Contract Profiling

Version 8.12.900

May 2, 2024

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 costs. 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-view-file module-graph-view-file
       | #:boundary-view-file boundary-view-file
       | #:boundary-view-key-file boundary-view-key-file
       | #:report-space-efficient? report-space-efficient?
```

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.

If *report-space-efficient?* is non-false, space-efficient contracts are marked specially

in the report. When using `raco contract-profile`, this is controlled using the `--report-space-efficient` flag.

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-view-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.

Examples:

```
> (define/contract (sum* numbers)
    (-> (listof integer?) integer?)
    (for/fold ([total 0])
              ([n (in-list numbers)])
      (+ total n)))
> (contract-profile (sum* (range (expt 10 7))))
```

```
Running time is 47.84% contracts
1094/2287 ms

(-> (listof integer?) integer?)                                    1094
ms
#<blame>::-1
    sum*                                                           1094
ms

49999995000000
> (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 7)))))
Running time is 93.76% contracts
4180/4458 ms

(-> (vectorof (listof any/c)) integer?)                            1939
ms
#<blame>::-1
    vector-max*                                                    1939
ms

(-> (listof integer?) integer?)                                    2241
ms
#<blame>::-1
    sum*                                                           2241
ms

499999950000000
```

```
(contract-profile-thunk
  thunk
  [#:module-graph-view-file module-graph-view-file
   #:boundary-view-file boundary-view-file
   #:boundary-view-key-file boundary-view-key-file
   #:report-space-efficient? report-space-efficient?])
→ any
  thunk : (-> any)
  module-graph-view-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
  report-space-efficient? : any/c = #f
```

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

Example:

```
> (contract-profile-thunk
    (lambda ()
      (sum* (range (expt 10 7)))))
Running time is 44.16% contracts
908/2056 ms

(-> (listof integer?) integer?)                                    908
ms
#<blame>::-1
    sum*                                                           908
ms

49999995000000
```