Feature-Specific Profiling

Vincent St-Amour

Leif Andersen

Matthias Felleisen

PLT @ Northeastern University

CC 2015 — April 18th, 2015

```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samples (quotient n-samples 12))
    (for/array #:shape (vector n-samples) #:fill 0.0
               ([i      (in-range n-different-samples)]
                [sample (in-producer random-sample (lambda _ #f))]
                #:when #t
                [j (in-range 12)])
      sample)))
(define snare
  ; 0.05 seconds of noise
  (build-array (vector (seconds->samples 0.05))
               (lambda (x) (random-sample))))

; limited drum machine
; drum patterns are simply lists with either O (bass drum), X (snare) or
; #f (pause)
(define (drum n pattern tempo)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (define (make-drum drum-sample samples-per-beat)
    (array-append*
     (list drum-sample
           (make-array (vector (- samples-per-beat
                                  (array-size drum-sample)))
                       0.0))))
  (define O     (make-drum bass-drum samples-per-beat))
  (define X     (make-drum snare     samples-per-beat))
  (define pause (make-array (vector samples-per-beat) 0.0))
  (array-append*
   (for*/list ([i    (in-range n)]
               [beat (in-list pattern)])
     (case beat
       ((X)  X)
       ((O)  O)
       ((#f) pause)))))
; TODO more drums, cymbals, etc.
```

```
#lang racket
; Simple WAVE encoder

; Very helpful reference:
; http://ccrma.stanford.edu/courses/422/projects/WaveFormat/

(provide write-wav)
(require racket/sequence)

; A WAVE file has 3 parts:
;  - the RIFF header: identifies the file as WAVE
;  - data subchunk
; data : sequence of 32-bit unsigned integers
(define (write-wav data
                   #:num-channels    [num-channels    1]
                   #:sample-rate     [sample-rate     44100]
                   #:bits-per-sample [bits-per-sample 16])

  (define bytes-per-sample (quotient bits-per-sample 8))
  (define (write-integer-bytes i [size 4])
    (write-bytes (integer->integer-bytes i size #f)))
  (define data-subchunk-size
    (* (sequence-length data) num-channels (/ bits-per-sample 8)))

  ; RIFF header
  (write-bytes #"RIFF")
  ; 4 bytes: 4 + (8 + size of fmt subchunk) + (8 + size of data subchunk)
  (write-integer-bytes (+ 36 data-subchunk-size))
  (write-bytes #"WAVE")

  ; fmt subchunk
  (write-bytes #"fmt ")
  ; size of the rest of the subchunk: 16 for PCM
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)

(define (freq->sample-period freq)
  (round (/ fs freq)))

(define (seconds->samples s)
  (inexact->exact (round (* s fs))))


; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Oscillators

(provide sine-wave square-wave sawtooth-wave inverse-sawtooth-wave
         triangle-wave)

; array functions receive a vector of indices
(define-syntax-rule (array-lambda (i) body ...)
  (lambda (i*) (let ([i (vector-ref i* 0)]) body ...)))

; These all need to return floats.
; TODO use TR? would also optimize for us

(define (sine-wave freq)
  (define f (exact->inexact (/ (* freq 2.0 pi) fs)))
  (array-lambda (x) (sin (* f (exact->inexact x)))))

(define (square-wave freq)
  (define sample-period (freq->sample-period freq))
  (define sample-period/2 (quotient sample-period 2))
  (array-lambda (x)
    ; 1 for the first half of the cycle, -1 for the other half
    (define x* (modulo x sample-period))
    (if (> x* sample-period/2) -1.0 1.0)))


(define ((make-sawtooth-wave coeff) freq)
  (define sample-period (freq->sample-period freq))
  (define sample-period/2 (quotient sample-period 2))
  (array-lambda (x)
    ; gradually goes from -1 to 1 over the whole cycle
    (define x* (exact->inexact (modulo x sample-period)))
    (* coeff (- (/ x* sample-period/2) 1.0))))
(define sawtooth-wave         (make-sawtooth-wave 1.0))
(define inverse-sawtooth-wave (make-sawtooth-wave -1.0))

(define (triangle-wave freq)
  (define sample-period (freq->sample-period freq))
  (define sample-period/2 (quotient sample-period 2))
  (define sample-period/4 (quotient sample-period 4))
  (array-lambda (x)
    ; go from 1 to -1 for the first half of the cycle, then back up
    (define x* (modulo x sample-period))
    (if (> x* sample-period/2)
        (- (/ x* sample-period/4) 3.0)
        (+ (/ x* sample-period/4 -1.0) 1.0))))

; TODO make sure that all of these actually produce the right frequency
;   (i.e. no off-by-an-octave errors)

; TODO add weighted-harmonics, so we can approximate instruments
;   and take example from old synth

; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(provide emit plot-signal)

; assumes array of floats in [-1.0,1.0]
; assumes gain in [0,1], which determines how loud the output is
(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
                         (first x))
                       ss))
  (define weights (map (lambda (x) ; : Weighted-Signal
                         (real->double-flonum (second x)))
                       ss))
  (define downscale-ratio (/ 1.0 (apply + weights)))

  ; scale-signal : Float -> (Float -> Float)
  (define ((scale-signal w) x) (* x w downscale-ratio))

  (parameterize ([array-broadcasting 'permissive]) ; repeat short signals
    (for/fold ([res (array-map (scale-signal (first weights))
                               (first signals))])
        ([s (in-list (rest signals))]
         [w (in-list (rest weights))])
      (define scale (scale-signal w))
      (array-map (lambda (acc  ; : Float
                          new) ; : Float
                   (+ acc (scale new)))
                 res s))))
```

```
#lang racket
(require math/array racket/flonum racket/unsafe/ops)

(require "synth.rkt" "mixer.rkt")

(provide scale chord note sequence mix)

(define (base+relative-semitone->freq base relative-semitone)
  (* 440 (expt (expt 2 1/12) -57)))

; details at http://www.phy.mtu.edu/~suits/notefreqs.html
(define (note-freq note)
  ; A4 (440Hz) is 57 semitones above C0, which is our base.
  (* 440 (expt (expt 2 1/12) (- note 57))))

; A note is represented using the number of semitones from C0.
(define (name+octave->note name octave)
  (+ (* 12 octave)
     (case name
       [(C) 0] [(C# Db) 1] [(D) 2] [(D# Eb) 3]  [(E) 4] [(F) 5] [(F# Gb) 6]
       [(G) 7] [(G# Ab) 8] [(A) 9] [(A# Bb) 10] [(B) 11])))

; Similar to scale, but generates a chord.
; Chords are pairs (listof note) + duration
(define (chord root octave duration type . notes*)
  (define notes (apply scale root octave duration type notes*))
  (cons (map car notes) duration))

; Single note.
(define (note name octave duration)
  (cons (name+octave->note name octave) duration))

; Accepts notes or pauses, but not chords.
(define (synthesize-note note n-samples function)
  (build-array (vector n-samples)
               (if note
                   (function (note-freq note))
                   (lambda (x) 0.0))))
; pause

; repeats n times the sequence encoded by the pattern, at tempo bpm
; pattern is a list of either single notes (note . duration) or
; chords ((note ...) . duration) or pauses (#f . duration)
; TODO accept quoted notes (i.e. args to `note'). o/w entry is painful
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
                            ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```

```
#lang racket
(require math/array)
```

```
#lang racket
(require math/array)
```

```
#lang racket
(require math/array)
```

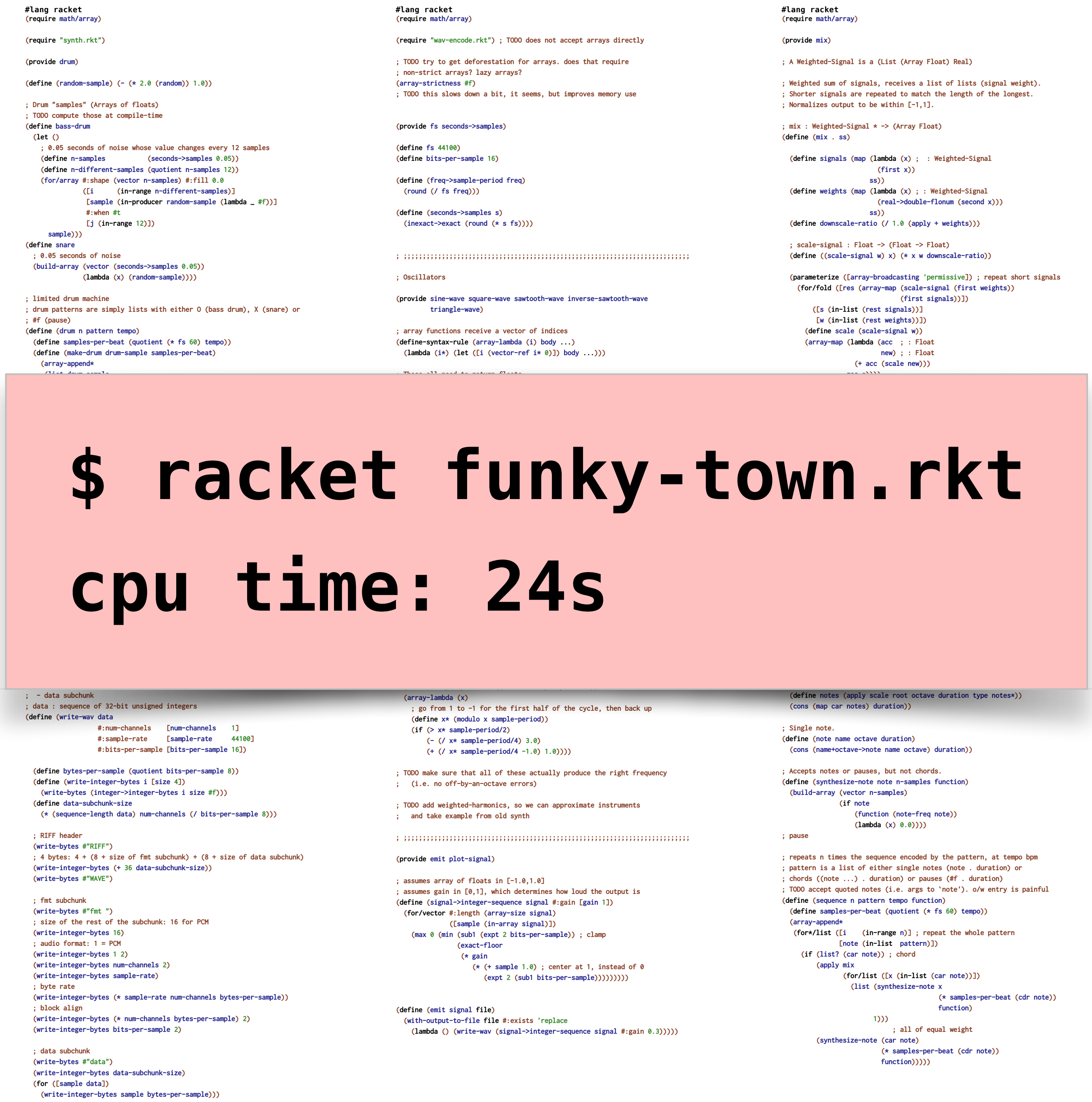
```
(emit
 (sequence
   sawtooth-wave #:bpm 380
   [(C 5) #f (C 5) #f (A# 4) #f (C 5) ...])
 "funky-town.wav")
```

```
                [beat (in-list pattern)])
      (case beat
        ((X)  X)
        ((O)  O)
        ((#f) pause)))))
; TODO more drums, cymbals, etc.
```

```
#lang racket
; Simple WAVE encoder

; Very helpful reference:
; http://ccrma.stanford.edu/courses/422/projects/WaveFormat/

(provide write-wav)
(require racket/sequence)

; A WAVE file has 3 parts:
;  - the RIFF header: identifies the file as WAVE
;  - data subchunk
; data : sequence of 32-bit unsigned integers
(define (write-wav data
                   #:num-channels    [num-channels    1]
                   #:sample-rate     [sample-rate     44100]
                   #:bits-per-sample [bits-per-sample 16])

  (define bytes-per-sample (quotient bits-per-sample 8))
  (define (write-integer-bytes i [size 4])
    (write-bytes (integer->integer-bytes i size #f)))
  (define data-subchunk-size
    (* (sequence-length data) num-channels (/ bits-per-sample 8)))

  ; RIFF header
  (write-bytes #"RIFF")
  ; 4 bytes: 4 + (8 + size of fmt subchunk) + (8 + size of data subch
  (write-integer-bytes (+ 36 data-subchunk-size))
  (write-bytes #"WAVE")

  ; fmt subchunk
  (write-bytes #"fmt ")
  ; size of the rest of the subchunk: 16 for PCM
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

```
  (define sample-period/2 (quotient sample-period 2))
  (array-lambda (x)
    ; 1 for the first half of the cycle, -1 for the other half
    (define x* (modulo x sample-period))
    (if (> x* sample-period/2) -1.0 1.0)))


(define ((make-sawtooth-wave coeff) freq)
```

```
(provide scale chord note sequence mix)

(define (base+relative-semitone->freq base relative-semitone)
  (* 440 (expt (expt 2 1/12) -57)))

; details at http://www.phy.mtu.edu/~suits/notefreqs.html
(define (note-freq note)
     (440Hz) is 57 semitones above C0, which is our base.
    40 (expt (expt 2 1/12) (- note 57))))

    te is represented using the number of semitones from C0.
    e (name+octave->note name octave)
    * 12 octave)
    case name
     [(C) 0] [(C# Db) 1] [(D) 2] [(D# Eb) 3]  [(E) 4] [(F) 5] [(F# Gb) 6]
     [(G) 7] [(G# Ab) 8] [(A) 9] [(A# Bb) 10] [(B) 11])))

    lar to scale, but generates a chord.
    ds are pairs (listof note) + duration
    e (chord root octave duration type . notes*)
    ine notes (apply scale root octave duration type notes*))
    s (map car notes) duration))

    le note.
    e (note name octave duration)
    s (name+octave->note name octave) duration))

    pts notes or pauses, but not chords.
    e (synthesize-note note n-samples function)
    ld-array (vector n-samples)
             (if note
                 (function (note-freq note))
                 (lambda (x) 0.0))))
    e

    ats n times the sequence encoded by the pattern, at tempo bpm
    ern is a list of either single notes (note . duration) or
    ds ((note ...) . duration) or pauses (#f . duration)
     accept quoted notes (i.e. args to `note'). o/w entry is painful
    e (sequence n pattern tempo function)
    ine samples-per-beat (quotient (* fs 60) tempo))
    ay-append*
    r*/list ([i    (in-range n)] ; repeat the whole pattern
             [note (in-list  pattern)])
    if (list? (car note)) ; chord
       (apply mix
              (for/list ([x (in-list (car note))])
                (list (synthesize-note x
                                       (* samples-per-beat (cdr note))
                                       function)
                      1)))
                        ; all of equal weight
       (synthesize-note (car note)
                        (* samples-per-beat (cdr note))
                        function)))))
```



```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samples (quotient n-samples 12))
    (for/array #:shape (vector n-samples) #:fill 0.0
               ([i      (in-range n-different-samples)]
                [sample (in-producer random-sample (lambda _ #f))]
                #:when #t
                [j (in-range 12)])
      sample)))
(define snare
  ; 0.05 seconds of noise
  (build-array (vector (seconds->samples 0.05))
               (lambda (x) (random-sample))))

; limited drum machine
; drum patterns are simply lists with either O (bass drum), X (snare) or
; #f (pause)
(define (drum n pattern tempo)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (define (make-drum drum-sample samples-per-beat)
    (array-append*
     (list drum-sample
```

```
;  - data subchunk
; data : sequence of 32-bit unsigned integers
(define (write-wav data
                   #:num-channels    [num-channels    1]
                   #:sample-rate     [sample-rate     44100]
                   #:bits-per-sample [bits-per-sample 16])

  (define bytes-per-sample (quotient bits-per-sample 8))
  (define (write-integer-bytes i [size 4])
    (write-bytes (integer->integer-bytes i size #f)))
  (define data-subchunk-size
    (* (sequence-length data) num-channels (/ bits-per-sample 8)))

  ; RIFF header
  (write-bytes #"RIFF")
  ; 4 bytes: 4 + (8 + size of fmt subchunk) + (8 + size of data subchunk)
  (write-integer-bytes (+ 36 data-subchunk-size))
  (write-bytes #"WAVE")

  ; fmt subchunk
  (write-bytes #"fmt ")
  ; size of the rest of the subchunk: 16 for PCM
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

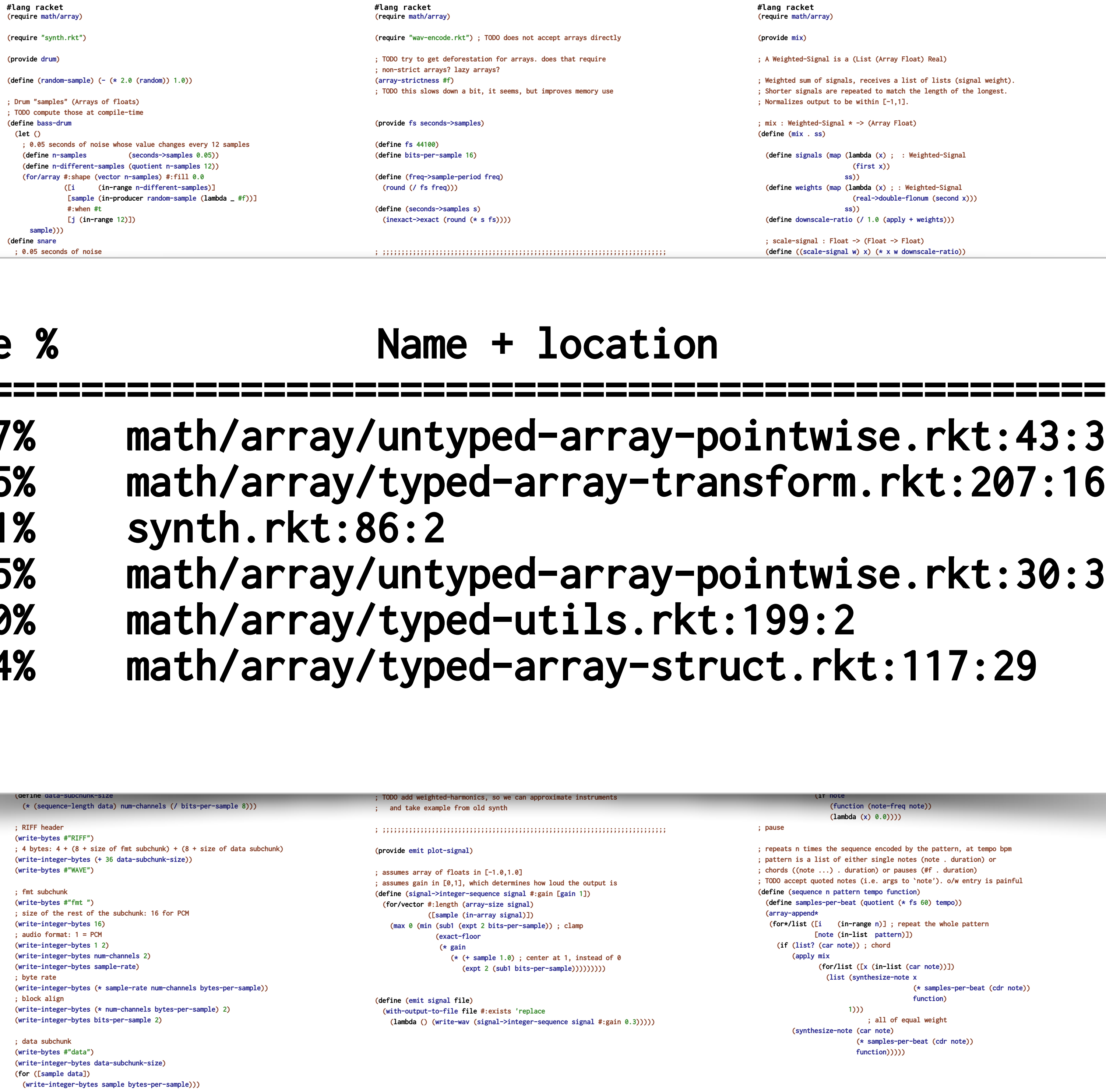
```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)

(define (freq->sample-period freq)
  (round (/ fs freq)))

(define (seconds->samples s)
  (inexact->exact (round (* s fs))))


; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Oscillators

(provide sine-wave square-wave sawtooth-wave inverse-sawtooth-wave
         triangle-wave)

; array functions receive a vector of indices
(define-syntax-rule (array-lambda (i) body ...)
  (lambda (i*) (let ([i (vector-ref i* 0)]) body ...)))

; These all need to return floats
```

```
  (array-lambda (x)
    ; go from 1 to -1 for the first half of the cycle, then back up
    (define x* (modulo x sample-period))
    (if (> x* sample-period/2)
        (- (/ x* sample-period/4) 3.0)
        (+ (/ x* sample-period/4 -1.0) 1.0))))

; TODO make sure that all of these actually produce the right frequency
;   (i.e. no off-by-an-octave errors)

; TODO add weighted-harmonics, so we can approximate instruments
;   and take example from old synth

; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(provide emit plot-signal)

; assumes array of floats in [-1.0,1.0]
; assumes gain in [0,1], which determines how loud the output is
(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
                         (first x))
                       ss))
  (define weights (map (lambda (x) ; : Weighted-Signal
                         (real->double-flonum (second x)))
                       ss))
  (define downscale-ratio (/ 1.0 (apply + weights)))

  ; scale-signal : Float -> (Float -> Float)
  (define ((scale-signal w) x) (* x w downscale-ratio))

  (parameterize ([array-broadcasting 'permissive]) ; repeat short signals
    (for/fold ([res (array-map (scale-signal (first weights))
                               (first signals))])
        ([s (in-list (rest signals))]
         [w (in-list (rest weights))])
      (define scale (scale-signal w))
      (array-map (lambda (acc  ; : Float
                          new) ; : Float
                   (+ acc (scale new)))
                 res s))))
```

```
  (define notes (apply scale root octave duration type notes*))
  (cons (map car notes) duration))

; Single note.
(define (note name octave duration)
  (cons (name+octave->note name octave) duration))

; Accepts notes or pauses, but not chords.
(define (synthesize-note note n-samples function)
  (build-array (vector n-samples)
               (if note
                   (function (note-freq note))
                   (lambda (x) 0.0)))) ; pause

; repeats n times the sequence encoded by the pattern, at tempo bpm
; pattern is a list of either single notes (note . duration) or
; chords ((note ...) . duration) or pauses (#f . duration)
; TODO accept quoted notes (i.e. args to `note'). o/w entry is painful
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
                            ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```

```
$ racket funky-town.rkt
cpu time: 24s
```



```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samples (quotient n-samples 12))
    (for/array #:shape (vector n-samples) #:fill 0.0
               ([i      (in-range n-different-samples)]
                [sample (in-producer random-sample (lambda _ #f))]
                #:when #t
                [j (in-range 12)])
      sample)))
(define snare
  ; 0.05 seconds of noise
```

```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)

(define (freq->sample-period freq)
  (round (/ fs freq)))

(define (seconds->samples s)
  (inexact->exact (round (* s fs))))


; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

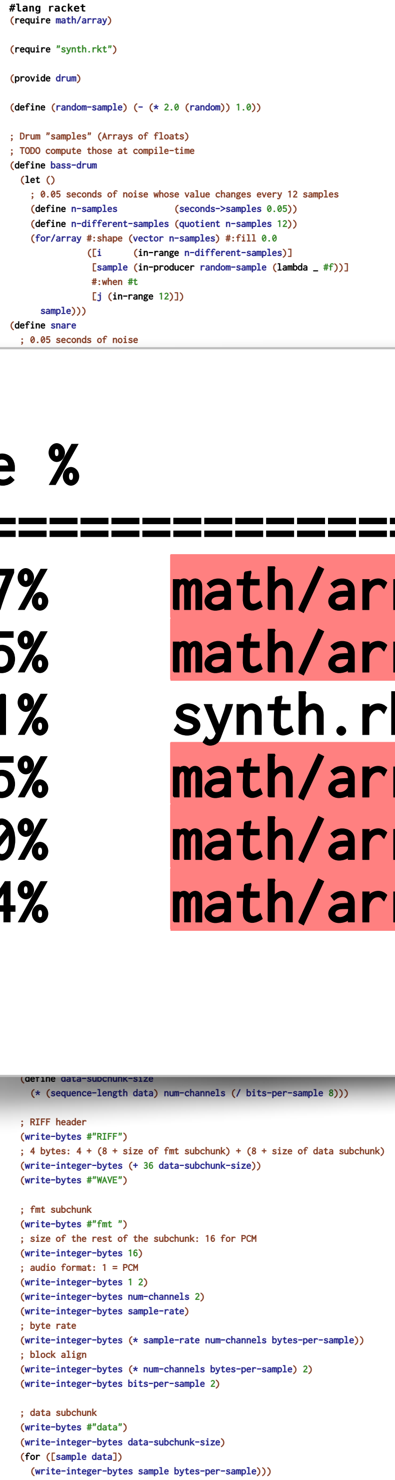
```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
                         (first x))
                       ss))
  (define weights (map (lambda (x) ; : Weighted-Signal
                         (real->double-flonum (second x)))
                       ss))
  (define downscale-ratio (/ 1.0 (apply + weights)))

  ; scale-signal : Float -> (Float -> Float)
  (define ((scale-signal w) x) (* x w downscale-ratio))
```

```
Time %                    Name + location
==========================================================
32.7%    math/array/untyped-array-pointwise.rkt:43:39
27.5%    math/array/typed-array-transform.rkt:207:16
18.1%    synth.rkt:86:2
 6.5%    math/array/untyped-array-pointwise.rkt:30:35
 6.0%    math/array/typed-utils.rkt:199:2
 4.4%    math/array/typed-array-struct.rkt:117:29
...
```

```
(define data-subchunk-size
  (* (sequence-length data) num-channels (/ bits-per-sample 8)))

; RIFF header
(write-bytes #"RIFF")
; 4 bytes: 4 + (8 + size of fmt subchunk) + (8 + size of data subchunk)
(write-integer-bytes (+ 36 data-subchunk-size))
(write-bytes #"WAVE")

; fmt subchunk
(write-bytes #"fmt ")
; size of the rest of the subchunk: 16 for PCM
(write-integer-bytes 16)
; audio format: 1 = PCM
(write-integer-bytes 1 2)
(write-integer-bytes num-channels 2)
(write-integer-bytes sample-rate)
; byte rate
(write-integer-bytes (* sample-rate num-channels bytes-per-sample))
; block align
(write-integer-bytes (* num-channels bytes-per-sample) 2)
(write-integer-bytes bits-per-sample 2)

; data subchunk
(write-bytes #"data")
(write-integer-bytes data-subchunk-size)
(for ([sample data])
  (write-integer-bytes sample bytes-per-sample)))
```

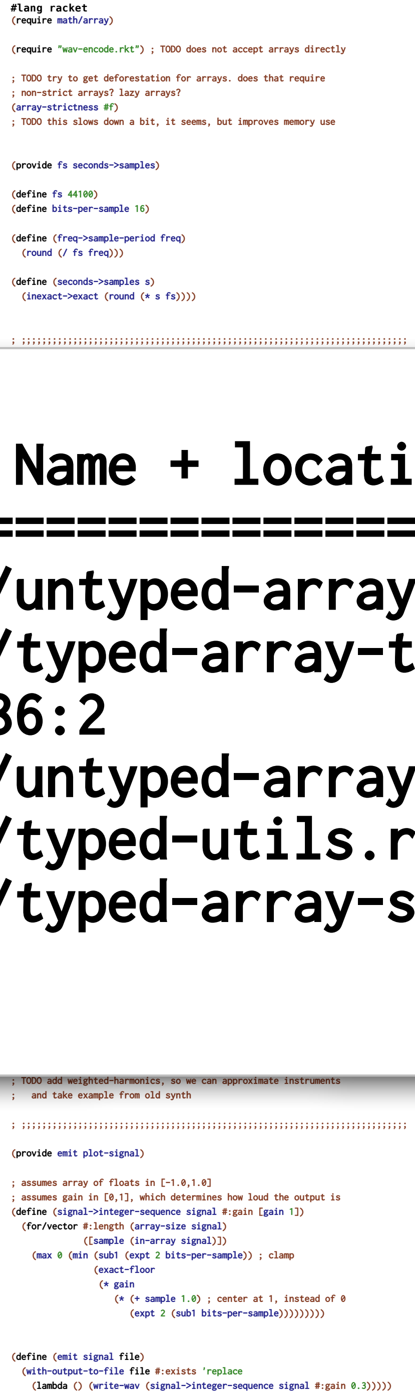
```
; TODO add weighted-harmonics, so we can approximate instruments
;   and take example from old synth

; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(provide emit plot-signal)

; assumes array of floats in [-1.0,1.0]
; assumes gain in [0,1], which determines how loud the output is
(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
            (if note
                (function (note-freq note))
                (lambda (x) 0.0))))
; pause

; repeats n times the sequence encoded by the pattern, at tempo bpm
; pattern is a list of either single notes (note . duration) or
; chords ((note ...) . duration) or pauses (#f . duration)
; TODO accept quoted notes (i.e. args to `note'). o/w entry is painful
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
                   ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```



```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samples (quotient n-samples 12))
    (for/array #:shape (vector n-samples) #:fill 0.0
               ([i      (in-range n-different-samples)]
                [sample (in-producer random-sample (lambda _ #f))]
                #:when #t
                [j (in-range 12)])
      sample)))
(define snare
  ; 0.05 seconds of noise
```

```
(define data-subchunk-size
  (* (sequence-length data) num-channels (/ bits-per-sample 8)))

; RIFF header
(write-bytes #"RIFF")
; 4 bytes: 4 + (8 + size of fmt subchunk) + (8 + size of data subchunk)
(write-integer-bytes (+ 36 data-subchunk-size))
(write-bytes #"WAVE")

; fmt subchunk
(write-bytes #"fmt ")
; size of the rest of the subchunk: 16 for PCM
(write-integer-bytes 16)
; audio format: 1 = PCM
(write-integer-bytes 1 2)
(write-integer-bytes num-channels 2)
(write-integer-bytes sample-rate)
; byte rate
(write-integer-bytes (* sample-rate num-channels bytes-per-sample))
; block align
(write-integer-bytes (* num-channels bytes-per-sample) 2)
(write-integer-bytes bits-per-sample 2)

; data subchunk
(write-bytes #"data")
(write-integer-bytes data-subchunk-size)
(for ([sample data])
  (write-integer-bytes sample bytes-per-sample)))
```



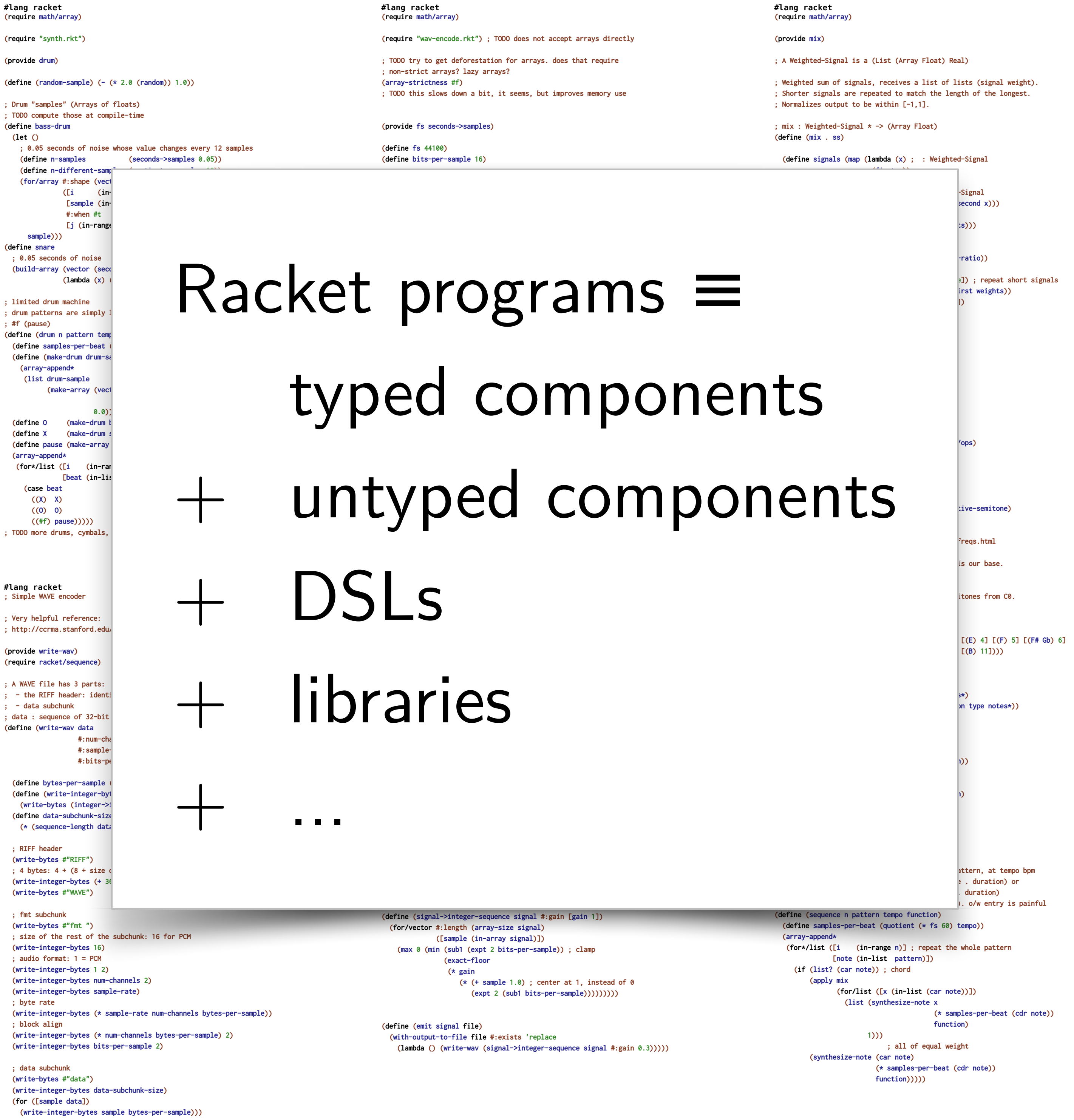
```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)

(define (freq->sample-period freq)
  (round (/ fs freq)))

(define (seconds->samples s)
  (inexact->exact (round (* s fs))))


; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; TODO add weighted-harmonics, so we can approximate instruments
;   and take example from old synth

; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(provide emit plot-signal)

; assumes array of floats in [-1.0,1.0]
; assumes gain in [0,1], which determines how loud the output is
(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
                         (first x))
                       ss))
  (define weights (map (lambda (x) ; : Weighted-Signal
                         (real->double-flonum (second x)))
                       ss))
  (define downscale-ratio (/ 1.0 (apply + weights)))

  ; scale-signal : Float -> (Float -> Float)
  (define ((scale-signal w) x) (* x w downscale-ratio))
```

```
                (if note
                    (function (note-freq note))
                    (lambda (x) 0.0))))
; pause

; repeats n times the sequence encoded by the pattern, at tempo bpm
; pattern is a list of either single notes (note . duration) or
; chords ((note ...) . duration) or pauses (#f . duration)
; TODO accept quoted notes (i.e. args to `note'). o/w entry is painful
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
                     ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```

```
Time %                  Name + location
==================================================
32.7%    math/array/untyped-array-pointwise.rkt:43:39
27.5%    math/array/typed-array-transform.rkt:207:16
18.1%    synth.rkt:86:2
 6.5%    math/array/untyped-array-pointwise.rkt:30:35
 6.0%    math/array/typed-utils.rkt:199:2
 4.4%    math/array/typed-array-struct.rkt:117:29
...
```



Racket programs ≡
typed components
+ untyped components
+ DSLs
+ libraries
+ ...

```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samp
    (for/array #:shape (vect
               ([i      (in-
                [sample (in-
                #:when #t
                [j (in-range
      sample)))
(define snare
  ; 0.05 seconds of noise
  (build-array (vector (secc
               (lambda (x) (

; limited drum machine
; drum patterns are simply l
; #f (pause)
(define (drum n pattern temp
  (define samples-per-beat (
  (define (make-drum drum-sa
    (array-append*
     (list drum-sample
           (make-array (vect

                      0.0))
  (define O     (make-drum b
  (define X     (make-drum s
  (define pause (make-array
  (array-append*
   (for*/list ([i    (in-ran
               [beat (in-lis
     (case beat
       ((X)  X)
       ((O)  O)
       ((#f) pause)))))
; TODO more drums, cymbals,
```

```
#lang racket
; Simple WAVE encoder

; Very helpful reference:
; http://ccrma.stanford.edu/

(provide write-wav)
(require racket/sequence)

; A WAVE file has 3 parts:
;  - the RIFF header: identi
;  - data subchunk
; data : sequence of 32-bit
(define (write-wav data
                   #:num-cha
                   #:sample-
                   #:bits-pe

  (define bytes-per-sample (
  (define (write-integer-byt
    (write-bytes (integer->i
  (define data-subchunk-size
    (* (sequence-length data

  ; RIFF header
  (write-bytes #"RIFF")
  ; 4 bytes: 4 + (8 + size c
  (write-integer-bytes (+ 36
  (write-bytes #"WAVE")

  ; fmt subchunk
  (write-bytes #"fmt ")
  ; size of the rest of the subchunk: 16 for PCM
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

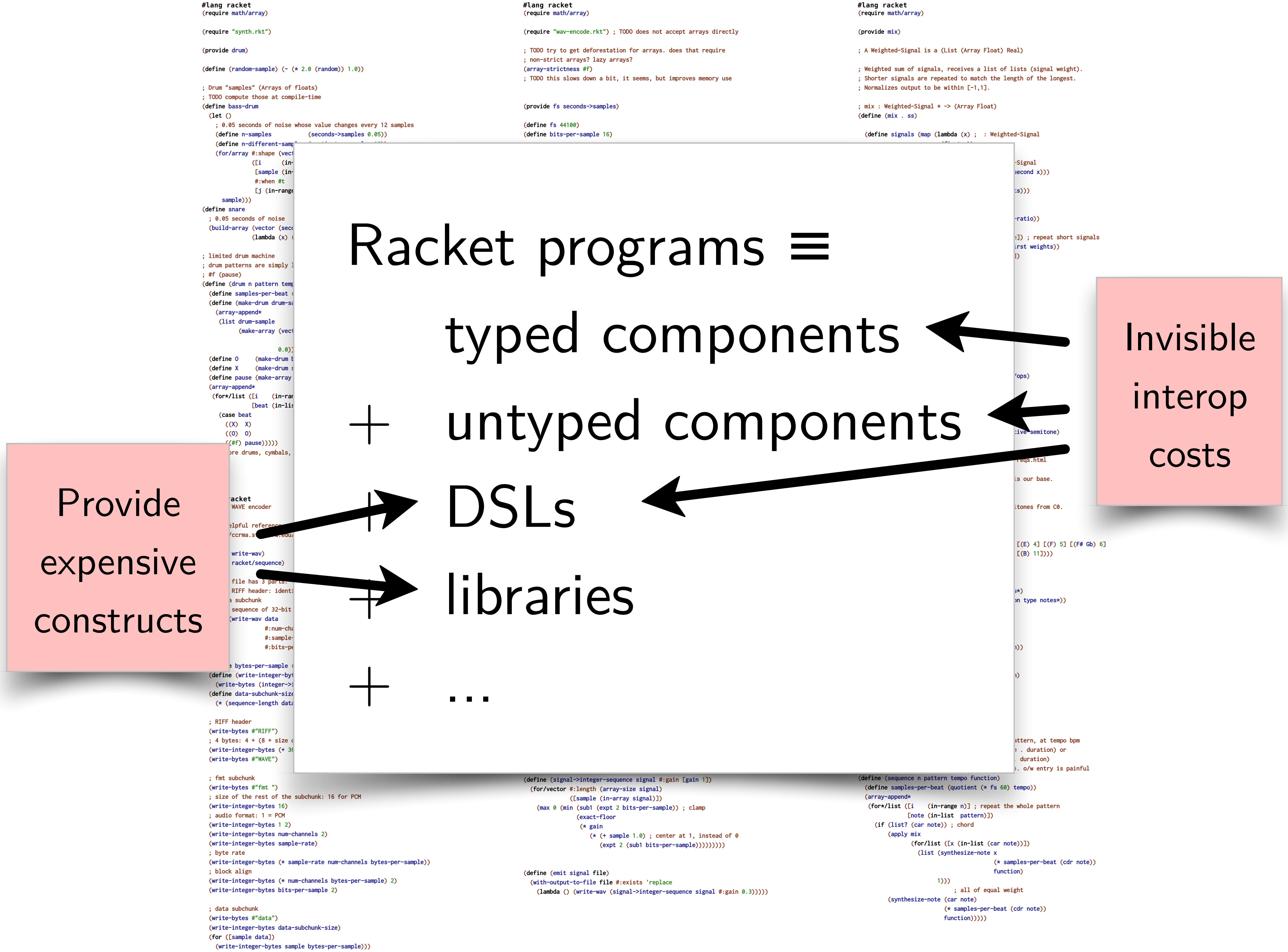
```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)
```

```
(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
```

```
-Signal
second x)))

s)))

-ratio))

]) ; repeat short signals
rst weights))
])

'ops)

ive-semitone)

freqs.html

is our base.

tones from C0.

 [(E) 4] [(F) 5] [(F# Gb) 6]
 [(B) 11])))

s*)
n type notes*))

))

)

attern, at tempo bpm
 . duration) or
 duration)
. o/w entry is painful
```

```
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
                   ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```



Racket programs ≡

typed components

+ untyped components

+ DSLs

+ libraries

+ ...

Provide expensive constructs

Invisible interop costs

```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of noise whose value changes every 12 samples
    (define n-samples           (seconds->samples 0.05))
    (define n-different-samp
    (for/array #:shape (vect
               ([i      (in-
                [sample (in-
                #:when #t
                [j (in-range
      sample)))
(define snare
  ; 0.05 seconds of noise
  (build-array (vector (seco
               (lambda (x) (

; limited drum machine
; drum patterns are simply l
; #f (pause)
(define (drum n pattern temp
  (define samples-per-beat (
  (define (make-drum drum-sa
    (array-append*
     (list drum-sample
           (make-array (vect

                     0.0)
  (define O     (make-drum b
  (define X     (make-drum s
  (define pause (make-array
  (array-append*
   (for*/list ([i    (in-ran
               [beat (in-lis
     (case beat
       ((X)  X)
       ((O)  O)
       ((#f) pause)))))
ore drums, cymbals,

acket
WAVE encoder

elpful referenc
/ccrma.st      .edu/

write-wav)
racket/sequence)

file has 3 parts:
RIFF header: identi
a subchunk
sequence of 32-bit
(write-wav data
           #:num-cha
           #:sample-
           #:bits-pe

e bytes-per-sample (
  (define (write-integer-byt
    (write-bytes (integer->i
  (define data-subchunk-size
    (* (sequence-length data

  ; RIFF header
  (write-bytes #"RIFF")
  ; 4 bytes: 4 + (8 + size o
  (write-integer-bytes (+ 36
  (write-bytes #"WAVE")

  ; fmt subchunk
  (write-bytes #"fmt ")
  ; size of the rest of the subchunk: 16 for PCM
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

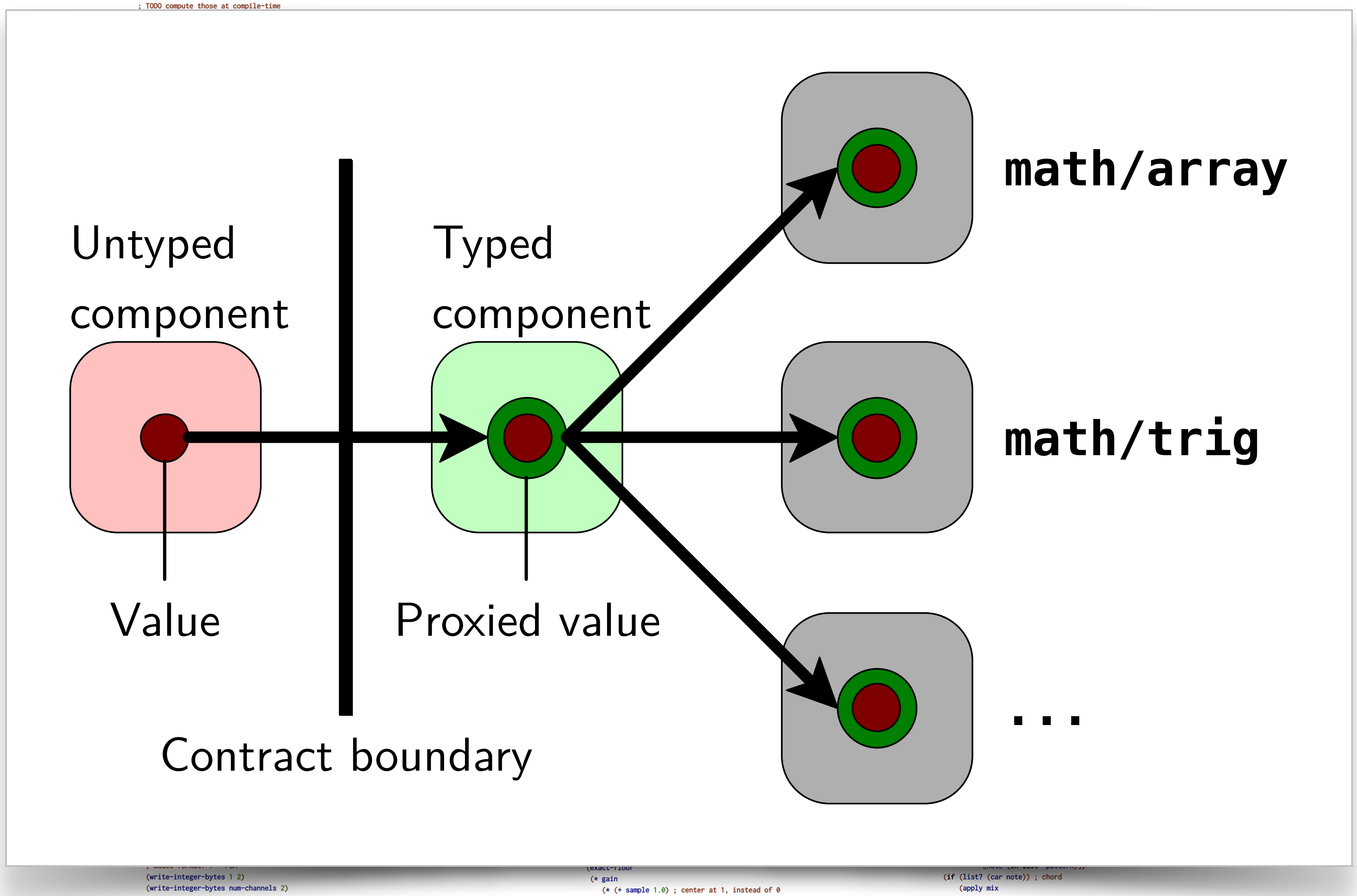
```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)

(define fs 44100)
(define bits-per-sample 16)

(define (signal->integer-sequence signal #:gain [gain 1])
  (for/vector #:length (array-size signal)
              ([sample (in-array signal)])
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)

  (define signals (map (lambda (x) ;  : Weighted-Signal
-Signal
second x)))

s)))

-ratio))

]) ; repeat short signals
rst weights))
])

ops)

ive semitone)

eqs.html

s our base.

tones from C0.

[(E) 4] [(F) 5] [(F# Gb) 6]
[(B) 11])))

s*)
n type notes*))

))

)

ttern, at tempo bpm
. duration) or
duration)
. o/w entry is painful
(define (sequence n pattern tempo function)
  (define samples-per-beat (quotient (* fs 60) tempo))
  (array-append*
   (for*/list ([i    (in-range n)] ; repeat the whole pattern
               [note (in-list  pattern)])
     (if (list? (car note)) ; chord
         (apply mix
                (for/list ([x (in-list (car note))])
                  (list (synthesize-note x
                                         (* samples-per-beat (cdr note))
                                         function)
                        1)))
         ; all of equal weight
         (synthesize-note (car note)
                          (* samples-per-beat (cdr note))
                          function)))))
```

```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
```

```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].
```



Untyped component

Typed component

Value

Proxied value

Contract boundary

```
; audio format: 1 = PCM
(write-integer-bytes 1 2)
(write-integer-bytes num-channels 2)
(write-integer-bytes sample-rate)
; byte rate
(write-integer-bytes (* sample-rate num-channels bytes-per-sample))
; block align
(write-integer-bytes (* num-channels bytes-per-sample) 2)
(write-integer-bytes bits-per-sample 2)

; data subchunk
(write-bytes #"data")
(write-integer-bytes data-subchunk-size)
(for ([sample data])
  (write-integer-bytes sample bytes-per-sample)))
```

```
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
          [note (in-list pattern)])
(if (list? (car note)) ; chord
    (apply mix
           (for/list ([x (in-list (car note))])
             (list (synthesize-note x
                                    (* samples-per-beat (cdr note))
                                    function)
                   1)))
                        ; all of equal weight
    (synthesize-note (car note)
                     (* samples-per-beat (cdr note))
                     function)))))
```

```
#lang racket
(require math/array)

(require "synth.rkt")

(provide drum)

(define (random-sample) (- (* 2.0 (random)) 1.0))

; Drum "samples" (Arrays of floats)
; TODO compute those at compile-time
(define bass-drum
  (let ()
    ; 0.05 seconds of n
    (define n-samples
    (define n-different
    (for/array #:shape
               ([i
                [sample
                #:when
                [j (in-
     sample)))
(define snare
  ; 0.05 seconds of noi
  (build-array (vector
               (lambda

; limited drum machine
; drum patterns are sim
; #f (pause)
(define (drum n pattern
  (define samples-per-b
  (define (make-drum dr
    (array-append*
     (list drum-sample
           (make-array
                0.0))))
  (define O     (make-drum bass-drum samples-per-beat))
  (define X     (make-drum snare     samples-per-beat))
  (define pause (make-array (vector samples-per-beat) 0.0))
  (array-append*
   (for*/list ([i    (in-range n)]
               [beat (in-list pattern)])
     (case beat
       ((X)  X)
       ((O)  O)
       ((#f)
; TODO more
```

```
#lang racket
(require math/array)

(require "wav-encode.rkt") ; TODO does not accept arrays directly

; TODO try to get deforestation for arrays. does that require
; non-strict arrays? lazy arrays?
(array-strictness #f)
; TODO this slows down a bit, it seems, but improves memory use


(provide fs seconds->samples)
```

```
(define (sine-wave freq)
  (define f (exact->inexact (/ (* freq 2.0 pi) fs)))
  (array-lambda (x) (sin (* f (exact->inexact x)))))

(define (square-wave freq)
  (define sample-period (freq->sample-period freq))
  (define sample-period/2 (quotient sample-period 2))
  (array-lambda (x)
    ; 1 for the first half of the cycle, -1 for the other half
    (define x* (modulo x sample-period))
```

```
#lang racket
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)

; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].

; mix : Weighted-Signal * -> (Array Float)
```

```
nal

al
d x)))

o))

 repeat short signals
weights))
```

```
#lang racket
(require math/array racket/flonum racket/unsafe/ops)

(require "synth.rkt" "mixer.rkt")

(provide scale chord note sequence mix)

(define (base+relative-semitone->freq base relative-semitone)
```

```
[(F# Gb) 6]

bpm

painful
```

```
#lang rack
; Simple WAV

; Very helpf
; http://ccr

(provide wri
(require rac

; A WAVE fil
;  - the RIF
;  - data su
; data : seq
(define (wri



  (define by
  (define (w
    (write-b
  (define da
    (* (sequ

  ; RIFF hea
  (write-byt
  ; 4 bytes:
  (write-int
  (write-byt

  ; fmt subc
  (write-byt
  ; size of
  (write-integer-bytes 16)
  ; audio format: 1 = PCM
  (write-integer-bytes 1 2)
  (write-integer-bytes num-channels 2)
  (write-integer-bytes sample-rate)
  ; byte rate
  (write-integer-bytes (* sample-rate num-channels bytes-per-sample))
  ; block align
  (write-integer-bytes (* num-channels bytes-per-sample) 2)
  (write-integer-bytes bits-per-sample 2)

  ; data subchunk
  (write-bytes #"data")
  (write-integer-bytes data-subchunk-size)
  (for ([sample data])
    (write-integer-bytes sample bytes-per-sample)))
```

```
    (max 0 (min (sub1 (expt 2 bits-per-sample)) ; clamp
                (exact-floor
                 (* gain
                    (* (+ sample 1.0) ; center at 1, instead of 0
                       (expt 2 (sub1 bits-per-sample)))))))))


(define (emit signal file)
  (with-output-to-file file #:exists 'replace
    (lambda () (write-wav (signal->integer-sequence signal #:gain 0.3)))))
```

```
(for*/list ([i    (in-range n)] ; repeat the whole pattern
            [note (in-list  pattern)])
  (if (list? (car note)) ; chord
      (apply mix
             (for/list ([x (in-list (car note))])
               (list (synthesize-note x
                                      (* samples-per-beat (cdr note))
                                      function)
                     1)))
                        ; all of equal weight
      (synthesize-note (car note)
                       (* samples-per-beat (cdr note))
                       function)))))
```

Hard to diagnose

Build a tool!

Today's menu

The user's view

- How to use the tool

The library author's view

- How to extend the tool

The tool builder's view

- How to build a similar tool

Evaluation

- How well does the tool work

The User's View

How to use the tool

```
$ racket funky-town-profile.rkt

Contracts account for 73.77% of running time
     (17568 / 23816 ms)

  6210 ms : Array-unsafe-proc
            (-> Array (-> (vectorof Int) any))
  3110 ms : array-append*
            (->* ((listof Array)) (Int) Array)
  2776 ms : unsafe-build-array
            (-> (vectorof Int) [...] Array)
  ...

Generic sequences account for 0.04% of running time
     (10 / 23816 ms)

  10 ms : wav-encode.rkt:51:16
```

```
$ racket funky-town-profile.rkt

Contracts account for 73.77% of running time
     (17568 / 23816 ms)

  6210 ms : Array-unsafe-proc
            (-> Array (-> (vectorof Int) any))
  3110 ms : array-append*
            (->* ((listof Array)) (Int) Array)
  2776 ms : unsafe-build-array
            (-> (vectorof Int) [...] Array)
  ...

Generic sequences account for 0.04% of running time
     (10 / 23816 ms)

  10 ms : wav-encode.rkt:51:16
```

```
$ racket funky-town-profile.rkt

Contracts account for 73.77% of running time
```

Report costs per feature / instance

```
Generic sequences account for 0.04% of running time
     (10 / 23816 ms)

  10 ms : wav-encode.rkt:51:16
```

Reporting costs per feature instance

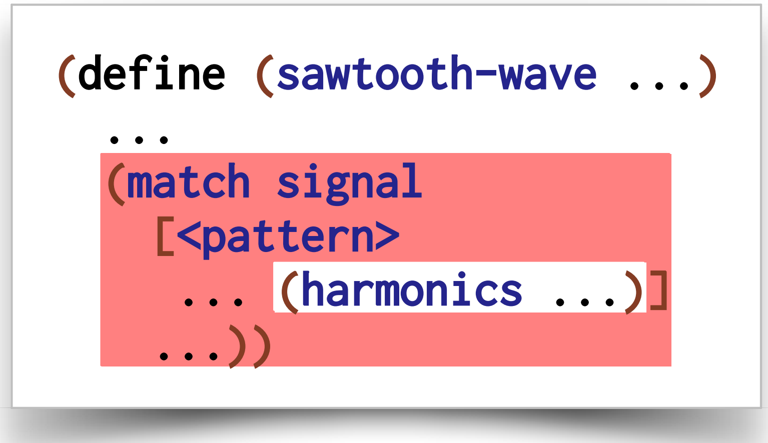
```
<linguistic feature> : <total cost>
   <cost> : <instance>
   <cost> : <instance>
   ...
```

E.g.

Output
Casts
Marketplace processes
Pattern matching
Keyword arguments
Generic sequences
Security checks
Contracts
Method dispatch
Backtracking

Reporting costs per feature instance

```
Pattern Matching : 1000ms
    600ms : sequencer.rkt:23
    200ms : drum.rkt:52
    ...
```

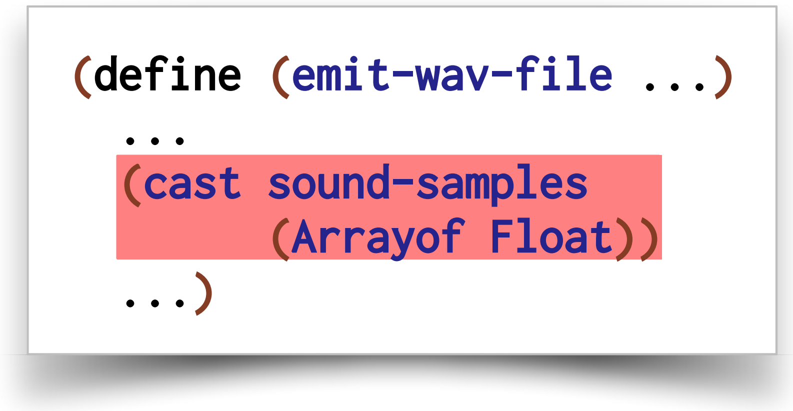


```
(define (sawtooth-wave ...)
  ...
  (match signal
    [<pattern>
     ... (harmonics ...)]
    ...))
```

Instance $\sim$ Source location

Reporting costs per feature instance

```
Checked Casts : 400ms
    200ms : drum.rkt:17
    100ms : mixer.rkt:34
    ...
```

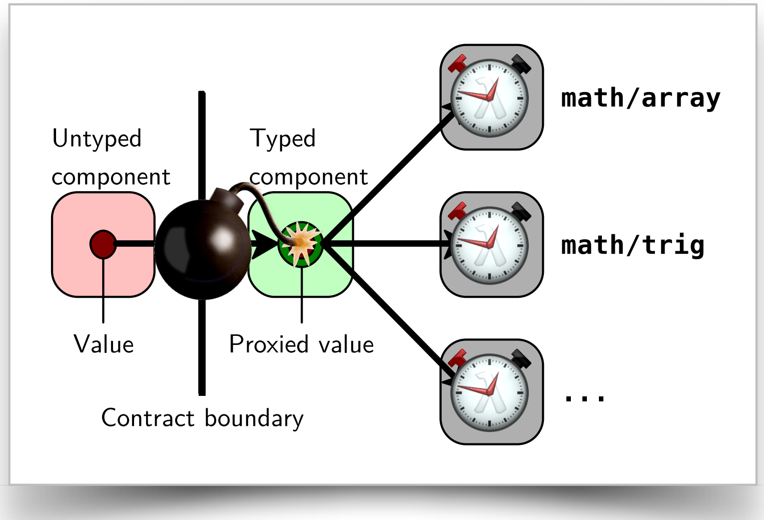


```
(define (emit-wav-file ...)
  ...
  (cast sound-samples
        (Arrayof Float))
  ...)
```

Instance ∼ Source location

Reporting costs per feature instance

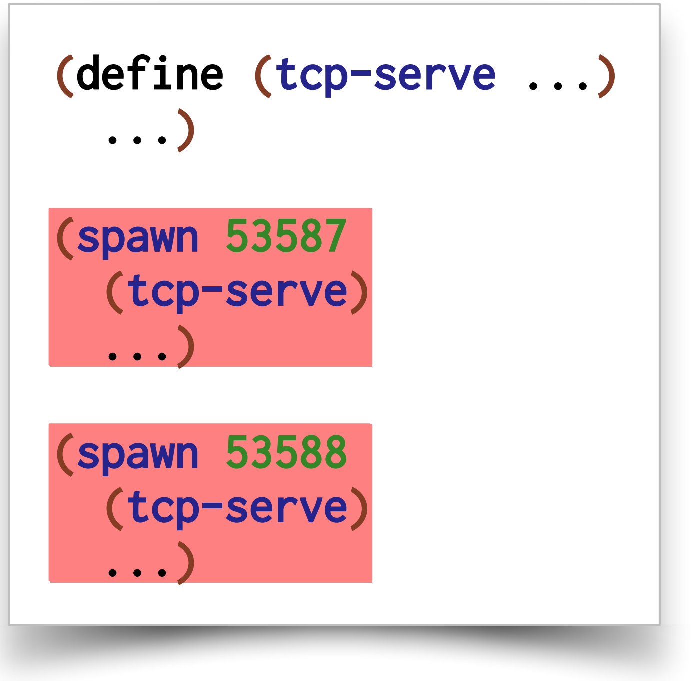
```
Contracts : 2400ms
    1300ms : make-waveform
    500ms  : generate-chord
    ...
```



1 instance: Costs in N locations

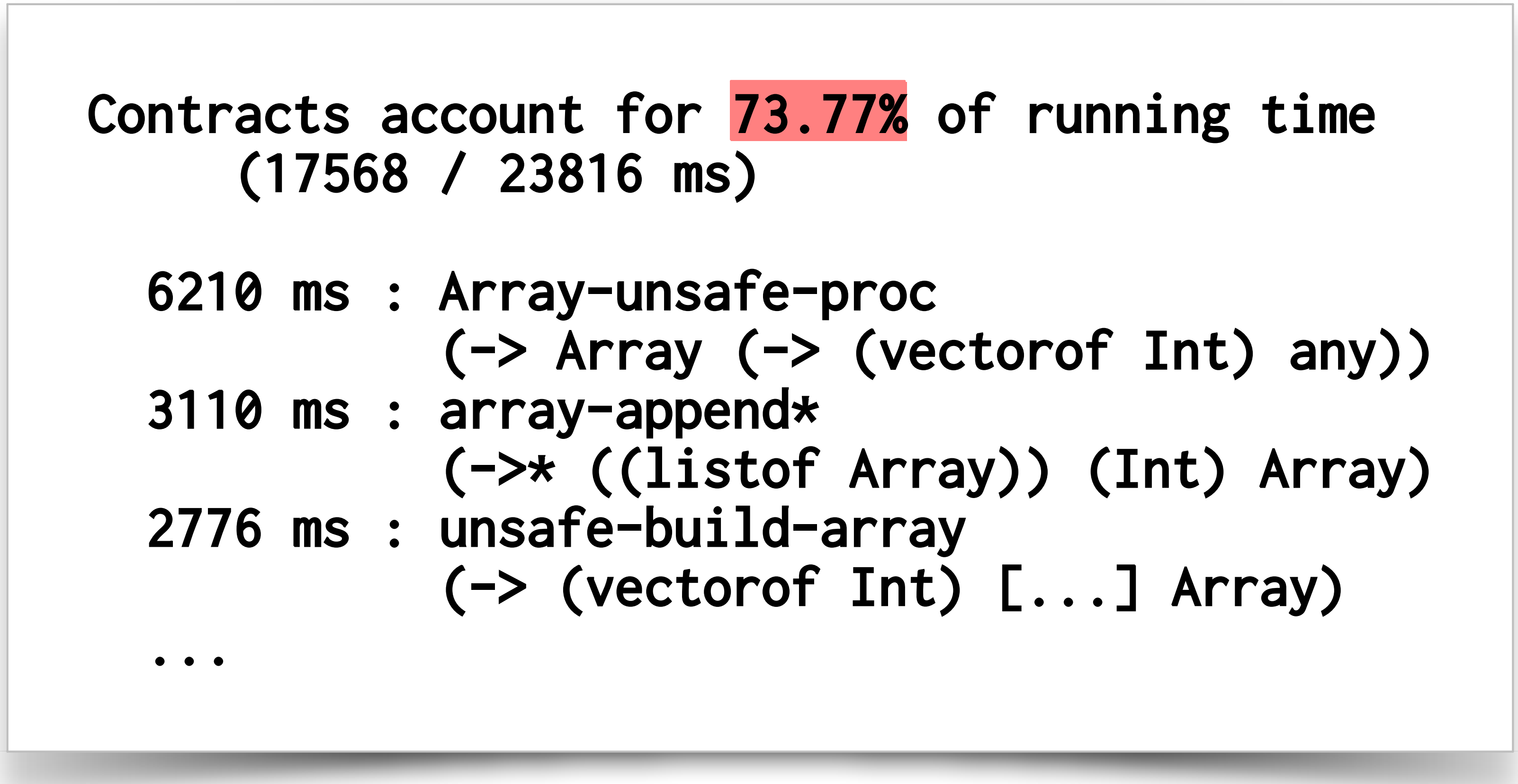
Reporting costs per feature instance

```
Marketplace Processes : 1300ms
    800ms : (tcp-serve 53588)
    400ms : (tcp-serve 53587)
    ...
```



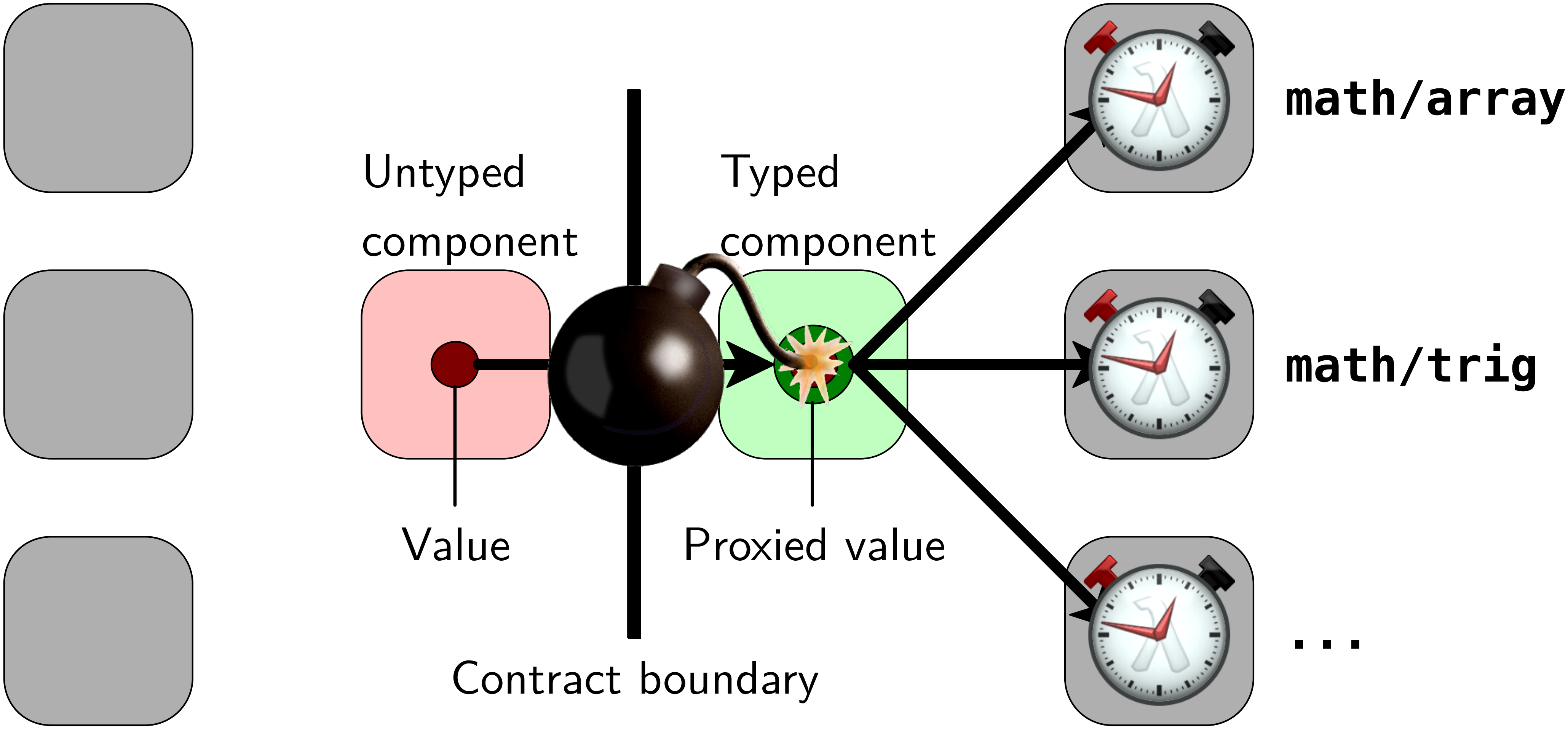
```
(define (tcp-serve ...)
  ...)

(spawn 53587
  (tcp-serve)
  ...)

(spawn 53588
  (tcp-serve)
  ...)
```

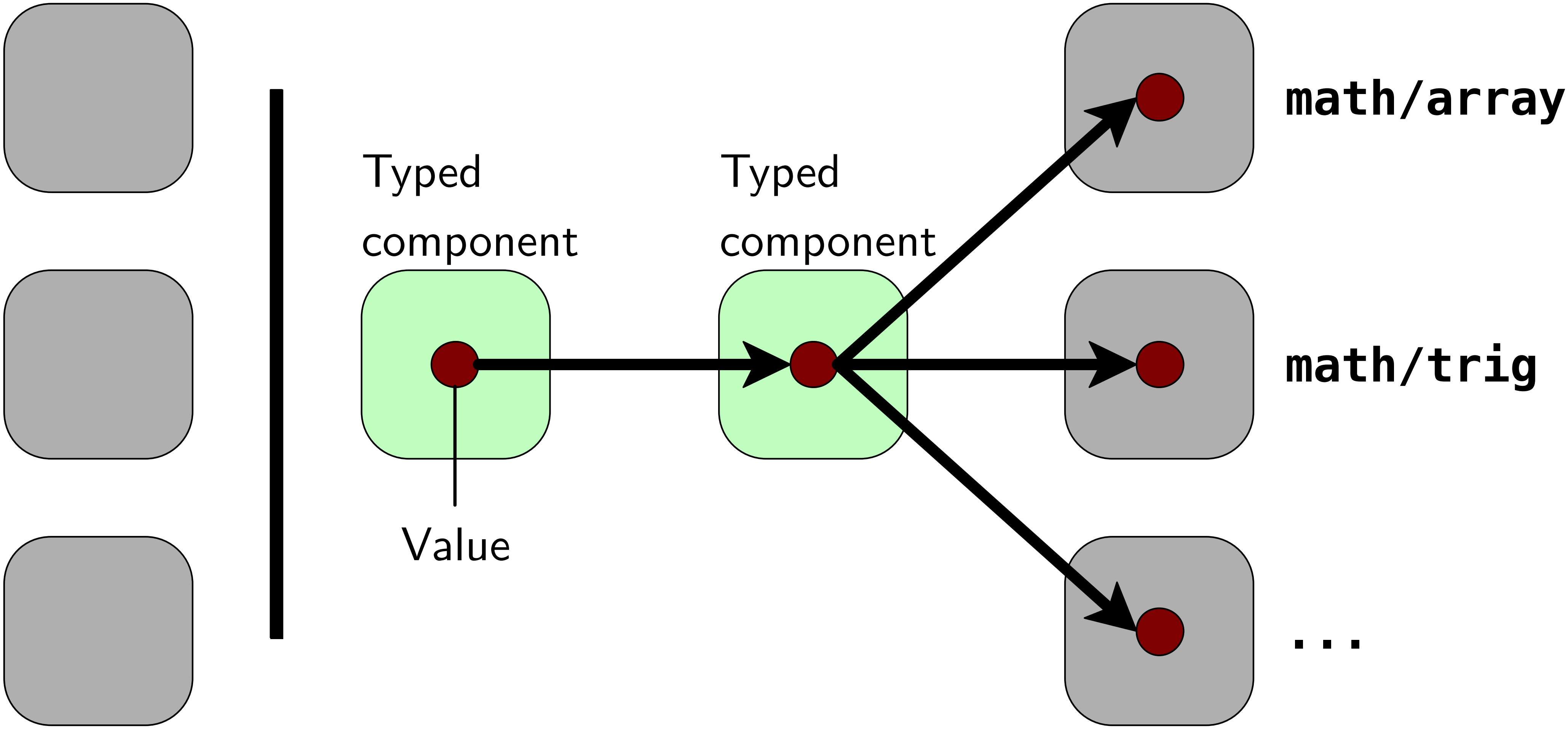
1 location: N instances

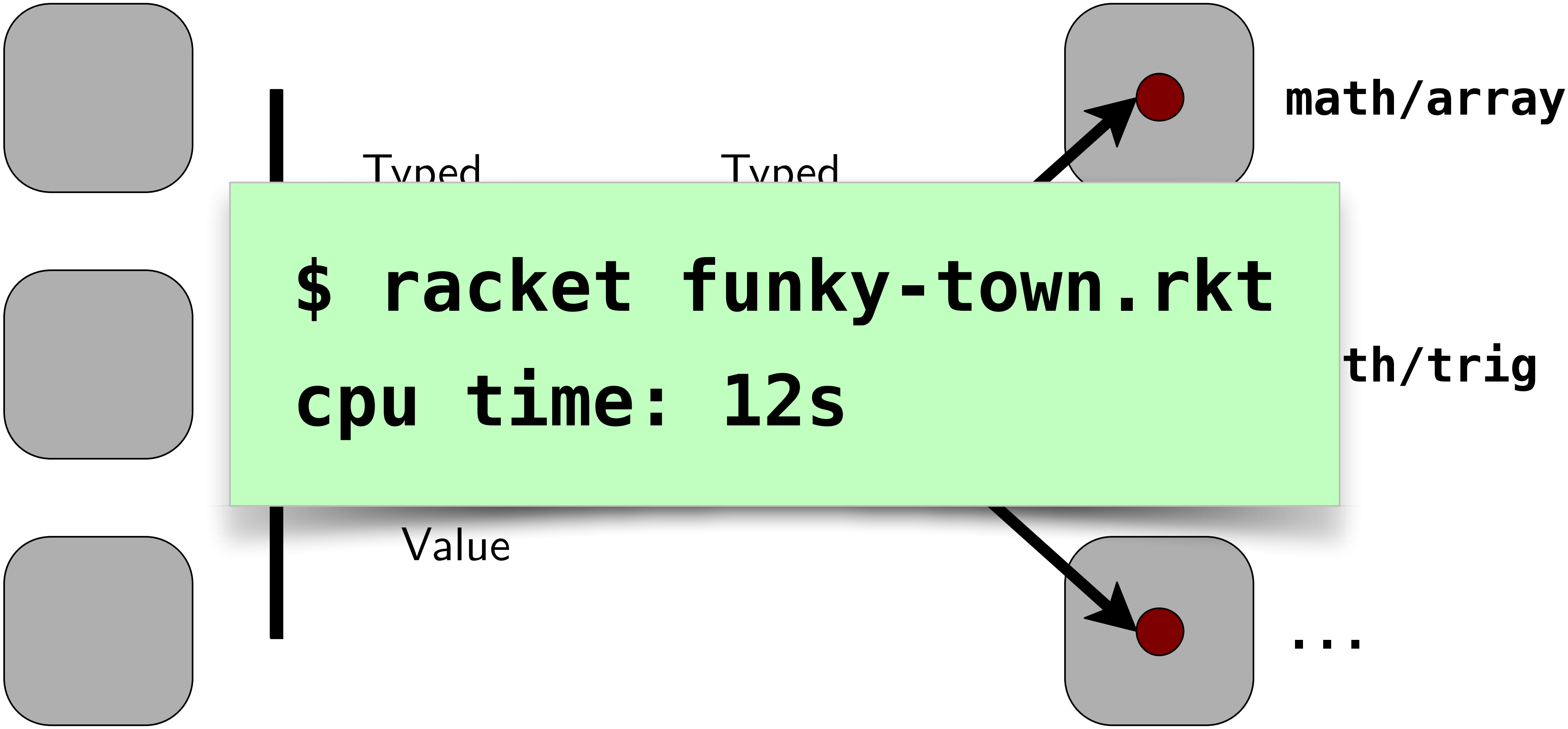


```
Contracts account for 73.77% of running time
     (17568 / 23816 ms)

  6210 ms : Array-unsafe-proc
            (-> Array (-> (vectorof Int) any))
  3110 ms : array-append*
            (->* ((listof Array)) (Int) Array)
  2776 ms : unsafe-build-array
            (-> (vectorof Int) [...] Array)
  ...
```

- Report costs per feature instance
- 1 instance: Costs in N locations
- Solution: fix contract usage


Untyped component
Typed component
Value
Proxied value
Contract boundary
math/array
math/trig
...


Typed
component
Typed
component
Value
math/array
math/trig
...

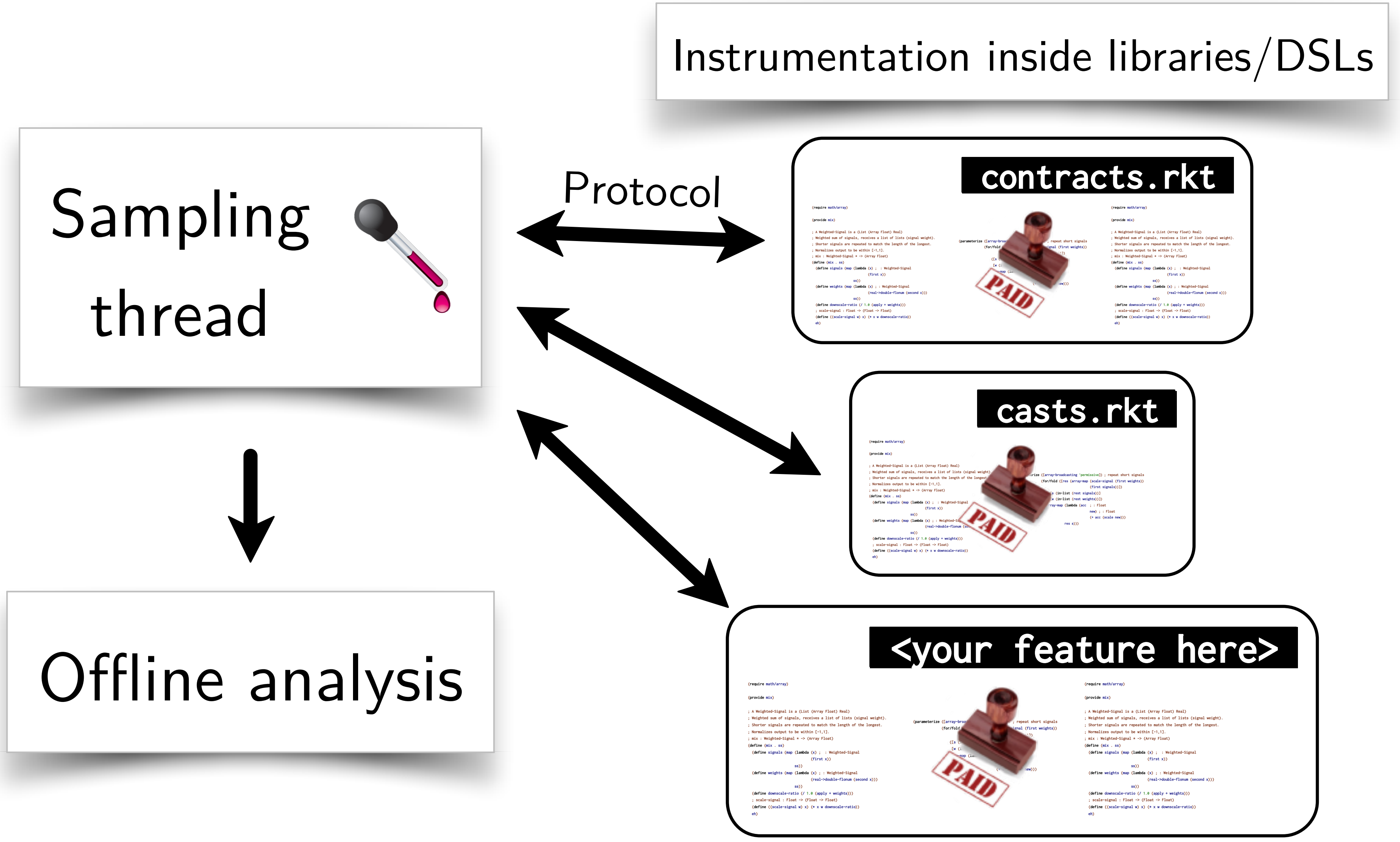


```
$ racket funky-town.rkt
cpu time: 12s
```

The Library Author's View

How to extend the tool

Architecture



Instrumentation inside libraries/DSLs

Sampling thread

Protocol

Offline analysis

contracts.rkt

casts.rkt

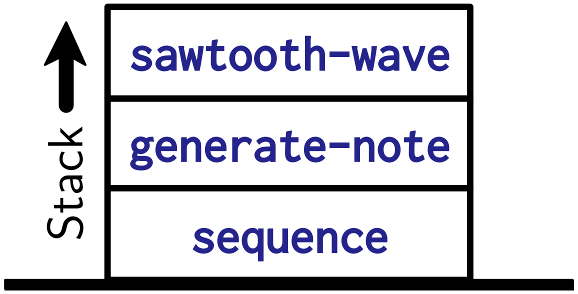
<your feature here>

```
(require math/array)

(provide mix)

; A Weighted-Signal is a (List (Array Float) Real)
; Weighted sum of signals, receives a list of lists (signal weight).
; Shorter signals are repeated to match the length of the longest.
; Normalizes output to be within [-1,1].
; mix : Weighted-Signal * -> (Array Float)
(define (mix . ss)
  (define signals (map (lambda (x) ;  : Weighted-Signal
                         (first x))
                       ss))
  (define weights (map (lambda (x) ; : Weighted-Signal
                         (real->double-flonum (second x)))
                       ss))
  (define downscale-ratio (/ 1.0 (apply + weights)))
  ; scale-signal : Float -> (Float -> Float)
  (define ((scale-signal w) x) (* x w downscale-ratio))
  eh)
```

```
erize ([array-broadcasting 'permissive]) ; repeat short signals
      (for/fold ([res (array-map (scale-signal (first weights))
                                 (first signals))])
                [s (in-list (rest signals))]
                [w (in-list (rest weights))])
      rray-map (lambda (acc  ; : Float
                                 new) ; : Float
                                 (+ acc (scale new)))
                   res s)))
```

PAID

Observing Feature Code



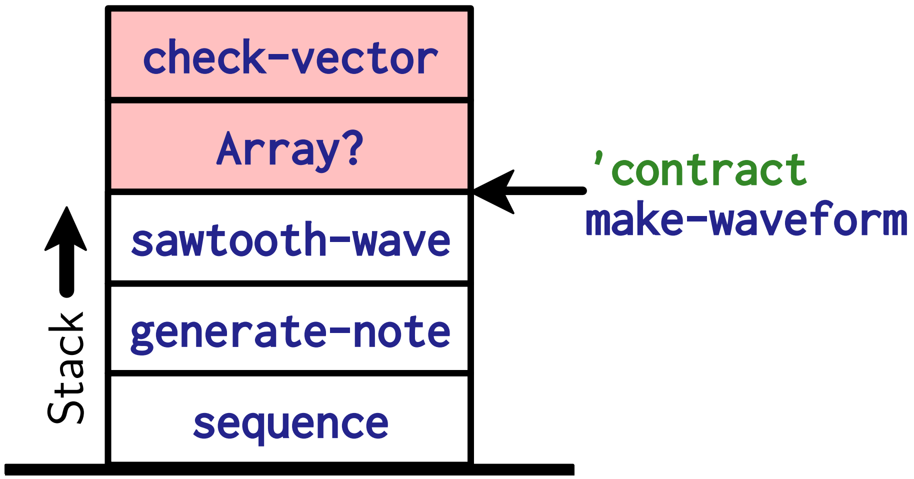
Stack

sawtooth-wave

generate-note

sequence

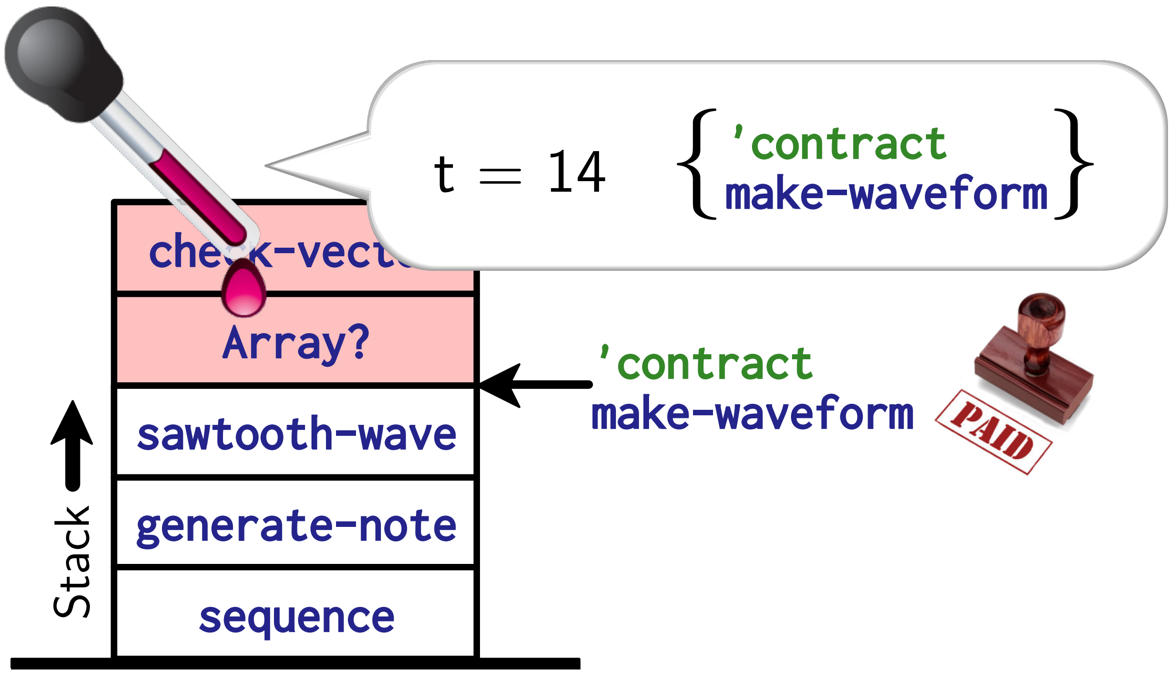
Observing Feature Code



```
(define (sawtooth-wave ...)
  (make-waveform ...)
  ...)
```

Mark present = Feature code is running

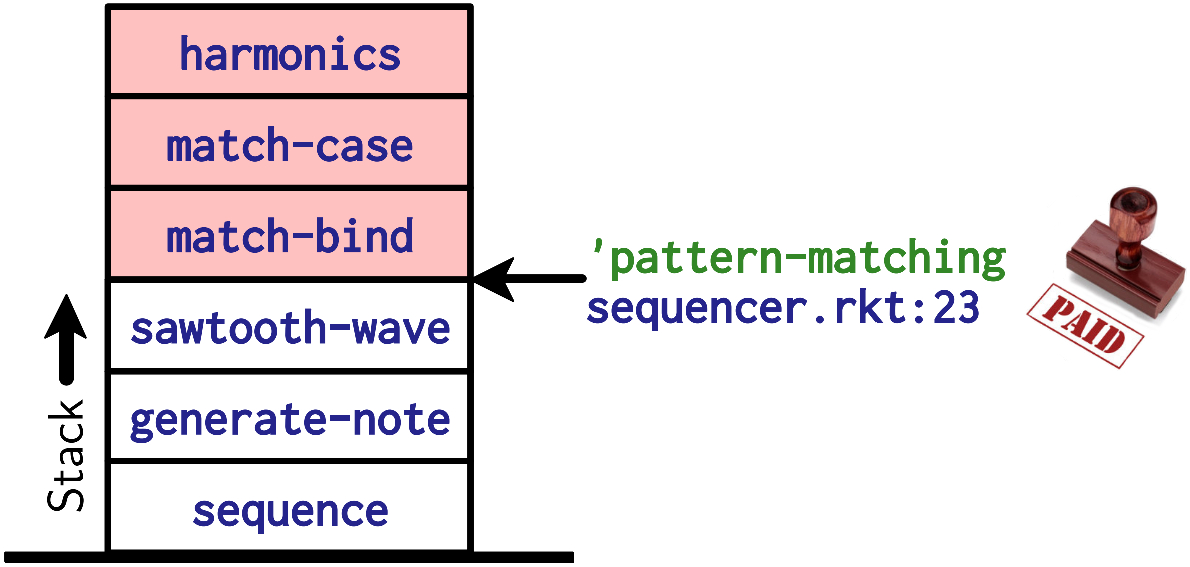
Observing Feature Code



```
(define (sawtooth-wave ...)
  (make-waveform ...)
  ...)
```

Mark present = Feature code is running

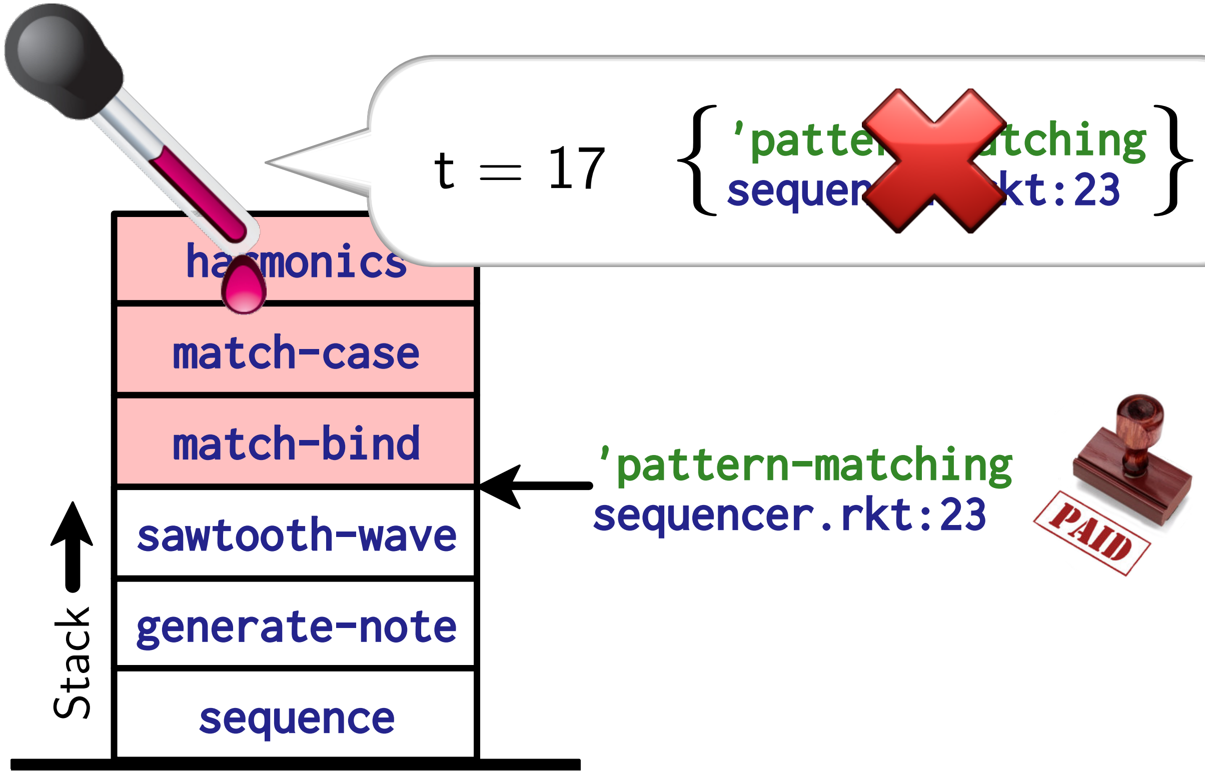
Observing Feature Code



```
(define (sawtooth-wave ...)
  ...
  (match signal
    [<pattern>
     ... (harmonics ...)]
    ...))
```

Mark present = Feature code is running

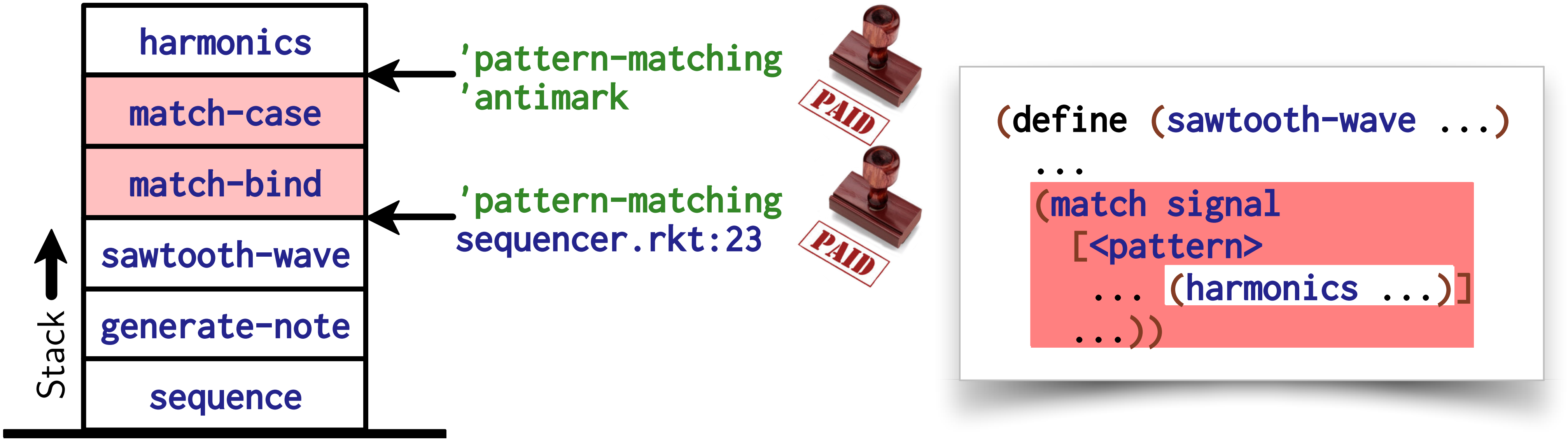
Observing Feature Code



```
(define (sawtooth-wave ...)
  ...
  (match signal
    [<pattern>
     ... (harmonics ...)]
    ...))
```

Mark present = Feature code is running

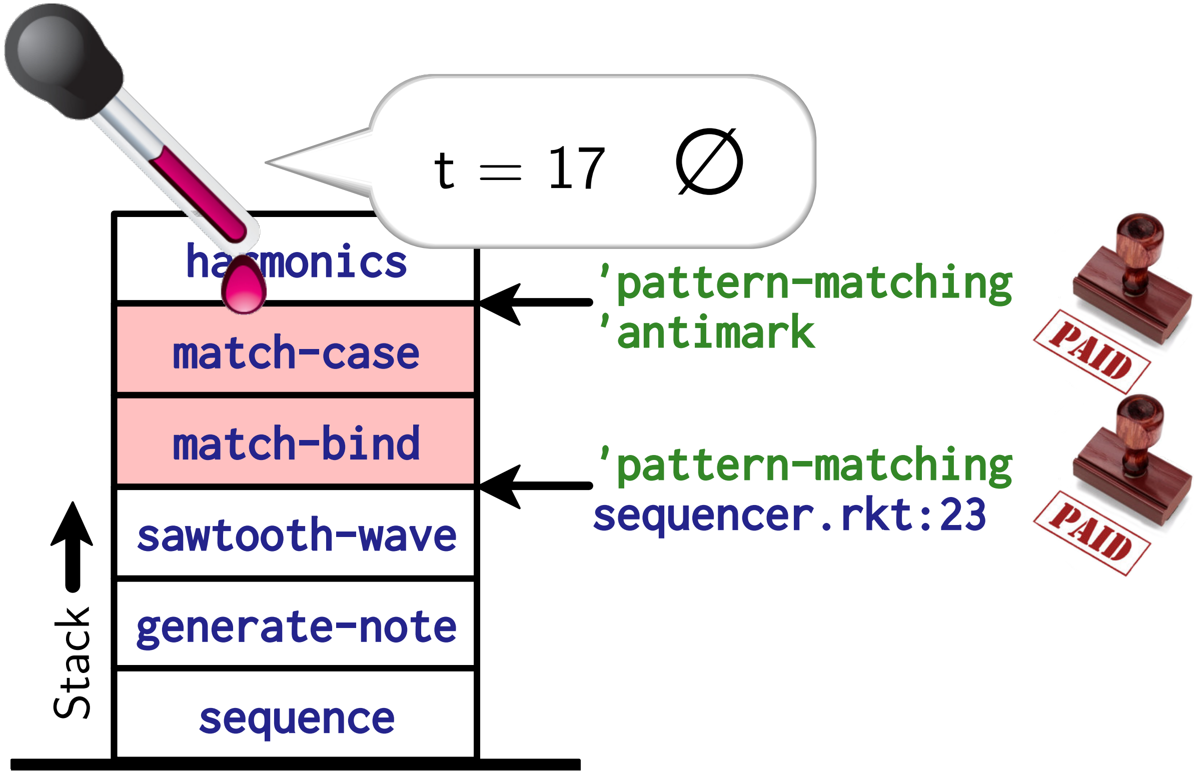
Observing Feature Code



```
(define (sawtooth-wave ...)
  ...
  (match signal
    [<pattern>
     ... (harmonics ...)]
    ...))
```

Antimark on top = Feature code is **not** running

Observing Feature Code



t = 17 ∅

harmonics

match-case

match-bind

sawtooth-wave

generate-note

sequence

Stack

'pattern-matching
'antimark

'pattern-matching
sequencer.rkt:23

PAID

PAID

```
(define (sawtooth-wave ...)
  ...
  (match signal
    [<pattern>
     ... (harmonics ...)]
    ...))
```

Antimark on top = Feature code is **not** running

If you still have room

Offline analysis

In the paper

Structurally rich features

In the paper

Instrumentation control

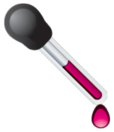
In the paper

The Tool Builder's View

How to build a similar tool

Necessary Ingredients

- Stack marking

  - Continuation marks (Racket, JavaScript, .Net, R)
  - Stack reflection (Smalltalk), stack introspection (GHC), etc.
- Sampling thread 
- Protocol (see previous section)
- Offline analysis

If you have those, you can build an FSP!

Future Work: Beyond Racket

- Works in Racket. Elsewhere?
- Ongoing work: R
  - Features: Object slices, summaries, etc.
  - Implementing continuation marks is easy!

Future Work: Beyond Sampling

- Event-based profiling
  - ➡ e.g. log messages
- Feature entry/exit events + timestamps
- No marking necessary!

Evaluation

How well does the tool work

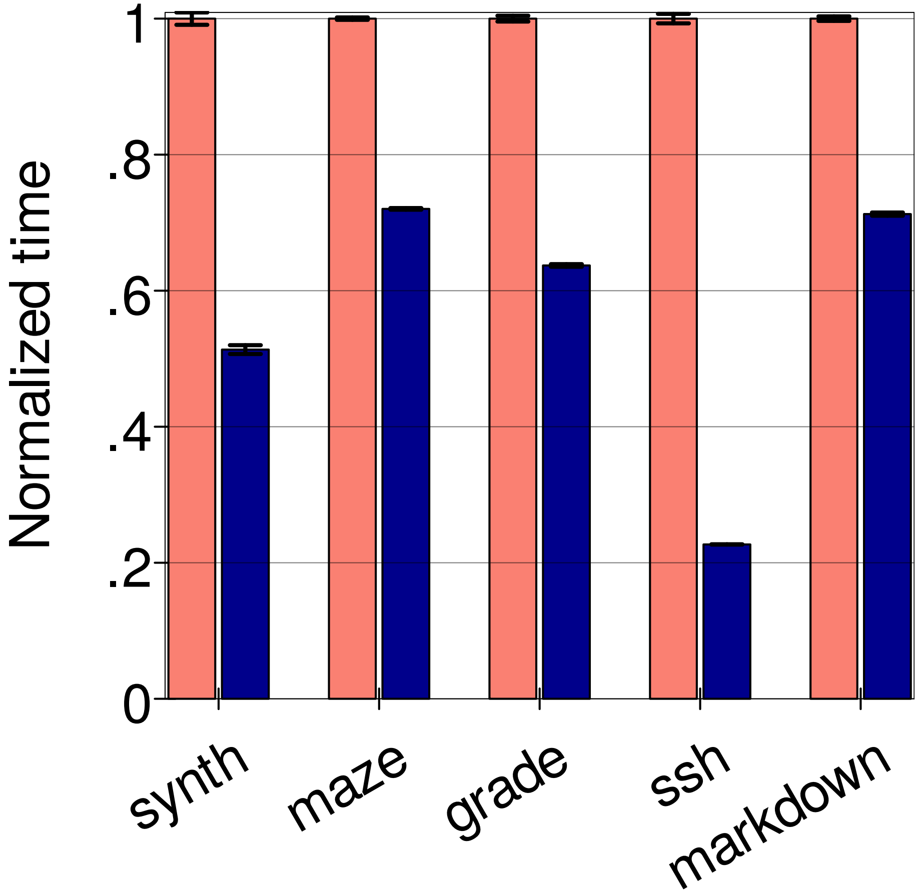
Performance Impact

Experiment

- Take existing Racket programs
- Run the feature-specific profiler
- Fix uses of features mentioned in the report
- Measure performance impact (running time)

🟥	Before:	Non-optimized
🟦	After:	Fixed feature usage

Execution time, lower is better



Instrumentation Effort

Feature	LOC
Contracts	183
Output	11
Generic sequences	18
Casts and assertions	37
Parser backtracking	18
Security policies	23
Marketplace processes	7
Pattern matching	18
Method dispatch	12
Keyword arguments	50

Reasonable for library creators

← 35 minutes for creator!
(+ 40 for extra analysis)

The take-away

The take-away

- Reporting costs in terms of ***feature instances***
- Extensible via ***marking + sampler protocol***
- Build yours using ***stack marking*** and ***sampling***

The take-away

- Reporting costs in terms of ***feature instances***
- Extensible via ***marking*** + ***sampler protocol***
- Build yours using ***stack marking*** and ***sampling***

`download.racket-lang.org`

```
raco pkg install feature-profile
```