A Year of Rapid Cross-Platform Application Development with Racket

Dominik Pantůček <dominik.pantucek@trustica.cz>

Trustica

5. 10. 2019

Agenda

- Racket – a modern Scheme dialect
- Embedded computer vision for manufacturing QA
- Desktop applications with USB hot-plug handling
- Big numbers crunching

- A dialect of Scheme which is a dialect of LISP
- `https://racket-lang.org/`
- Solve problems · make languages
- Useful languages:
  - Racket
  - Scribble – used for documenting the code
  - Typed Racket

- Less useful languages:
  - Algol 60
  - Lazy Racket
  - R6RS: Scheme
- How to Design Programs Languages:
  - Beginning Student
  - Beginning Student with List Abbreviations
  - Intermediate Student
  - Intermediate Student with Lambda
  - Advanced Student

Hello Racket

- hello.rkt:

```
#lang racket
(displayln "Hello Racket!")
```

- Run the program:

```
$ racket hello.rkt
Hello Racket!
$
```

- Everything is a S-Expression (sexp for short)
  - Delimited by parentheses `()`, brackets `[]`, or curly brackets `{}`
- The source code maps 1:1 to the Abstract Syntax Tree (AST)

- Define – bind value to a name:

```
(define something 123)
```

- Create anonymous function:

```
(lambda (args ...) thunk ...)
(λ (args ...) thunk ...)
```

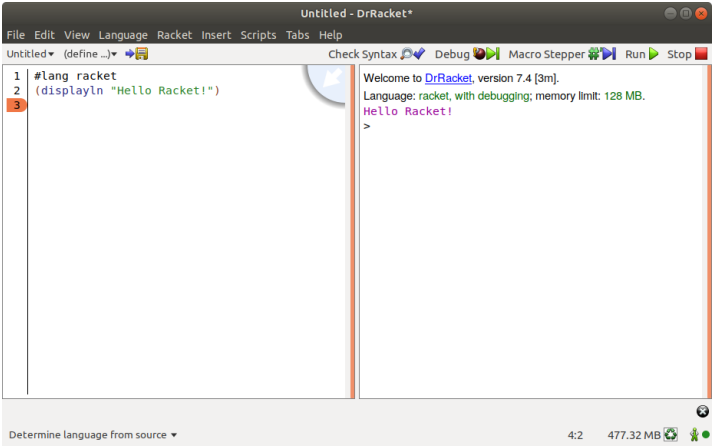
- Define and call a function

```
(define my-func (λ (args ...) thunk ...))
(my-func arg-values ...)
```

- Shorthand for defining named function:

```
(define (args ...) thunk ...)
```

DrRacket


Untitled - DrRacket*
File Edit View Language Racket Insert Scripts Tabs Help
Untitled (define ...)
Check Syntax Debug Macro Stepper Run Stop
1 #lang racket
2 (displayln "Hello Racket!")
3
Welcome to DrRacket, version 7.4 [3m].
Language: racket, with debugging; memory limit: 128 MB.
Hello Racket!
>
Determine language from source
4:2
477.32 MB

Backends

- Racket's own VM – traditional CGC
  - Conservative Garbage Collector
- Racket's own VM – the new 3m variant
  - Moving Memory Manager
- Chez Scheme
  - `https://www.scheme.com/`
  - `https://cisco.github.io/ChezScheme/`

- GNU/Linux
- macOS X
- NetBSD
- Windows
- ...

- Creating a structure type:

```
(struct my-struct (a b c) [#:mutable])
```

- Creating structure:

```
(define my-data (my-struct 1 2 3))
```

- Accessing structure fields:

```
(displayln (my-struct-a my-data))
(displayln (my-struct-b my-data))
(displayln (my-struct-c my-data))
```

- Modifying structure fields:

```
(set-my-struct-a! my-data 4)
```

Classes

- Yes, like `structs` but with methods and slightly different syntax
- Creating new class:

```
(define my-class%
  (class superclass
    (field a)
    (init-field (b #f))
    (super-new)
    (define/public (a-method args ...)
      thunk ...)))
```

- The classes form a tree with `object%` class as its root
- Naming convention is that class names should end with % character

- Interface specifies what the class should implement
- Example interface definition:

```
(define my-interface<%>
  (interface (interfaces ...)
    methods ...))
```

- The `interfaces ...` list can be empty
- Naming convention is that interface names should end with <%>

- Creating new class which implements some interfaces:

```
(define my-class%
  (class* object% (interfaces ...)
    (field a)
    (init-field (b #f))
    (super-new)
    (define/public (a-method args ...)
      thunk ...)))
```

- Objects are instances of classes
- Creating new object:

```
(define my-object
  (new my-class%
    ((a 1)
     (b 2))))
```

- You can get field value:

```
(displayln (get-field my-object a))
```

- You can also set field value:

```
(set-field! my-object a 10)
```

- And you can call any public method of given object

```
(send my-object a-method args ...)
```

- Used by `racket/draw` and `racket/gui` libraries

- Racket on Raspberry Pi
- Acquiring images
- Accessing image pixels
- Colorspace conversions
- Finding spotlights
- Affine transformations
- Matching colors
- Exporting results

- Raspbian
- Ubuntu
- Nightly builds for ARM work on both!
- Ubuntu x86 (_64) – PPA with latest Racket

- Platform specific binary: `raspistill`
- `racket/gui`'s `bitmap%` can load various formats: BMP, PNG, JPG ...
- Let's be lazy:

```
(define (raspistill->bitmap)
  (define tmpfname (~a "/tmp/img" (random) ".png"))
  (system (~a "raspistill ... -o " tmpfname))
  (define bmp (make-object bitmap% tmpfname))
  (delete-file tmpfname)
  bmp)
```

Embedded CV: Accessing Image Pixels

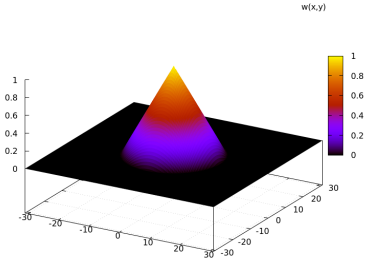
- `bitmap%` allows only drawing onto its canvas
- No way to read pixels
- We can get pixels data using the `(get-argb-pixels)` method
- Let's implement a pixel reader:

```
(define (bitmap-reader bmp)
  (define w (send bitmap get-width))
  (define h (send bitmap get-height))
  (define argb-bytes (make-bytes (* w h 4)))
  (send bitmap get-argb-pixels 0 0 w h argb-bytes)
  (λ (x y)
    (define offset (* 4 (+ (* w y) x)))
    (for/list ((i (in-range 1 4)))
      (bytes-ref argb-bytes (+ off i)))))
```

- We need to do some fuzzy color matching in HSV colorspace
- Let's convert the RGB values to HSV:

```
(define (rgb->hsv rgb)
  (define-values (r g b) (apply values rgb))
  (define minimum (min r g b))
  (define v (max r g b))
  (define delta (- v minimum))
  (define s (/ delta v))
  (define h (* 60 (cond
        ((= r v) (/ (- g b) delta))
        ((= g v) (+ (/ (- b r) delta) 2))
        (else (+ (/ (- r g) delta) 4)))))
  (list h s v))
```

- Finding spotlights:
  - Tresholding pixels based on HSV `value`
  - Averaging pixel positions, HSV `value` is the weight
- Blurred color picker:
  - Weighted average of HSV components from given area
  - Weight decreases with the distance from center
  - $w_{x,y} = 1 - \frac{\sqrt{(x-x_0)^2+(y-y_0)^2}}{r}$
- Matching colors:
  - Is trivial in 2D HS part of HSV space



Embedded CV: In Action

h = 353.53
s = 0.46
v = 227.45
n = 0.43
r = 255
g = 0
b = 27
LED: red

h = 30.61
s = 0.66
v = 233.20
n = 0.34
r = 255
g = 130
b = 0
LED: orange

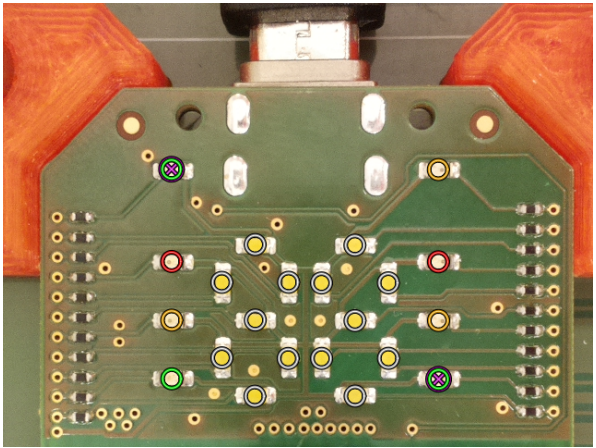
h = 142.03
s = 0.33
v = 231.30
n = 0.44
r = 0
g = 255
b = 94
LED: green

- Affine transform in the form of $P' = M \times P$, where:
  - $P = [P_x, P_y]$ – point in original coordinates
  - $P' = [P'_x, P'_y]$ – point in transformed coordinates
  - $M = \begin{bmatrix} M_{(0,0)} & M_{(1,0)} \\ M_{(0,1)} & M_{(1,1)} \end{bmatrix}$ – transformation matrix
- M can be computed from two points A and B in original coordinates with known A' and B' in transformed coordinates
- Trivial in Racket

- Create a function accepting A, B, A', and B', returning transformation function:

```
(define (make-point-transformer A A- B B-)
  (define M (compute-transformation-matrix A A- B B-))
    (λ (P)
      (multiply-point-by-matrix M P)))
```

Embedded CV: Affine Transformations In Action



- Global variables can give you a headache
- Parameters are global but bound to context (for example to a thread)
- Creating parameters:

```
(define my-parameter (make-parameter #f))
```

- Using parameters:

```
(displayln (my-parameter))
```

- Modifying parameters

```
(parameterize ((my-parameter 123))
  thunk ...)
```

Hash Maps

- Associative array – as in most programming languages used nowadays
- Create:

```
(define my-hash (make-hash))
```

- Add or modify elements:

```
(hash-set! my-hash 'key "value")
```

- And get them:

```
(displayln (hash-ref my-hash 'key))
```

- Be lazy – upload some JSON somewhere
- Generating JSON – JS Expressions – jsexps
- JSON library is installed with default Racket installation:

```
(require json)
```

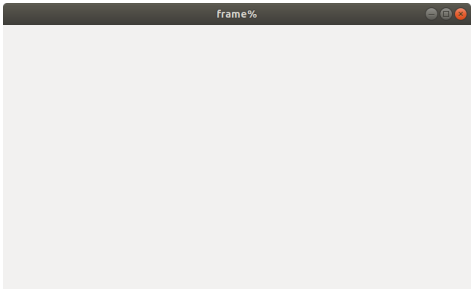
- We use any JSON-compatible expression – like hash map
- And with the help of parameters:

```
(call-with-output-file output.json #:exists 'replace
  (λ (out)
    (parameterize ((current-output-port out))
      (displayln (jsexpr->string data)))))
```

- GUI in Racket
  - Windows
  - Buttons
  - Messages
  - Layout control
- Foreign Function Interface – cross-platform hidapi bindings

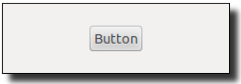
- Top-level window – a `frame%`:

```
#lang racket/gui
(define win
  (new frame%
       (label "frame%")
       (width 640)
       (height 360)))
(send win show #t)
```



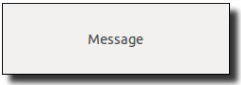
- Clickable button – a `button%`:

```
(define button
  (new button%
       (parent panel)
       (label "Button")))
```



- Informative label – a `message%`:

```
(define message
  (new message%
       (parent panel)
       (label "Message")))
```

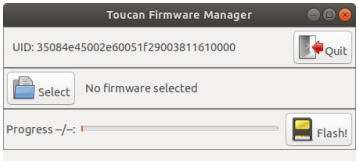


- Stack widgets vertically in `vertical-pane%`:

```
(define top-vbox
  (new vertical-pane%
       (parent this)))
```

- Stack widgets horizontally in `horizontal-pane%`:

```
(define toolbar
  (new horizontal-panel%
       (parent top-vbox)
       (border 1)
       (style '(border))
       (stretchable-height #f)))
```



- Mostly two libraries are needed:

```
(require ffi/unsafe
         ffi/unsafe/define)
```

- "Unsafe"by nature – you can directly access program resources (like memory)
- You can:
  - Access binary structures
  - Call library functions
  - Manage memory buffers for foreign functions
  - Call back to Racket code from foreign code

- Loading dynamic library:

```
(define hidapi-lib
  (ffi-lib "libhidapi-libusb.so"
           #:fail (λ ()
                    (ffi-lib "libhidapi-0.dll"))))
```

- If no #:fail argument is provided – raises an exception on error
- Defining C-style structure:

```
(define-cstruct _hid_device_info
  ([path                  _bytes]
   [vendor-id             _uint16]
    ...))
```

- Defining a definer:

```
(define-ffi-definer define-hidapi
  hidapi-lib
  #:make-c-id convention:hyphen->underscore
  #:default-make-fail make-not-available)
```

- Binding a function using previously defined definer:

```
(define-hidapi hid-enumerate
  (_fun _uint16 _uint16
   -> _hid_device_info-pointer/null))
```

- Sort 2^{27} numbers
  - $134217728 \approx 134M$
- List
- Fixnums
- Futures

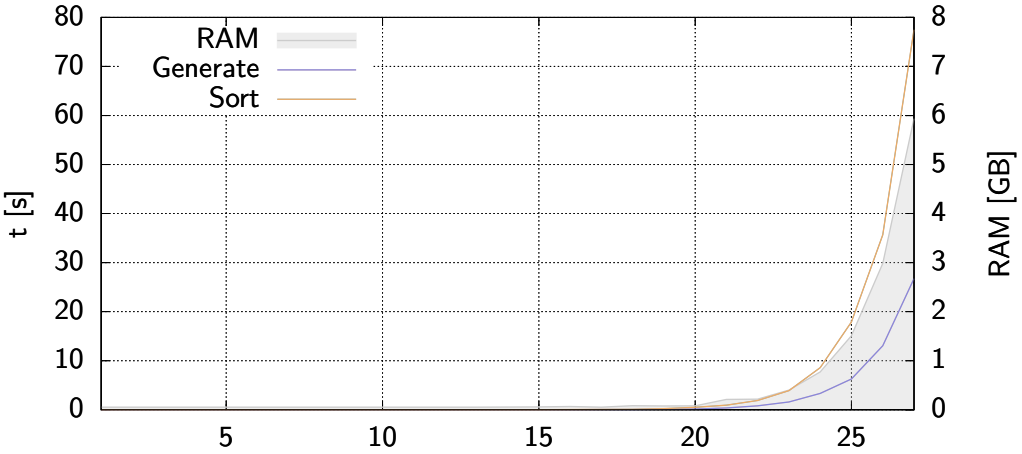
- Generating the list:

```
(define expn 27)
(define lst
  (for/list ((i (expt 2 expn)))
    (random 1000000000)))
```

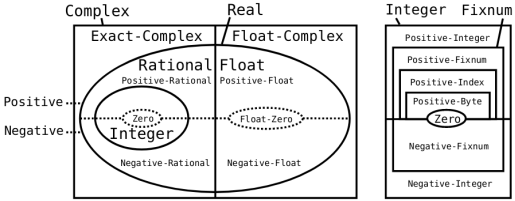
- Sorting the list:

```
(define lst-sorted (sort lst <))
```

Sorting a List – Performance



- A `fixnum?` is a number $n \in \langle -2^{62}, 2^{62} - 1 \rangle$ on 64bit platforms
- Stored in 64 bits = 8 bytes
- Arithmetic operations are faster with fixnums than with other numbers types
- `+` → `fx+`, `-` → `fx-`, $<$ → `fx<`, ...
- Numeric tower[1]:

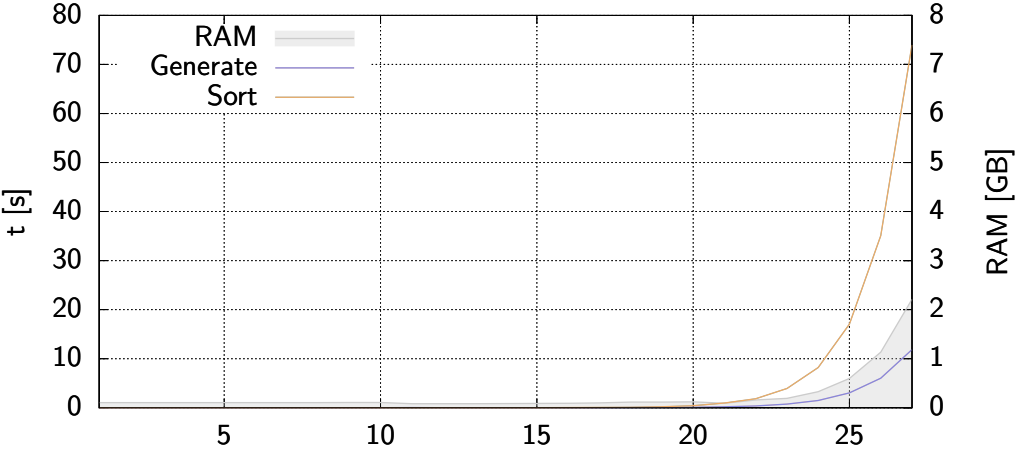


- Assuming the value is a fixnum means we can bypass type checking
- These operations are deemed "unsafe" and can be pulled-in by:

```
(require racket/unsafe/ops)
```

- All fixnum operations get their unsafe versions:
  - `unsafe-fx+`
  - `unsafe-fx-`
  - `unsafe-fx<`
  - ...
- Merge-sort implementation using fixnums

Sorting Fixnums – Performance



Futures

- Threads in Racket are (almost) thread safe
- Explicit parallel execution is not simple
- Future – an expression for which we will want evaluated value later:

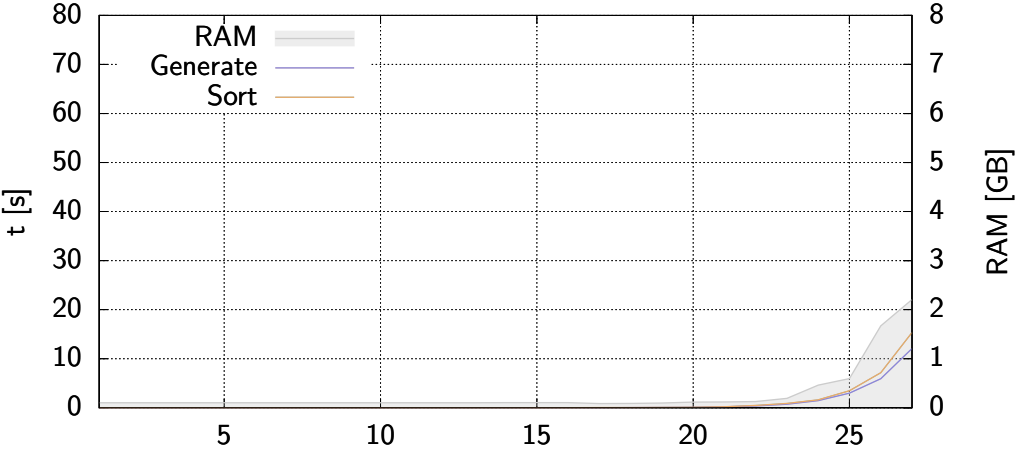
```
(define f (future (λ () (+ 1 2))))
```

- Touching the future yields its value

```
(displayln (touch f))
```

- Racket will try its best to have the value ready *before* it is touched

Sorting Fixnums with Futures – Performance



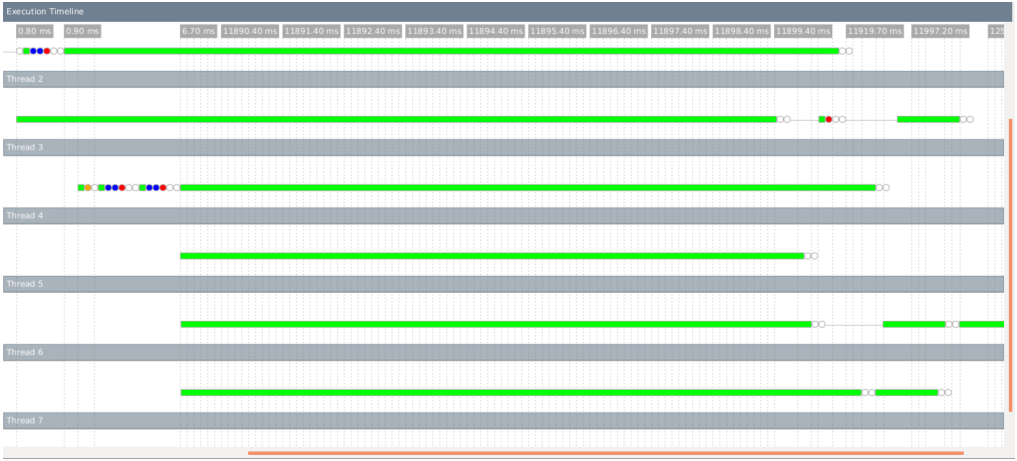
- Useful GUI library: future-visualizer:

```
(require future-visualizer)
```

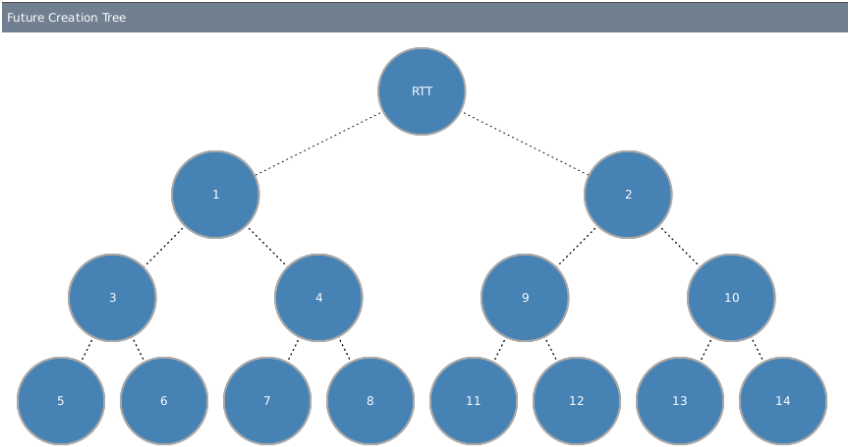
- Wrap your code in `visualize-futures` form:

```
(visualize-futures
  (fxvector-futures-sort! fxvec))
```

Futures Analysis – Timeline



Futures Analysis – Creation Tree



- [1] Typing the Numeric Tower, Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen, Northeastern University, University of Utah, Proceedings of PADL 2012
- [2] Racket: solve problems · make languages, available online at `https://racket-lang.org/`

Questions...

... and answers.

Thank you.

Now you know why you should become a Racketeer.