Profile: Statistical Profiler

Version 5.1.3

August 15, 2011

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

Contents

1 Toplevel Interface

```
(require profile)
```

This module provides one procedure and one macro that are convenient high-level entry points for timing expressions. This hides the 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]) → void?
  thunk : (-> any/c)
  delay : nonnegative-number? = 0.05
  iterations : exact-nonnegative-integer? = 1
  threads? : any/c = #f
  renderer : (profile? . -> . any/c) = text:render
  periodic-renderer : (or/c #f (list/c nonnegative-number? (profile? . -> . any/c)))
                      = #f
```

Executes the given thunk while collecting profiling data, and render this data when done. Keyword arguments can customize the profiling:

- The profiler works by `create-sampler` starting a "sampler" thread whose job is to collect stack samples periodically (using `continuation-mark-set->context`). *delay* determines the amount of time the sampler `sleep`s for between samples. Note that this is will be close, but not identical to, the frequency in which data is actually sampled.

- When the profiled computation takes a short amount of time, the collected data will not be accurate. In this case, you can specify an *iterations* argument to repeat the evaluation a number of times which will improve the accuracy of the resulting report.

- Normally, the sampler collects snapshots of the `current-thread`'s stack. If there is some computation that happens on a different thread, that work will not be reflected in the results: the only effect will be suspiciously small value for the observed time, because the collected data is taking into account the cpu time that the thread actually performed (it uses `current-process-milliseconds` with the running thread as an argument). Specifying a non-`#f` value for the *threads?* argument will arrange for all threads that are started during the evaluation to be tracked. Note that this means that the computation will actually run in a new sub-custodian, as this is the only way to be able to track such threads.

- Once the computation has finished, the sampler is stopped, and the accumulated data is collected. It is then analyzed by `analyze-samples`, and the analyzed profile data is fed into a renderer. Use an identity function (`values`) to get the analyzed result, and render it yourself, or use one of the existing renderers (see §4 "Profile Renderers").

- The *periodic-renderer* argument can be set to a list holding a delay time and a renderer. In this case, the given renderer will be called periodically. This is useful for cases where you want a dynamically updated display of the results. This delay should be larger than the sampler delay.

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

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

2   Collecting Profile Information

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

```
(create-sampler to-track delay [super-cust])
 → ((symbol?) (any/c) . ->* . any/c)
  to-track : (or/c thread? custodian?
                   (listof (or/c thread? custodian?)))
  delay : nonnegative-number?
  super-cust : custodian? = (current-custodian)
```

Creates a sample 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.

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 data collection. These messages can be nested. Note that the thread will continue running it will just stop collecting snapshots.

- `'stop` kills the controlled 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 objects (initially specified as the `to-track` argument) to the given value.

- `'set-tracked!` with a value will change the delay that the sampler us taking between snapshots. Note that although changing this means that the snapshots are not uniformly distributed, the results will still be sensible — this is because the cpu time between samples is taken into account when the resulting data is analyzed.

- Finally, 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 thatis 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, 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.

3 Analyzing Profile Data

```
(require profile/analyzer)
```

Once a profile run is done, and the results are collected, the next step is to analyze the data. In this step the sample time are computed and summed, a call-graph representing the 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 (which is given in an undocumented form), analyzes it, and returns a `profile` value holding the analyzed results. Without this function, the results of the sampler are meaningless.

```
(struct profile (total-time
                  cpu-time
                  sample-number
                  thread-times
                  nodes
                  *-node)
         #:extra-constructor-name make-profile)
  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 the analyzed profile result.

- `total-time` is the total observed time (in milliseconds) included in the profile. This is different than the actual time the profiling took, due to unaccounted-for time spent in untracked threads. (E.g., 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. 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.

- `*-node` holds a "special" node value that is constructed for every 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 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 for every sample — so for recursive functions this value is different from.)

```
(struct node (id src thread-ids total self callers callees)
        #:extra-constructor-name make-node)
  id : (or/c #f symbol?)
  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 node in the call graph of an analyzed profile result.

- The `id` and `src` field hold a symbol naming the function and/or its source location as a `srcloc` value. This is the same as the results of `continuation-mark-set->context`, so at most of of these can be `#f`, except for the special `*-node` (see the `profile` struct) that can be identified by both of these being `#f`.

- `thread-ids` holds a list of thread identifiers that were observed executing this function.

- `total` holds the total time (in milliseconds) where this function 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 derive the work that was done, but they don't do any hard work directly.

- `self` holds the total time (in milliseconds) where this function was observed as the leaf of the stack. It represents the actual work done by this function, rather than `total` that represents the work done by both the function and its callees.

- `callers` and `callees` hold the list of caller and callee nodes. The nodes are not actually held in these lists, instead, `edge` values are used — and provide information specific to an edge in the call-graph.

```
(struct edge (total caller caller-time callee callee-time)
        #:extra-constructor-name make-edge)
  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.

- `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 chared 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 aspect of profiling 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 your own renderer.

4.1   Textual Rendering

```
(require profile/render-text)
```

```
(render  profile-data
         [#:truncate-source truncate-source
          #:hide-self hide-self%
          #:hide-subs hide-subs%])              → void?
  profile-data : profile?
  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 some general facts about the profile, and then a table that represents the call-graph is printed. Each row in this table looks like:

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

Where actual numbers appear in the printout. The meaning of the numbers and labels is as follows:

- `A` — the name of the function that this node represents, followed by the source location for the function if it is known. The name can be "???" for functions with no identifier, but in this case the source location will identify them.

- `N1` — an index number associated with this node. This is important in references to this function, since the symbolic names are not unique (and some can be missing). The number itself has no significance, it simply goes from 1 up.

- `N2` — the time (in milliseconds) that this function has been anywhere in a stack snapshot. This is the total time that the execution was somewhere in this function or in its callees. (Corresponds to the `node-total` field.)

9

- `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 that was active throughout the whole execution.

- `N4` — the time (in milliseconds) that this function has been at the top of the stack snapshot. This is the time that this function consumed doing work itself rather than calling other functions. (Corresponds to the `node-self` field.)

- `N5` — this is the percentage of `N4` out of the total observed time of the profile. Functions with high values here can be good candidates for optimization, But, of course, they can represent doing real work due to one of its callers that need to be optimized.

- `B` and `C` — these are labels for the callers and callees of the function. Any number of callers and callees can appear here (including 0). The function 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.

4.2   Graph Rendering

```
(require profile/render-graphviz)
```

```
(render  profile-data
        [#:hide-self hide-self%
         #:hide-subs hide-subs%]) → void?
  profile-data : profile?
  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.