# Profile: Statistical Profiler

Version 8.9

May 7, 2023

The `profile` collection implements a statistical profiler. The profiling is done by running a background thread that collects stack snapshots either via `continuation-mark-set->context` or via Errortrace, meaning that the result is an estimate of the execution costs.

When using `continuation-mark-set->context`, it is limited to the kind of information that `continuation-mark-set->context` produces (most notably being limited to functions calls, and subject to compiler optimizations); but the result is often useful. In practice, since this method does not require recompilation of your source and has very little runtime overhead, it can be used for longer runs which compensates for these limits.

When using Errortrace, profiles are more precise and more fine-grained (expression-level instead of function-level) but profiling has higher overhead and recompilation may be necessary.

# Contents

# 1 Toplevel Interface

The profiler can be invoked directly from the command-line using the `raco profile` command, which takes a file name as argument, and runs the profiler on the `main` submodule of that file (if it exists), or on the module itself (if there is no `main` submodule).

To allow control over its behavior, `raco profile` accepts flags that correspond to those of `profile-thunk` below.

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

This module provides one procedure and one macro that are convenient high-level entry points for profiling expressions. It abstracts over details that are available through other parts of the library, and is intended as a convenient tool for profiling code.

```
(profile-thunk  thunk
                [#:delay delay
                 #:repeat iterations
                 #:threads threads?
                 #:render renderer
                 #:periodic-renderer periodic-renderer
                 #:use-errortrace? use-errortrace?
                 #:order order])                          → any/c
  thunk : (-> any/c)
  delay : (>=/c 0.0) = 0.05
  iterations : exact-nonnegative-integer? = 1
  threads? : any/c = #f
  renderer : (-> profile? (or/c 'topological 'self 'total) any/c)
           = text:render
  periodic-renderer : (or/c #f (list/c (>=/c 0.0)
                                       (-> profile?
                                           (or/c 'topological 'self 'total)
                                           any/c)))
                    = #f
  use-errortrace? : any/c = #f
  order : (or/c 'topological 'self 'total) = 'topological
```

Executes the given *thunk* and collect profiling data during execution, eventually analyzing and rendering this. Returns the value of the profiled expression. Keyword arguments can customize the profiling:

- The profiler works by starting a "sampler" thread to periodically collect stack snapshots (using `continuation-mark-set->context`). To determine the frequency of these collections, the sampler thread sleeps *delay* seconds between collections. Note that this will be close, but not identical to, the frequency in which data is actually sam-

pled. (The *delay* value is passed on to create-sampler, which creates the sampler thread.)

- Due to the statistical nature of the profiler, longer executions result in more accurate analysis. You can specify a number of *iterations* to repeat the *thunk* to collect more data.
- Normally, the sampler collects only snapshots of the current-thread's stack. Profiling a computation that creates threads will therefore lead to bad analysis: the timing results will be correct, but because the profiler is unaware of other threads the observed time will be suspiciously small, and work done in other threads will not be included in the results. To track all threads, specify a non-#f value for the *threads?* argument—this will execute the computation in a fresh custodian, and keep track of all threads under this custodian.
- Once the computation is done and the sampler is stopped, the accumulated data is analyzed (by analyze-samples) and the resulting profile value is sent to the *renderer* function. See §4 "Profile Renderers" for available renderers.
- To provide feedback information during execution, specify a *periodic-renderer*. This should be a list holding a delay time (in seconds) and a renderer function. The delay determines the frequency in which the renderer is called, and it should be larger than the sampler delay (usually much larger since it can involve more noticeable overhead, and it is intended for a human observer).
- When *use-errortrace?* is not #f, more accurate stack snapshots are captured using Errortrace. Note that when this is provided, it will only profile uncompiled files and files compiled while using errortrace-compile-handler, and the profiled program must be run using

  ```
  racket -l errortrace -t program.rkt
  ```

  Removing compiled files (with extension .zo) is sufficient to enable this.
- The *order* value is passed to the *renderer* to control the order of its output. By default, entries in the profile are sorted topologically, but they can also be sorted by the time an entry is on top of the stack ('self) or appears anywhere on the stack ('total). Some renderers may ignore this option.

(profile *expr keyword-arguments ...*)

A macro version of profile-thunk. Keyword arguments can be specified as in a function call: they can appear before and/or after the expression to be profiled.

# 2 Collecting Profile Information

```
(require profile/sampler)
```

package: profile-lib

```
(create-sampler to-track
                delay
                [super-cust
                 custom-keys
                 #:use-errortrace? use-errortrace?])
 → ((symbol?) (any/c) . ->* . any/c)
  to-track : (or/c thread? custodian?
                   (listof (or/c thread? custodian?)))
  delay : (>=/c 0.0)
  super-cust : custodian? = (current-custodian)
  custom-keys : (listof any/c) = '()
  use-errortrace? : any/c = #f
```

Creates a stack-snapshot collector thread, which tracks the given *to-track* value every *delay* seconds. The *to-track* value can be either a thread (track just that thread), a custodian (track all threads managed by the custodian), or a list of threads and/or custodians. If a custodian is given, it must be subordinate to *super-cust*, which defaults to the current custodian.

When *custom-keys* are provided, the sampler takes snapshots of the continuation marks corresponding to the given keys, in addition to taking snapshots of the stack.

When *use-errortrace?* is not `#f`, the `errortrace-key` is used to sample snapshots instead of the implicit key used by `continuation-mark-set->context`.

The resulting value is a controller function, which consumes a message consisting of a symbol and an optional argument, and can affect the sampler. The following messages are currently supported:

- `'pause` and `'resume` will stop or resume snapshot collection. These messages can be nested. Note that the thread will continue running—it will just stop collecting snapshots.
- `'stop` kills the sampler thread. It should be called when no additional data should be collected. (This is currently irreversible: there is no message to start a new sampler thread.)
- `'set-tracked!` with a value will change the tracked object(s) which were initially specified as the *to-track* argument.
- `'set-tracked!` with a numeric value will change the delay that the sampler is taking between snapshots. Note that although changing this means that the snapshots are not

uniformly distributed, the results will still be correct: the cpu time between samples is taken into account when the collected data is analyzed.

- A `'get-snapshots` message will make the controller return the currently collected data. Note that this can be called multiple times, each call will return the data that is collected up to that point in time. In addition, it can be (and usually is) called after the sampler was stopped.

  The value that is returned should be considered as an undocumented internal detail of the profiler, intended to be sent to `analyze-samples` for analysis. The reason this is not done automatically, is that a future extension might allow you to combine several sampler results, making it possible to combine a profile analysis from several individual runs, possibly from different machines.

- Finally, a `'get-custom-snapshots` message will make the controller return the currently collected snapshots corresponding to *`custom-keys`*. This returns a list of samples, where each sample is a list of vectors of marks in the same format as the output of `continuation-mark-set->list*`.

# 3 Analyzing Profile Data

```
(require profile/analyzer)
```
package: profile-lib

Once a profile run is done and the results are collected, the next step is to analyze the data. In this step sample times are computed and summed, a call-graph representing observed function calls is built, and per-node and per-edge information is created. This is the job of the main function provided by `profile/analyzer`.

```
(analyze-samples raw-sample-data) → profile?
  raw-sample-data : any/c
```

This function consumes the raw result of the sampler (given in an undocumented form), analyzes it, and returns a `profile` value holding the analyzed results. Without this function, the results of the sampler should be considered meaningless.

```
(struct profile (total-time
                 cpu-time
                 sample-number
                 thread-times
                 nodes
                 *-node))
  total-time : exact-nonnegative-integer?
  cpu-time : exact-nonnegative-integer?
  sample-number : exact-nonnegative-integer?
  thread-times : (listof (cons exact-nonnegative-integer?
                               exact-nonnegative-integer?))
  nodes : (listof node?)
  *-node : node?
```

Represents an analyzed profile result.

- `total-time` is the total observed time (in milliseconds) included in the profile run. This can be different from the actual time the profiling took, due to unaccounted-for time spent in untracked threads. (E.g., time spent in the sampler thread itself.)
- `cpu-time` is the actual cpu time consumed by the process during the profiler's work.
- `sample-number` holds the number of samples taken during the profile. This can be used to compute the average time frame each of the input samples represented.
- `thread-times` holds an association list mapping thread identifiers to cpu time for the corresponding threads. As samples are collected, each thread that is observed is assigned a small integer identifier. These identifiers are listed for each function call, and the total time spent in each thread is in this field.

- `nodes` is a list of nodes representing all observed functions (or, when using Errortrace, expressions). These nodes are the components of the call-graph that the analyzer assembles (see the `edge` field). The nodes are sorted by a topological top-to-bottom sort, and by decreasing total amount of time (time spent either in the function or in its callees) as a secondary key. It does not include the special `*-node`.
- `*-node` holds a "special" root node value that is constructed for every call graph. This node is used as the caller for all top-level function nodes and as the callee for all leaf nodes. It can therefore be used to start a recursive scan of the call graph. In addition, the times associated with its "callers" and "callees" actually represent the time these functions spent being the root of the computation or its leaf. (This can be different from a node's "self" time, since it is divided by the number of instances a function had on the stack in each sample—so for recursive functions this value is always different from the "self" time.)

```
(profile-merge profile ...) → profile?
  profile : profile?
```

This function combines multiple profiles into a single one. This is most useful when the multiple profiles are for multiple runs of the same code. The total time for the output profile is the sum of the total times of the input profiles, and the total and self times for each output profile node is the sum of its total and self times in each input profile.

```
(struct node (id src thread-ids total self callers callees))
  id : (or/c #f symbol? any/c)
  src : (or/c #f srcloc?)
  thread-ids : (listof exact-nonnegative-integer?)
  total : exact-nonnegative-integer?
  self : exact-nonnegative-integer?
  callers : (listof edge?)
  callees : (listof edge?)
```

Represents a function call (or, when using Errortrace, expression) node in the call graph of an analyzed profile result.

- The `id` and `src` fields hold a symbol naming the function (in `continuation-mark-set->context` mode) or an S-expression stub (in Errortrace mode), and/or its source location as a `srcloc` value. This is the same as the results of `continuation-mark-set->context`, so at most one of these can be `#f`, except for the special `*-node` (see the `profile` struct) that can be identified by both being `#f`.
- `thread-ids` holds a list of thread identifiers that were observed executing this function.
- `total` holds the total time (in milliseconds) that this function (or expression) was anywhere on the stack. It is common to see a few toplevel functions that have close to

a 100% total time, but otherwise small `self` times—these functions are the ones that initiate the actual work, but they don't do any hard work directly.

- `self` holds the total time (in milliseconds) that this function (or expression) was observed as the leaf of the stack. It represents the actual work done by this function, rather than the `total` time spent by both the function and its callees.
- `callers` and `callees` hold the list of callers and callees or, in Errortrace modes, expressions that step to or from the current expression. The nodes are not actually held in these lists, instead, `edge` values are used—and provide information specific to each edge in the call-graph.

```
(struct edge (total caller caller-time callee callee-time))
  total : exact-nonnegative-integer?
  caller : node?
  caller-time : exact-nonnegative-integer?
  callee : node?
  callee-time : exact-nonnegative-integer?
```

Represents an edge between two function call nodes in the call graph of an analyzed profile result or, in Errortrace mode, an edge corresponding to an evaluation step between two expressions.

- `total` is analogous to the `total` field of a `node` value: the total time that this edge was anywhere on the stack.
- `caller` and `callee` hold the two nodes that are connected by this edge.
- `caller-time` and `callee-time` hold the time spent on this edge from the caller's or the callee's perspective. These times are different from each other (as well as from the total time) because the sums that make them are each divided by the number of times the caller or the callee was on the stack.

  To understand this difference, consider a stack snapshot holding `A` → `B` → `B` → `B` → `A`, and representing a second of observed cpu time. For this sample, the `A` → `B` edge is charged by a whole second for its total time (the same goes for the `A` → `A` edge, for example). Its caller time is charged 1/2 second because `A` appears twice in this stack snapshot (in the other half, `A` is charged for being a leaf — the caller of the special `*-node`), and its callee time is charged 1/3 respectively.

# 4 Profile Renderers

After collecting the profile samples and analyzing the data, the last step of the profiling process is to render the results. The profile collection provides several renderers, each providing a rendering function that consumes a `profile` instance. See the analyzer section for a description of the `profile` struct if you want to implement a new renderer.

## 4.1 Textual Rendering

```
(require profile/render-text)      package: profile-lib
```

```
(render  profile-data
         [order
          #:truncate-source truncate-source
          #:hide-self hide-self%
          #:hide-subs hide-subs%])          → void?
  profile-data : profile?
  order : (or/c 'topological 'self 'total) = 'topological
  truncate-source : exact-nonnegative-integer? = 50
  hide-self% : (between/c 0 1) = 1/100
  hide-subs% : (between/c 0 1) = 1/50
```

Prints the given `profile` results as a textual table.

The printout begins with general information about the profile, followed by a table with an entry for each node in the call graph. The entries are displayed in a topological order by default (roughly, since the graph can have cycles). This means that it is usually easy to find the callers and callees of a function in its close environment.

Each row in the table has the following format:

```
                          B [M1] M2%
[N1] N2(N3%) N4(N5%)   A ...path/to/source.rkt:12:34
                          C [M3] M4%
```

with the following meaning of the numbers and labels:

- `A` — the name of the function (or a stub of the expression) that this node represents, followed by the source location for the function if it is known. The name can be "???" for anonymous functions, which will be identified with their source location.
- `N1` — an integer label associated with this node in the printout. This label is used to mark references to this function/expression, since symbolic names are not unique (and they can be missing or very long). The labels are assigned from the top.

- `N2` — the time (in milliseconds) that this function/expression has been anywhere in a stack snapshot. This is the total time that the execution was somewhere in this function/expression or in its callees. (Corresponds to the `node-total` field.)
- `N3` — this is the percentage of the node's total time (`N2`) from the total observed time of the profile. An entry with a 100% refers to a function/expression that was active throughout the whole execution.
- `N4` — the time (in milliseconds) that this function/expression has been at the top of the stack snapshot. This is the time that this function/expression was itself doing work rather than calling other functions/expressions. (Corresponds to the `node-self` field.)
- `N5` — this is the percentage of `N4` out of the total observed time of the profile. Functions/expressions with high values here can be good candidates for optimization. But, of course, they can represent doing real work for a caller that needs to be optimized.
- `B` and `C` — these are labels for the callers and callees of the function/expression. Any number of callers and callees can appear here (including 0). The function/expression itself can also appear in both places if it is (non-tail) recursive.
- `M1` and `M3` — the index numbers for `B` and `C`. They can be used to disambiguate functions with the same name, as well as a quick way to find the corresponding entry in the table.
- `M2` and `M4` — the percentages of the time `A` spent being called by `B` and calling `C`. These percentages represent the time that this edge was found on a stack snapshot, divided by the number of occurrences of `A` on the same snapshot. The number is the percentage of these times out of `N2`, the total time `A` has been active.

  The total percentages for the all caller and for all callees should be close to 100% minus the time `A` was the leaf or the root.

  These values correspond to the `edge-caller-time` and `edge-callee-time` fields; see the documentation for further details.

The function has a few keyword arguments to customize its output:

- The *`truncate-source`* argument determines the length that the source string should take (together with its label).
- *`hide-self%`* and *`hide-subs%`* control hiding some of the nodes. A node is hidden if its self time (`N3` in the above example) is smaller than *`hide-self%`* *and* if all places where it occurs as a caller or a callee have percentages that are smaller than *`hide-subs%`*. The reason for requiring both conditions is to avoid having "dangling references" to hidden nodes.
- The *`order`* argument determines the order in which entries appear in the output. If *`order`* is `'topological` (the default), entries are sorted topologically, grouping callers and callees close together. If *`order`* is `'self`, entries are sorted by how often they appear at the top of a stack snapshot. If *`order`* is `'total`, entries are sorted by how often they appear anywhere in a stack snapshot.

## 4.2 Graph Rendering

```
(require profile/render-graphviz)
```
package: profile-lib

```
(render  profile-data
         [order
          #:hide-self hide-self%
          #:hide-subs hide-subs%]) → void?
  profile-data : profile?
  order : (or/c 'topological 'self 'total) = 'topological
  hide-self% : (between/c 0 1) = 1/100
  hide-subs% : (between/c 0 1) = 1/50
```

Prints the given `profile` results as a Graphviz directed graph.

This is an experimental module, provided mostly as a proof-of-concept. It renders the profile's call-graph as a graph representation for one of the Graphviz tools to render. Nodes are colored according to their 'self' percentages, and edges.

The keyword arguments control hiding nodes in the same way as with the textual renderer. The *order* argument is ignored.

## 4.3 JSON Rendering

```
(require profile/render-json)
```
package: profile-lib

```
(profile->json profile-data) → jsexpr?
  profile-data : profile?
```

Converts the given `profile` results to a `jsexpr?`, which can be converted to JSON format using the `json` package.

This module is intended for transmitting profile information between threads or machines. It does not fully preserve source locations, and of course if those source locations are transmitted between machines they may not match local file names.

```
(json->profile json-data) → profile?
  json-data : jsexpr?
```

The inverse of the `profile->json` function, transforming an `jsexpr?` read from a JSON file back into a Racket `profile` object. It is not a perfect inverse (source locations may not be preserved) but can be used for transmitting profile information between different processes or machines.