Futures Visualizer

Version 9.4.0.3

November 1, 2026

```
(require future-visualizer)
```
package: `future-visualizer`

§20.2 “Parallelism with Futures” in *The Racket Guide* introduces

The *futures visualizer* is a graphical profiling tool for parallel programs written using `future`. The tool shows a timeline of a program’s execution including all future-related events, as well as the overall amount of processor utilization at any point during the program’s lifetime.

```
(visualize-futures e ...)
(visualize-futures-thunk thunk) → any
  thunk : (-> any)
```

The `visualize-futures` macro enables the collection of data required by the visualizer and displays a profiler window showing the corresponding trace. The `visualize-futures-thunk` provides similar functionality where program code is contained within *thunk*.

A typical program using profiling might look like the following:

```
(require racket/future
         future-visualizer)

(visualize-futures
 (let ([f (future (lambda () ...))])
   ...
   (touch f)))
```

The preceding program is equivalent to:

```
(require racket/future
         future-visualizer/trace
         future-visualizer)
```

```
(start-future-tracing!)
(let ([f (future (lambda () ...))])
  ...
  (touch f))
(stop-future-tracing!)
(show-visualizer)
```

```
(show-visualizer #:timeline timeline) → void?
  timeline : (listof indexed-future-event?)
```

Displays the visualizer window. If the function is called with no arguments, it must be preceded by the following sequence: a call to `start-future-tracing!`, program code that is being traced, and a call to `stop-future-tracing!` – in which case the visualizer will show data for all events logged in between those calls (via `timeline-events`). Note that `visualize-futures` and `visualize-futures-thunk` are simpler alternatives to using these primitives directly. The *`timeline`* argument can be used to show the visualizer for a previously-generated trace.

1 Execution Timeline

The *execution timeline*, shown in the top left-hand corner of the profiler window, displays a history of the program and all events associated with its futures, with OS-level threads or *processes* organized along the y-axis and time increasing along the x-axis. Garbage collections are shown as translucent maroon bars spanning the height of the timeline. A coloring convention is used to distinguish between different types of events (see §11.4.3 "Future Performance Logging" for a full description of these event types):

- Blue dot: `'create`
- Green bar: `'start-work`, `'start-0-work`
- Orange dot: `'sync`
- Red dot: `'block`, `'touch`
- White dot: `'result`, `'end-work`
- Green dot: `'touch-pause`, `'touch-resume`
- Maroon bar: `'gc`

Mousing over any non-GC event connects it via purple lines to the sequence of events for its future. Additionally, orange dotted lines with arrowheads may be shown to indicate operations performed from one future to another (e.g. `'create` or `'touch` actions). To view details about two events simultaneously, a selection can be tacked by clicking the mouse.

The timeline displays vertical lines at 100-microsecond intervals. Note that though the time interval is fixed, the pixel distance between lines varies based on the event density for any given time range to prevent overlapping event circles.

```
(timeline-pict  events
                [#:timeline-width timeline-width
                 #:x x
                 #:y y
                 #:width width
                 #:height height
                 #:selected-event-index selected-event-index])
 → pict?
  events : (listof indexed-future-event?)
  timeline-width : exact-nonnegative-integer? = 1000
  x : (or #f exact-nonnegative-integer?) = #f
  y : (or #f exact-nonnegative-integer?) = #f
  width : (or #f exact-nonnegative-integer?) = #f
  height : (or #f exact-nonnegative-integer?) = #f
  selected-event-index : (or #f exact-nonnegative-integer?) = #f
```

Returns a `pict` showing the execution timeline for the trace in *`events`*. The optional *`timeline-width`* provides a minimum width of the timeline. The optional arguments *`x`*, *`y`*, *`width`*, and *`height`* can be used to obtain a specific area (in pixels) of the timeline image. The *`selected-event-index`* argument, if specified, shows the timeline image as if the user placed the mouse pointer over the `indexed-future-event` with the corresponding index.

Changed in version 1.1 of package `future-visualizer`: Added the `#:timeline-width` argument.

2 Future Creation Tree

The *creation tree* shows a tree with a single node per future created by the program. This display can be particularly useful for programs which spawn futures in nested fashion (futures within futures). For any given future node, the children of that node represent futures which were created by that future (within the scope of its thunk). For all programs, the root of the tree is a special node representing the main computation thread (the runtime thread), and is denoted *RTT*.

```
(creation-tree-pict  events
                     [#:x x
                      #:y y
                      #:width width
                      #:node-width node-width
                      #:height height
                      #:padding padding
                      #:zoom zoom])            → pict?
  events : (listof indexed-future-event?)
  x : (or #f exact-nonnegative-integer?) = #f
  y : (or #f exact-nonnegative-integer?) = #f
  width : (or #f exact-nonnegative-integer?) = #f
  node-width : (or #f exact-nonnegative-integer?) = #f
  height : (or #f exact-nonnegative-integer?) = #f
  padding : (or #f exact-nonnegative-integer?) = #f
  zoom : exact-nonnegative-integer? = 1
```

Returns a `pict` showing the future creation tree for the trace in *`events`*. The optional arguments *`x`*, *`y`*, *`width`*, and *`height`* can be used to obtain a specific area (in pixels) of the creation tree image. The *`node-width`* argument specifies (in pixels) the diameter of each node. The *`padding`* argument specifies the minimum space vertically between each depth and horizontally between siblings. The *`zoom`* argument specifies the zoom factor for the tree image in the range 1-5, where 5 returns a 500% zoom.

3 Futures Tracing

```
(require future-visualizer/trace)
      package: future-visualizer-pict
```

The *futures trace* module exposes low-level information about the execution of parallel programs written using `future`.

```
(trace-futures e ...)
(trace-futures-thunk thunk) → (listof indexed-future-event?)
  thunk : (-> any)
```

The `trace-futures` macro and `trace-futures-thunk` function track the execution of a program using futures and return the program trace as a list of `indexed-future-event` structures.

This program:

```
(require racket/future
         future-visualizer/trace)

(trace-futures
 (let ([f (future (lambda () ...))])
   ...
   (touch f)))
```

Is equivalent to:

```
(require racket/future
         future-visualizer/trace)

(start-future-tracing!)
(let ([f (future (lambda () ...))])
  ...
  (touch f))
(stop-future-tracing!)
(timeline-events)
```

```
(start-future-tracing!) → void?
(stop-future-tracing!) → void?
(timeline-events) → (listof indexed-future-event?)
```

The `start-future-tracing!` procedure enables the collection of future-related execution data. This function should be called immediately prior to executing code the programmer wishes to profile.

The `stop-future-tracing!` procedure must be used to indicate the end of code the programmer wishes to trace. Tracing works by simply using a log receiver to record all future-related log events; this procedure logs a special message that is well-known to the log receiver to mean 'stop recording'.

The `timeline-events` procedure returns the program trace as a list of `indexed-future-event` structures.

```
(struct indexed-future-event (index event)
    #:extra-constructor-name make-indexed-future-event)
  index : exact-nonnegative-integer?
  event : any
```

Represents an individual log message in a program trace. In addition to future events, the tracing code also records garbage collection events; hence the `event` field may contain either a `future-event` or gc-info prefab struct (see §16.4 "Garbage Collection"), where the latter describes a GC operation. Because multiple `future-event` structures may contain identical timestamps, the `index` field ranks them in the order in which they were recorded in the log output.

```
(struct future-event (future-id
                      proc-id
                      action
                      time-id
                      prim-name
                      user-data)
    #:extra-constructor-name make-future-event
    #:prefab)
  future-id : (or exact-nonnegative-integer? #f)
  proc-id : exact-nonnegative-integer?
  action : symbol?
  time-id : real?
  prim-name : (or symbol? #f)
  user-data : (or #f symbol? exact-nonnegative-integer?)
```

Represents a future event as logged by the run-time system. See §11.4.3 "Future Performance Logging" for more information.