is defined, then `provide/opt-contract` will expand to `(provide (contract-out ...))`, otherwise it will expand to `(provide ...)` ignoring the specified contracts.

6 What's next?

If you are eager to learn more, I hope I can provide you with a few pointers on where to go next.

- Contracts are closely related to two other concepts: chaperones and impersonators. If you are interested in learning how contracts are implemented and what exactly chaperones and impersonators are then take some time to read about them.
- Class Contracts: if you use or intend to use the excellent Racket object system, then you should be aware that it has contract support.
- A recent feature of the contract system are collapsible contracts, which are an optimization for the contract system. There will probably be a lot more research and development on this topic in the future so keep a close eye on this.
- Syntax contracts is another of those topics that is in an experimental phase and might see a lot more research and development in the near future - keep a close eye on this if it interests you.
- The Racket Guide is your *friend* and the Racket Reference Manual is the ultimate resource when you need more detail.