Contract Profiling

Version 6.4

February 8, 2016

This package provides support for profiling the execution of §7 “Contracts”.

Contracts are a great mechanism for enforcing invariants and producing good error messages, but they introduce run-time checking which may impose significant posts. The goal of the contract profiler is to identify where these costs are, and provide information to help control them.

The simplest way to use this tool is to use the `raco contract-profile` command, which takes a file name as argument, and runs the contract profiler on the `main` submodule of that file (if it exists), or on the module itself (if there is no `main` submodule). The tool's output is decribed below.

```
(require contract-profile)
```
package: `contract-profile`

In addition to using `raco contract-profile`, it is possible to invoke the contract profiler programmatically. This allows for profiling particular portions of programs, and for controlling the output.

```
(contract-profile option ... body ...)

option = #:module-graph-file module-graph-file
       | #:boundary-view-file boundary-view-file
       | #:boundary-view-key-file boundary-view-key-file
```

Produces a report of the performance costs related to contract checking in *body* on standard output.

Specifically, displays the proportion of *body*'s running time that was spent checking contracts and breaks that time down by contract, and then breaks down the cost of each contract between the different contracted values that use it.

Additional visualizations are available on-demand, controlled by keyword arguments which specify their destination files. An argument of `#f` (the default) disables that visualization.

- *Module Graph View*: Shows a graph of modules (nodes) and the contract boundaries (edges) between them that were crossed while running *body*.

  The weight on each contract boundary edge corresponds to the time spent checking contracts applied at this boundary. Modules written in Typed Racket are displayed in green and untyped modules are displayed in red.

  These graphs are rendered using Graphviz, and are only available if the contract profiler can locate a Graphviz install.

  When using `raco contract-profile`, controlled using the `--module-graph-file` flag.

- *Boundary View*: Shows a detailed view of how contract checking costs are spread out across contracted functions, broken down by contract boundary.

  Contracted functions are shown as rectangular nodes colored according to the cost of checking their contracts. Edges represent function calls that cross contract boundaries and cause contracts to be applied. These edges are extracted from profiling information, and therefore represent incomplete information. Because of this, the contract profiler sometimes cannot determine the callers of contracted functions. Non-contracted functions that call contracted functions across a boundary are shown as gray ellipsoid nodes. Nodes are clustered by module. Each node reports its (non-contract-related) self time. In addition, contracted function nodes list the contract boundaries the function participates in, as well as the cost of checking the contracts associated with each boundary. For space reasons, full contracts are not displayed on the graph and are instead numbered. The mapping from numbers to contracts is found in *boundary-view-key-file*.

  These graphs are rendered using Graphviz, and are only available if the contract profiler can locate a Graphviz install.

  When using `raco contract-profile`, controlled using the `--boundary-view-file` and `--boundary-view-key-file` flags.

```
(contract-profile-thunk
  thunk
 [#:module-graph-file module-graph-file
  #:boundary-view-file boundary-view-file
  #:boundary-view-key-file boundary-view-key-file])
→ any
 thunk : (-> any)
 module-graph-file : (or/c path-string #f) = #f
 boundary-view-file : (or/c path-string #f) = #f
 boundary-view-key-file : (or/c path-string #f) = #f
```

Like `contract-profile`, but as a function which takes a thunk to profile as argument.

Examples:

```
> (define/contract (sum* numbers)
    (-> (listof integer?) integer?)
    (for/fold ([total 0])
              ([n (in-list numbers)])
      (+ total n)))

> (contract-profile (sum* (range (expt 10 6))))
Running time is 23.47% contracts
231/982 ms

(-> (listof integer?) integer?)                                  230.5
ms
#<blame>:1:0
    sum*                                                         230.5
ms

499999500000
```

The example shows that a large proportion of the call to `sum*` with a list of 1 million integers is spent validating the input list.

Note that the contract profiler is unlikely to detect fast-running contracts that trigger other, slower contract checks. In the following example, there is a higher chance that the profiler samples a `(listof integer?)` contract than the underlying `(vectorof list?)` contract.

Examples:

```
> (define/contract (vector-max* vec-of-numbers)
    (-> (vectorof list?) integer?)
    (for/fold ([total 0])
              ([numbers (in-vector vec-of-numbers)])
      (+ total (sum* numbers))))

> (contract-profile (vector-max* (make-vector 10 (range (expt 10 6)))))
Running time is 80.15% contracts
2815/3512 ms

(-> (listof integer?) integer?)                                  1700
ms
#<blame>:1:0
    sum*                                                         1700
ms

(-> (vectorof (listof any/c)) integer?)                          1115
ms
eval:3:0
```

```
    vector-max*                                                1115
ms

4999995000000
```