

Optimization Coaching

Vincent St-Amour

Sam Tobin-Hochstadt

Matthias Felleisen

PLT

OOPSLA 2012 - October 23th, 2012

```

#lang typed/racket/base
(require racket/math racket/flonum
         (except-in racket/fixnum fl->fx fl)
         "flonum.rkt"
         "flonap-struct.rkt")
(provide flonap-flip-horizontal flonap-flip-vertical flonap-transpose
         flonap-cw-rotate flonap-ccw-rotate
         (struct-out invertible-2d-function) Flonap-Transform
         transform-compose rotate-transform whirl-and-pinch-transform
         flonap-transform)
(: (flonap-flip-horizontal (flonap -> flonap))
   (define (flonap-flip-horizontal fn)
     (match-define (flonap vs c w h) fn)
     (define w-1 (fx- w 1))
     (inline-build-flonap c w h (λ (k x y z)
      (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x))))))
   (define (flonap-flip-vertical fn)
     (match-define (flonap vs c w h) fn)
     (define h-1 (fx- h 1))
     (inline-build-flonap c w h (λ (k x y z)
      (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y))))))
   (define (flonap-transpose fn)
     (match-define (flonap vs c w h) fn)
     (inline-build-flonap c h w (λ (k x y z)
      (unsafe-flvector-ref vs (coords->index c w k y x))))
     (define (flonap-cw-rotate fn)
       (match-define (flonap vs c w h) fn)
       (define h-1 (fx- h 1))
       (inline-build-flonap c h w (λ (k x y z)
        (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))
       (define (flonap-ccw-rotate fn)
         (match-define (flonap vs c w h) fn)
         (define w-1 (fx- w 1))
         (inline-build-flonap c h w (λ (k x y z)
          (unsafe-flvector-ref vs (coords->index c w k y (fx- w-1 x))))))
         (struct: invertible-2d-function [(f : (Flonum Flonum -> (values Flonum Flonum)))]
          [g : (Flonum Flonum -> (values Flonum Flonum))])
         (define-type Flonap-Transform (Intruder Integer -> invertible-2d-function))
         (: transform-compose (Flonap-Transform Flonap-Transform -> Flonap-Transform))
         (define ((transform-compose t1 t2) w h)
           (match-define (invertible-2d-function f1 g1) (t1 w h))
           (match-define (invertible-2d-function f2 g2) (t2 w h))
           (invertible-2d-function (λ: [(x : Flonum) (y : Flonum)]
             (let-values [(x1 y1) (f2 x y)]
               (f1 x1 y1))
              (λ: [(x : Flonum) (y : Flonum)]
                (let-values [(x1 y1) (g1 x y)]
                  (g2 x1 y1))))))
         (: flonap-transform (case-> (flonap Flonap-Transform -> flonap)
          (flonap Flonap-Transform Real Real Real -> flonap)))
         (define flonap-transform
           (case-lambda
            [(t)
             (match-define (flonap vs c w h) fn)
             (match-define (invertible-2d-function f g) (t w h))
             (define x-min +inf.0)
             (define x-max -inf.0)
             (define y-min +inf.0)
             (define y-max -inf.0)
             (let: y-loop : Void [(y : Integer) 0]
              (when (y <= h)
                (let: x-loop : Void [(x : Integer) 0]
                 (cond [(x <= w)
                        (define-values (new-x new-y) (f (+ 0.5 (fx->fl x)) (+ 0.5 (fx->fl y))))
                        (when (new-x < . x-min) (set! x-min new-x))
                        (when (new-x > . x-max) (set! x-max new-x))
                        (when (new-y < . y-min) (set! y-min new-y))
                        (when (new-y > . y-max) (set! y-max new-y))
                        (x-loop (fx+ x 1))]
                      [else
                       (y-loop (fx+ y 1))]))
                 (flonap-transform fn t x-min x-max y-min y-max))
                (let [(x-min (read-double-flonum x-min))
                      (x-max (read-double-flonum x-max))
                      (y-min (read-double-flonum y-min))
                      (y-max (read-double-flonum y-max))]
                  (match-define (flonap vs c w h) fn)
                  (match-define (invertible-2d-function f g) (t w h))
                  (define int-x-min (fl->fx (ceiling x-min)))
                  (define int-x-max (fl->fx (ceiling x-max)))
                  (define int-y-min (fl->fx (ceiling y-min)))
                  (define int-y-max (fl->fx (ceiling y-max)))
                  (define new-w (- int-x-max int-x-min))
                  (define new-h (- int-y-max int-y-min))
                  (define x-offset (+ 0.5 (fx->fl int-x-min)))
                  (define y-offset (+ 0.5 (fx->fl int-y-min)))
                  (inline-build-flonap
                   c new-w new-h
                   (λ (k x y z)
                     (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                                                         (+ (fx->fl y) y-offset)))
                      (flonap-bilinear-ref fn k old-x old-y))))))
            ]))

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fl->fx fl)
         racket/math racket/math
         "flonum.rkt"
         "flonap-struct.rkt"
         "flonap-stats.rkt")
(provide flonap-lift flonap-lift2 flonap-lift-helper flonap-lift-helper2
         flonap-flms flonap-flms2 flonap-flms3 flonap-flms4 flonap-flms5 flonap-flms6 flonap-flms7
         flonap-flms8 flonap-flms9 flonap-flms10 flonap-flms11 flonap-flms12 flonap-flms13 flonap-flms14
         flonap-flms15 flonap-flms16 flonap-flms17 flonap-flms18 flonap-flms19 flonap-flms20
         flonap-normalize flonap-multiply-alpha flonap-divide-alpha)
; =====
; Unary
(: flonap-lift-helper : (Float -> Float) -> (flonap -> flonap))
(define (flonap-lift-helper f)
  (λ: [(fn : flonap)]
    (match-define (flonap vs c w h) fn)
    (flonap (inline-build-flvector (* c w h) (λ (i) (f (unsafe-flvector-ref vs i))))
            c w h)))
(: flonap-lift ((Flonum -> Real) -> (flonap -> flonap)))
(define (flonap-lift op)
  (flonap-lift-helper (λ (x) (real->double-flonum (op x)))))
(define flms3 (flonap-lift-helper *)
  (define flms4 (flonap-lift-helper abs))
  (define flms5 (flonap-lift-helper sq))
  (define flms6 (flonap-lift-helper sin))
  (define flms7 (flonap-lift-helper cos))
  (define flms8 (flonap-lift-helper tan))
  (define flms9 (flonap-lift-helper fllog))
  (define flms10 (flonap-lift-helper exp))
  (define flms11 (flonap-lift-helper flsqrt))
  (define flms12 (flonap-lift-helper flasin))
  (define flms13 (flonap-lift-helper acos))
  (define flms14 (flonap-lift-helper atan))
  (define flms15 (flonap-lift-helper floor))
  (define flms16 (flonap-lift-helper ceiling))
  (define flms17 (flonap-lift-helper truncate))
  (define flms18 (flonap-lift-helper (λ (x) (if (x = . 0.0) 1.0 0.0))))
; ; ;
; Binary
(: flonap-lift-helper2 : Symbol (Float Float -> Float) -> ((U Real flonap) (U Real flonap) -> flonap))
(define (flonap-lift-helper2 name f)
  (let:
   [(x1 (U Real flonap)) (x2 (U Real flonap))]
    (cond
     [(and (real? x1) (real? x2))
      (error name "expected at least one flonap argument; given ~e and ~e" x1 x2)]
     [(real? x1) (let [(fml (real->double-flonum x1))]
                   ((flonap-lift-helper (λ (v) (f v1 v2)) x1) fml))]
     [(real? x2) (let [(fml2 (real->double-flonum x2)]
                   ((flonap-lift-helper (λ (v) (f v1 v2)) x2) fml2))]
     [else
      (match-define (flonap vs1 c1 w1 h1) x1)
      (match-define (flonap vs2 c2 w2 h2) x2)
      (cond
       [(not (and (= w1 w2) (= h1 h2)))
        (error name "expected same-size flonaps; given sizes ~e-e and ~e-e" w1 w2 h1 h2)]
       [(= c1 c2)
        (define rns-vs (make-flvector n)
          (flonap (inline-build-flvector n (λ (i) (f (unsafe-flvector-ref vs1 i)
                                                       (unsafe-flvector-ref vs2 i))))
                  c1 w1 h1))
        [(= c1 1) (inline-build-flonap
                   c2 w1 h1
                   (λ (k x y i) (f (unsafe-flvector-ref vs1 (coords->index 1 w 0 x y))
                                   (unsafe-flvector-ref vs2 i)))]
        [(= c2 1) (inline-build-flonap
                   c1 w1 h1
                   (λ (k x y i) (f (unsafe-flvector-ref vs1 i)
                                   (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y)))]
        [else
         (error name (string-append "expected flonaps with the same number of components, "
                                     "or a flonap with 1 component and any same-size flonap; "
                                     "given flonaps with ~e and ~e components"
                                     c1 c2))]]))
     (: flonap-lift2 (Symbol (Flonum Flonum -> Real) -> ((U Real flonap) (U Real flonap) -> flonap)))
     (define (flonap-lift2 name f)
       (flonap-lift-helper2 name (λ (x y) (real->double-flonum (f x y))))
       (define fn (flonap-lift-helper2 'fn +))
       (define fm (flonap-lift-helper2 'fn -))
       (define fa (flonap-lift-helper2 'fn *)
       (define fb (flonap-lift-helper2 'fn /))
       (define fmin (flonap-lift-helper2 'fmin min))
       (define fmax (flonap-lift-helper2 'fmax max))
       (: flonap-normalize (flonap -> flonap))
       (define (flonap-normalize fn)
         (define-values (v-min v-max) (flonap-extreme-values fn))
         (define v-size (+ v-max v-min))
         (let* [(fn (fn v v-min))
                (fn (fn (v-size . . 0.0) fn) (fn v v-size))]]
           fn)
       (define fndiv/zero
         (flonap-lift-helper2 'fndiv/zero (λ (x y) (if (y = . 0.0) 0.0 (/ x y))))
       (: flonap-divide-alpha (flonap -> flonap))
       (define (flonap-divide-alpha fn)
         (match-define (flonap c w h) fn)
         (cond [(c . = . 1) fn]
              [else
               (define alpha-fn (flonap-ref-component fn 0))
               (flonap-append-components alpha-fn (fndiv/zero (flonap-drop-components fn 1) alpha-fn))]]))

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fx->fl fx)
         racket/math racket/math
         "flonum.rkt"
         "flonap.rkt")
(provide deep-flonap-deep-flonap? deep-flonap-argb deep-flonap-z
         deep-flonap-width deep-flonap-height deep-flonap-z-min deep-flonap-z-max
         deep-flonap-size deep-flonap-alpha deep-flonap-rgb
         flonap->deep-flonap
         : Sizing
         deep-flonap-inset deep-flonap-trim deep-flonap-scale deep-flonap-resize
         : Z-adjusting
         deep-flonap-scale-z deep-flonap-smooth-z deep-flonap-raise deep-flonap-tilt
         deep-flonap-emboss
         deep-flonap-bulge deep-flonap-bulge-round deep-flonap-bulge-round-rect
         deep-flonap-bulge-spheroid deep-flonap-bulge-horizontal deep-flonap-bulge-vertical
         deep-flonap-bulge-ripple
         : Compositing
         deep-flonap-pin deep-flonap-pin*
         deep-flonap-lt-superimpose deep-flonap-lc-superimpose deep-flonap-lb-superimpose
         deep-flonap-ct-superimpose deep-flonap-cc-superimpose deep-flonap-rb-superimpose
         deep-flonap-rt-superimpose deep-flonap-rc-superimpose deep-flonap-rb-superimpose
         deep-flonap-vc-append deep-flonap-vc-append
         deep-flonap-hc-append deep-flonap-hc-append deep-flonap-hb-append)
(struct: deep-flonap ([argb : flonap] [z : flonap])
  #:transparent
  #:guard
  (λ (argb-fn z-fn name)
   (match-define (flonap _ 4 w h) argb-fn)
   (match-define (flonap _ 1 zw zh) z-fn)
   (unless (and (= w zw) (= h zh))
    (error "deep-flonap
            "expected flonaps of equal dimension; given dimensions ~e-e and ~e-e" w h zw zh))
   (values argb-fn z-fn)))
(: flonap->deep-flonap (flonap -> deep-flonap))
(define (flonap->deep-flonap argb-fn)
  (match-define (flonap _ 4 w h) argb-fn)
  (deep-flonap-argb-fn (make-flonap 1 w h)))
(: deep-flonap-width (deep-flonap -> Nonnegative-Fixnum))
(define (deep-flonap-width dfn)
  (define w (flonap-width (deep-flonap-argb dfn)))
  (with-asserts [(w nonnegative-fixnum?)
                w])
  (: deep-flonap-height (deep-flonap -> Nonnegative-Fixnum))
  (define h (flonap-height (deep-flonap-argb dfn)))
  (with-asserts [(h nonnegative-fixnum?)
                h])
  (: deep-flonap-z-min (deep-flonap -> flonap))
  (define (deep-flonap-z-min dfn)
    (flonap-min-value (deep-flonap-z dfn)))
  (: deep-flonap-z-max (deep-flonap -> flonap))
  (define (deep-flonap-z-max dfn)
    (flonap-max-value (deep-flonap-z dfn)))
  (: deep-flonap-size (deep-flonap -> (values Nonnegative-Fixnum Nonnegative-Fixnum)))
  (define (deep-flonap-size dfn)
    (values (deep-flonap-width dfn) (deep-flonap-height dfn)))
  (: deep-flonap-alpha (deep-flonap -> flonap))
  (define (deep-flonap-alpha dfn)
    (flonap-ref-component (deep-flonap-argb dfn) 0))
  (: deep-flonap-rgb (deep-flonap -> flonap))
  (define (deep-flonap-rgb dfn)
    (flonap-drop-components (deep-flonap-argb dfn) 1))
; =====
; Z adjusters
(: deep-flonap-scale-z (deep-flonap (U Real flonap) -> deep-flonap))
(define (deep-flonap-scale-z dfn z)
  (match-define (deep-flonap argb-fn z-fn) dfn)
  (deep-flonap-argb-fn (fn' z-fn z)))
(: deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
(define (deep-flonap-smooth-z dfn o)
  (let [(o (exact->inexact o))]
    (match-define (deep-flonap argb-fn z-fn) dfn)
    (define new-z-fn (flonap-blur z-fn o))
    (deep-flonap-argb-fn new-z-fn)))
; deep-flonap-raise and everything derived from it observe an invariant:
; when z is added, added z must be 0.0 everywhere alpha is 0.0
(: deep-flonap-raise (deep-flonap (U Real flonap) -> deep-flonap))
(define (deep-flonap-raise dfn z)
  (match-define (deep-flonap argb-fn z-fn) dfn)
  (define alpha-fn (deep-flonap-alpha dfn))
  (deep-flonap-argb-fn (fn' z-fn (fn' alpha-fn z))))
(: deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
(define (deep-flonap-emboss dfn xy-ant z-ant)
  (let [(o (/ xy-ant 3.0))]
    (define z-fn (flonap-normalize (deep-flonap-alpha dfn)))
    (define new-z-fn (fn' (flonap-blur z-fn o) z-ant))
    (deep-flonap-raise dfn new-z-fn)))
(: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> Flonum) -> deep-flonap))
(define (deep-flonap-bulge-helper dfn f)
  (let ()
   (define-values (w h) (deep-flonap-size dfn))
   (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
   (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
   (define z-fn
     (inline-build-flonap
      1 w h
      (λ (x y)
        (f (- (/ (fx->fl x) half-x-size) 1.0)
            (- (/ (fx->fl y) half-y-size) 1.0))))))
     (deep-flonap-raise dfn z-fn)))
  (: deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
  (define (deep-flonap-bulge dfn f)
    (deep-flonap-bulge-helper dfn (λ (cx cy) (real->double-flonum (f cx cy))))

```



```

#lang typed/racket/base
(require racket/match racket/math racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         "flonum.rkt"
         "flonap-struct.rkt")
(provide flonap-flip-horizontal flonap-flip-vertical flonap-transpose
         flonap-cw-rotate flonap-ccw-rotate
         (struct-out invertible-2d-function) Flonap-Transform
         transform-compose rotate-transform whirl-and-pinch-transform
         Flonap-Transform)
(: flonap-flip-horizontal (flonap -> flonap))
(define (flonap-flip-horizontal fn)
  (match-define (flonap vs c w h) fn)
  (define w-1 (fx- w 1))
  (inline-build-flonap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))))
(define (flonap-flip-vertical fn)
  (match-define (flonap vs c w h) fn)
  (define h-1 (fx- h 1))
  (inline-build-flonap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y) z))))))
(define (flonap-transpose fn)
  (match-define (flonap vs c w h) fn)
  (inline-build-flonap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x y z))))))
(define (flonap-cw-rotate fn)
  (match-define (flonap vs c w h) fn)
  (define h-1 (fx- h 1))
  (inline-build-flonap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) z))))))
(define (flonap-ccw-rotate fn)
  (match-define (flonap vs c w h) fn)
  (define w-1 (fx- w 1))
  (inline-build-flonap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- w-1 y) z))))))
(struct invertible-2d-function ([f : (Flonum Flonum -> (values Flonum Flonum))])
  #:transparent)
(define-type Flonap-Transform (Integer Integer -> invertible-2d-function))
(: transform-compose (Flonap-Transform Flonap-Transform -> Flonap-Transform))
(define ((transform-compose t1 t2) w h)
  (match-define (invertible-2d-function f1 g1) (t1 w h))
  (match-define (invertible-2d-function f2 g2) (t2 w h))
  (invertible-2d-function (λ (x : Flonum) [y : Flonum])
    (let-values (((k x y) (f2 x y))
                (f1 x y))
      (λ (x : Flonum) [y : Flonum])
        (let-values (((k x y) (g1 x y))
                    (g2 x y))
          (g2 x y))))))
(: flonap-transform (case-> (flonap Flonap-Transform -> flonap)
  (flonap Flonap-Transform Real Real Real -> flonap)
  #:transparent))
(define flonap-transform
  (case-lambda
    [(t)
     (match-define (flonap vs c w h) fn)
     (match-define (invertible-2d-function f g) (t w h))
     (define x-min +inf.0)
     (define x-max -inf.0)
     (define y-min +inf.0)
     (define y-max -inf.0)
     (let [y-loop : Void [y : Integer 0]]
       (when (y = fxc . h)
         (let [x-loop : Void [x : Integer 0]]
           (cond [(x = fxc . w)
                  (define-values (new-x new-y) (f (+ 0.5 (fx->fl x)) (+ 0.5 (fx->fl y)))
                    (when (new-x < . x-min) (set! x-min new-x))
                    (when (new-x > . x-max) (set! x-max new-x))
                    (when (new-y < . y-min) (set! y-min new-y))
                    (when (new-y > . y-max) (set! y-max new-y))
                    (x-loop (fx+ x 1)))]
                 [else
                  (y-loop (fx+ y 1))])]))
     (flonap-transform fn t x-min x-max y-min y-max))
    [(fn t x-min x-max y-min y-max)
     (let [(x-min (real->double-flonum x-min))
           (x-max (real->double-flonum x-max))
           (y-min (real->double-flonum y-min))
           (y-max (real->double-flonum y-max))]
       (match-define (flonap vs c w h) fn)
       (match-define (invertible-2d-function f g) (t w h))
       (define int-x-min (fl->fx (ceiling x-min)))
       (define int-y-min (fl->fx (ceiling y-min)))
       (define int-x-max (fl->fx (ceiling x-max)))
       (define int-y-max (fl->fx (ceiling y-max)))
       (define new-w (- int-x-max int-x-min))
       (define new-h (- int-y-max int-y-min))
       (define x-offset (+ 0.5 (fx->fl int-x-min)))
       (define y-offset (+ 0.5 (fx->fl int-y-min)))
       (inline-build-flonap
        c new-w new-h
        (λ (k x y z)
          (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                                           (+ (fx->fl y) y-offset)))
            (flonap-bilinear-ref fn k old-x old-y))))))])

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         racket/match racket/math
         "flonum.rkt"
         "flonap-struct.rkt"
         "flonap-stats.rkt")
(provide flonap-lift flonap-lift2 flonap-lift-helper flonap-lift-helper2
         fmsq fmsb fmsr fmsi fmsc fmat fmsl fmsp fmsq fmsin fmscos fmatan
         fmsf fms . fm . fmin fmsmax
         flonap-normalize flonap-multiply-alpha flonap-divide-alpha)
; =====
; Unary
(: flonap-lift-helper : (Float -> Float) -> (flonap -> flonap))
(define (flonap-lift-helper f)
  (λ (fn : flonap)
    (match-define (flonap vs c w h) fn)
    (flonap (inline-build-flvector (* c w h) (λ (i) (f (unsafe-flvector-ref vs i))))
            c w h)))
(: flonap-lift ((Flonum -> Real) -> (flonap -> flonap)))
(define (flonap-lift op)
  (flonap-lift-helper (λ (x) (real->double-flonum (op x)))))
(define fmsq (flonap-lift-helper *)
  (define fmsb (flonap-lift-helper -)
  (define fmsr (flonap-lift-helper sq)
  (define fmsi (flonap-lift-helper sin)
  (define fmsc (flonap-lift-helper cos)
  (define fmsn (flonap-lift-helper tan)
  (define fmsl (flonap-lift-helper log)
  (define fmsp (flonap-lift-helper exp)
  (define fmsq (flonap-lift-helper sqrt)
  (define fmsin (flonap-lift-helper asin)
  (define fmscos (flonap-lift-helper acos)
  (define fmatan (flonap-lift-helper atan)
  (define fmsfloor (flonap-lift-helper floor)
  (define fmsceiling (flonap-lift-helper ceiling)

```



```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fx->fl fl->fx)
         racket/match racket/math
         "flonum.rkt"
         "flonap.rkt")
(provide deep-flonap-deep-flonap? deep-flonap-argb deep-flonap-z
         deep-flonap-width deep-flonap-height deep-flonap-z-min deep-flonap-z-max
         deep-flonap-size deep-flonap-alpha deep-flonap-rgb
         flonap->deep-flonap
         ; Sizing
         deep-flonap-inset deep-flonap-trim deep-flonap-scale deep-flonap-resize
         ; Z-adjusting
         deep-flonap-scale-z deep-flonap-smooth-z deep-flonap-raise deep-flonap-tilt
         deep-flonap-emboss
         deep-flonap-bulge deep-flonap-bulge-round deep-flonap-bulge-round-rect
         deep-flonap-bulge-spheroid deep-flonap-bulge-horizontal deep-flonap-bulge-vertical
         deep-flonap-bulge-ripple
         ; Compositing
         deep-flonap-pin deep-flonap-pin*
         deep-flonap-lt-superimpose deep-flonap-lc-superimpose deep-flonap-lb-superimpose
         deep-flonap-ct-superimpose deep-flonap-cc-superimpose deep-flonap-cb-superimpose
         deep-flonap-rt-superimpose deep-flonap-rc-superimpose deep-flonap-rb-superimpose
         deep-flonap-vc-append deep-flonap-vc-superimpose deep-flonap-vc-append
         deep-flonap-ht-append deep-flonap-hc-append deep-flonap-hb-append)
(struct: deep-flonap ([argb : flonap] [z : flonap])
  #:transparent
  #:guard
  (λ (argb-fn z-fn name)
    (match-define (flonap _ 4 w h) argb-fn)
    (match-define (flonap _ 1 w zh) z-fn)
    (unless (and (= w zw) (= h zh))
      (error 'deep-flonap
              "expected flonaps of equal dimension; given dimensions ~e-e and ~e-e" w h zw zh))
    (values argb-fn z-fn)))
(: flonap->deep-flonap (flonap -> deep-flonap))
(define (flonap->deep-flonap argb-fn)
  (match-define (flonap _ 4 w h) argb-fn)
  (match-define (flonap _ 1 w h) argb-fn)
  (let [(z-fn (make-flonap 1 w h))
        (h (deep-flonap -> Nonnegative-Fixnum))
        (p-width dfn)
        (-width (deep-flonap-argb dfn))
        (nonnegative-fixnum?)]
    (ht (deep-flonap -> Nonnegative-Fixnum))
    (p-height dfn)
    (-height (deep-flonap-argb dfn))
    (nonnegative-fixnum?))
  (n (deep-flonap -> Flonum))
  (p-z-min dfn)
  (deep-flonap-z dfn))
  (x (deep-flonap -> Flonum))
  (p-z-max dfn)
  (deep-flonap-z dfn))
  (deep-flonap -> (values Nonnegative-Fixnum Nonnegative-Fixnum))
  (p-size dfn)
  (map-width dfn) (deep-flonap-height dfn))
  (a (deep-flonap -> flonap))
  (p-alpha dfn)
  (nent (deep-flonap-argb dfn) 0))
  (deep-flonap -> flonap))
  (p-rgb dfn)
  (onents (deep-flonap-argb dfn) 1))
; =====
; Z (deep-flonap (U Real flonap) -> deep-flonap)
(define (deep-flonap-scale-z dfn z)
  (match-define (deep-flonap argb-fn z-fn) dfn)
  (deep-flonap-argb-fn (fn* z-fn z)))
(: deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
(define (deep-flonap-smooth-z dfn o)
  (let [(o (exact->inexact o))]
    (match-define (deep-flonap argb-fn z-fn) dfn)
    (define new-z-fn (flonap-blur z-fn o))
    (deep-flonap-argb-fn new-z-fn)))
; deep-flonap-raise and everything derived from it observe an invariant:
; when z is added, added z must be 0.0 everywhere alpha is 0.0
(: deep-flonap-raise (deep-flonap (U Real flonap) -> deep-flonap))
(define (deep-flonap-raise dfn z)
  (match-define (deep-flonap argb-fn z-fn) dfn)
  (define alpha-fn (deep-flonap-alpha dfn))
  (deep-flonap-argb-fn (fn* z-fn (fn* alpha-fn z))))
(: deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
(define (deep-flonap-emboss dfn xy-ant z-ant)
  (let [(o (/ xy-ant 3.0))]
    (define z-fn (flonap-normalize (deep-flonap-alpha dfn))
    (define new-z-fn (fn* (flonap-blur z-fn o) z-ant))
    (deep-flonap-raise dfn new-z-fn)))
(: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> Flonum) -> deep-flonap))
(define (deep-flonap-bulge-helper dfn f)
  (let ()
    (define-values (w h) (deep-flonap-size dfn))
    (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
    (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
    (define z-fn
      (inline-build-flonap
       1 w h
       (λ (x y z)
         (f (- (/ (fx->fl x) half-x-size) 1.0)
             (- (/ (fx->fl y) half-y-size) 1.0))))))
    (deep-flonap-raise dfn z-fn)))
(: deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
(define (deep-flonap-bulge dfn f)
  (deep-flonap-bulge-helper dfn (λ (cx cy) (real->double-flonum (f cx cy))))

```



```
#lang typed/racket/base
(require racket/match racket/math racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         "flonum.rkt"
         "flomap-struct.rkt")
(provide flomap-flip-horizontal flomap-flip-vertical flomap-transpose
         flomap-cw-rotate)
(struct-out invert
  transform-compose
  flomap-transform)
(: flomap-flip-horizontal
  (define (flomap-flip-hor
    (match-define (flomap
      (define w-1 (fx- 1)
      (inline-build-flomap
    (define (flomap-flip-vert
    (match-define (flomap
      (define h-1 (fx- 1)
      (inline-build-flomap
    (define (flomap-transpose
    (match-define (flomap
      (inline-build-flomap
    (define (flomap-cw-rotate
    (match-define (flomap
      (define h-1 (fx- 1)
      (inline-build-flomap
    (define (flomap-cw-rotate
    (match-define (flomap
      (define w-1 (fx- 1)
      (inline-build-flomap
    (struct: invertible-2d
  (define-type Flomap-Trans
  (: transform-compose (F
  (define ((transform-com
  (match-define (invert
  (match-define (invert
  (invertible-2d-functio
```

```
(: flomap-transform (cas
(define flomap-transform
  (case-lambda
    [(fx 1)
     (match-define (flom
     (match-define (inve
     (define x-min +inf
     (define x-max -inf
     (define y-min +inf
     (define y-max -inf
     (let: y-loop :void
     (when (y = fx- 1)
       (let: x-loop :
       (cond [(x = -
             (define
             (when
             (when
             (when
             (x-loop
             (else
             (y-loop
             (flomap-transform
             [(fx t x-min x-max x
             (let [(x-max (real
                   (y-min (real
                   (y-max (real
             (match-define (li
             (match-define (li
             (define int-x-min
             (define int-x-max
             (define int-y-min
             (define int-y-max
             (define new-w (-
             (define new-h (-
             (define x-offset
             (define y-offset
             (inline-build-fl
             c new-w new-h
             (A (k x y)
             (define-values
             (flomap-bilin
```

```
#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         "flonum.rkt"
         "flomap-struct.rkt"
         "flomap-stats.rkt")
(provide flomap-lift flomap-lift2 flomap-liftf
         flomap-flms flomap-flmsc flomax flomaxc
         flomaxd flomaxdc flomaxf flomaxfc
         flomap-normalize flomap-multiply-alpha)
; =====
; Unary
(: flomap-lift-helper : (Float -> Float) -> (
  (define (flomap-lift-helper f)
    (lambda (fl : flonum)
      (match-define (flomap vs c w h) fm)
      (flomap (inline-build-ffvector (* c w h)
        c w)))
  (: flomap-lift ((Flonum -> Real) -> (flomap ->
  (define (flomap-lift op)
    (flomap-lift-helper (lambda (x) (real->double-fl
  (define flmsc (flomap-lift-helper abs))
  (define flmsq (flomap-lift-helper sq))
  (define flmsin (flomap-lift-helper sin))
  (define flmcos (flomap-lift-helper cos))
  (define flmtan (flomap-lift-helper tan))
  (define flmlog (flomap-lift-helper log))
  (define flmexp (flomap-lift-helper exp))
  (define flmsqrt (flomap-lift-helper sqrt))
  (define flmasin (flomap-lift-helper asin))
  (define flmacos (flomap-lift-helper acos))
  (define flmtan (flomap-lift-helper atan))
  (define flmround (flomap-lift-helper round))
  (define flmfloor (flomap-lift-helper floor))
  (define flmceiling (flomap-lift-helper ceiling)
```

```
neil@schroder:~/plt/collects/images$ find . -name 'compiled' -exec rm
-rf \{\} \;

neil@schroder:~/plt/bin$ time ./raco setup --no-docs -l images
...
raco setup: --- parallel build using 8 processes ---
raco setup: 7 making: images
raco setup: 7 making: images/scribblings
raco setup: 7 making: images/icons
raco setup: 7 making: images/icons/private
...
real    2m25.441s
user    2m35.570s
sys     0m3.370s
```

```
Now with "images/private" first:

neil@schroder:~/plt/collects/images$ find . -name 'compiled' -exec rm
-rf \{\} \;

neil@schroder:~/plt/bin$ time ./raco setup --no-docs -l images/private
...
real    1m13.332s
user    1m46.640s
sys     0m3.120s
```

2 hours later...

```
Neil I

((= c2 1) (inline-build-flomap
  c1 w h
  (lambda (k x y) (f (unsafe-flvector-ref vs1 i)
                    (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y))))))
  (error name (string-append "expected flomaps with the same number of components, "
    "or a flomap with 1 component and any same-size flomap;"
    "given flomaps with +e and -e components")
    c1 c2))))))
(: flomap-lift2 (Symbol) (Flonum Flonum -> Real) -> ((U Real Flonum) (U Real Flonum) -> flomap))
(define (flomap-lift2 name f)
  (flomap-lift-helper2 name (lambda (x y) (real->double-flonum (f x y))))
  (define f+ (flomap-lift-helper2 'f+ +))
  (define f- (flomap-lift-helper2 'f- -))
  (define f* (flomap-lift-helper2 'f* *))
  (define f/ (flomap-lift-helper2 'f/ /))
  (define fmin (flomap-lift-helper2 'fmin min))
  (define fmax (flomap-lift-helper2 'fmax max))
  (: flomap-normalize (flomap -> flomap))
  (define (flomap-normalize fm)
    (define-values (v-min v-max) (flomap-extreme-values fm))
    (define v-size (+ v-max v-min))
    (let* [(f+ (f+ fm v-min))
           (f- (f- fm v-max))
           (f (lambda (v) (if (v-size = . 0.0) fm (f+ fm v-size)))]
      (define fndiv/zero
        (flomap-lift-helper2 'fndiv/zero (lambda (x y) (if (y = . 0.0) 0.0 (/ x y))))
      (: flomap-divide-alpha (flomap -> flomap))
      (define (flomap-divide-alpha fm)
        (match-define (flomap _ c w h) fm)
        (cond [(c = - 1) fm]
              [else
               (define alpha-fm (flomap-ref-component fm 0)
                 (flomap-append-components alpha-fm (fndiv/zero (flomap-drop-components fm 1) alpha-fm)))]))
    (deep-flomap-smooth-z (deep-flomap Real -> deep-flomap))
    (define (deep-flomap-smooth-z dfm o)
      (let [(o (exact->inexact o))]
        (match-define (deep-flomap argb-fm z-fm dfm)
          (define new-z-fm (flomap-blur z-fm o))
          (deep-flomap argb-fm new-z-fm)))
      : deep-flomap-raise and everything derived from it observe an invariant:
      : when z is added, added z must be 0.0 everywhere alpha is 0.0
      (: deep-flomap-raise (deep-flomap (U Real Flonum) -> deep-flomap))
      (define (deep-flomap-raise dfm z)
        (match-define (deep-flomap argb-fm z-fm dfm)
          (define alpha-fm (deep-flomap-alpha dfm))
          (deep-flomap argb-fm (f+ z-fm (f* alpha-fm z))))
      (: deep-flomap-emboss (deep-flomap Real (U Real Flonum) -> deep-flomap))
      (define (deep-flomap-emboss dfm xy-ant z-ant)
        (let [(o (/ xy-ant 3.0))]
          (define z-fm (flomap-normalize (deep-flomap-alpha dfm)))
          (define new-z-fm (f+ (flomap-blur z-fm o) z-ant))
          (deep-flomap-raise dfm new-z-fm)))
      (: deep-flomap-bulge-helper (deep-flomap (Flonum Flonum -> Flonum) -> deep-flomap))
      (define (deep-flomap-bulge-helper dfm f)
        (let ()
          (define-values (w h) (deep-flomap-size dfm))
          (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
            (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
            (define z-fm
              (inline-build-flomap
                1 w h
                (lambda (x y)
                  (f (- (/ (fx->fl x) half-x-size) 1.0)
                    (- (/ (fx->fl y) half-y-size) 1.0))))
              (deep-flomap-raise dfm z-fm)))
          (: deep-flomap-bulge (deep-flomap (Flonum Flonum -> Real) -> deep-flomap))
          (define (deep-flomap-bulge dfm f)
            (deep-flomap-bulge-helper dfm (A (cx cy) (real->double-flonum (f cx cy)))))
```

images

2:35. This is


```
#lang typed/racket/base
(require racket/match racket/math racket/flonum
  (except-in racket/fixnum fl->fx fx->fl)
  "flonum.rkt"
  "flomap-struct.rkt")
(provide flomap-flip-horizontal flomap-flip-vertical flomap-transpose
  flomap-cw-rotate flomap-ccw-rotate
  (struct-out invertible-2d-function) Flomap-Transform
  transform-compose rotate-transform whirl-and-pinch-transform
  Flomap-Transform)
(: flomap-flip-horizontal (flomap -> flomap))
(define (flomap-flip-horizontal fm)
  (match-define (flomap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flomap c w h (lambda (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))))
(define (flomap-flip-vertical fm)
  (match-define (flomap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flomap c w h (lambda (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y) z))))))
(define (flomap-transpose fm)
  (match-define (flomap vs c w h) fm)
  (inline-build-flomap c h w (lambda (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k y z))))))
(define (flomap-cw-rotate fm)
  (match-define (flomap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flomap c w h (lambda (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) z))))))
(define (flomap-ccw-rotate fm)
  (match-define (flomap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flomap c h w (lambda (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- w-1 y) z))))))
(struct invertible-2d-function ([f : (Flonum Flonum -> (values Flonum Flonum))])
  #:fields [f])
(define-type Flomap-Transform (Invertible-2d-function))
(: transform-compose (Flomap-Transform Flomap-Transform -> Flomap-Transform))
(define (transform-compose t1 t2)
  (match-define (invertible-2d-function f1 g1) (t1 w h))
  (match-define (invertible-2d-function f2 g2) (t2 w h))
  (invertible-2d-function (lambda (x : Flonum) (y : Flonum)
    (let-values ([x1 (f1 x)] [y1 (f2 x)])
      (let-values ([x2 (g1 x1 y1)] [y2 (g2 x1 y1)])
        (values x2 y2)))))))
(: flomap-transform (case-> (Flomap Flomap-Transform -> Flomap)
  (flomap Flomap-Transform Real Real Real -> flomap)))
(define flomap-transform
  (case-lambda
    [(fx t)
     (match-define (flomap vs c w h) fm)
     (match-define (invertible-2d-function f) (t w h))
     (define x-min +inf.0)
     (define x-max -inf.0)
     (define y-min +inf.0)
     (define y-max -inf.0)
     (let [y-loop : Void (y : Integer 0)]
       (when (y = fx . h)
         (let [x-loop : Void (x : Integer 0)]
           (cond [(x = fx . w)
                  (define-values (new-x new-y) (f (+ 0.5 (fx->fl x) (+ 0.5 (fx->fl y))))
                 (when (new-x < . x-min) (set! x-min new-x))
                 (when (new-x > . x-max) (set! x-max new-x))
                 (when (new-y < . y-min) (set! y-min new-y))
                 (when (new-y > . y-max) (set! y-max new-y))
                 (x-loop (fx- x 1))]
                 [else
                  (y-loop (fx- y 1))]))]))
     (flomap-transform fm t x-min x-max y-min y-max)
     [(fm t x-min x-max y-min y-max)
      (let [(x-min (real->double-flonum x-min))
            (x-max (real->double-flonum x-max))
            (y-min (real->double-flonum y-min))
            (y-max (real->double-flonum y-max))]
        (match-define (flomap vs c w h) fm)
        (match-define (invertible-2d-function f) (t w h))
        (define int-x-min (fl->fx (floor x-min)))
        (define int-x-max (fl->fx (ceiling x-max)))
        (define int-y-min (fl->fx (floor y-min)))
        (define int-y-max (fl->fx (ceiling y-max)))
        (define new-w (- int-x-max int-x-min))
        (define new-h (- int-y-max int-y-min))
        (define x-offset (+ 0.5 (fx->fl int-x-min)))
        (define y-offset (+ 0.5 (fx->fl int-y-min)))
        (inline-build-flomap
          c new-w new-h
          (lambda (k x y z)
            (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
              (+ (fx->fl y) y-offset)))
            (flomap-bilinear-ref fm k old-x old-y))))))]))]
```

```
#lang typed/racket/base
(require racket/flonum
  (except-in racket/fixnum fl->fx fx->fl)
  racket/match racket/math
  "flonum.rkt"
  "flomap-struct.rkt"
  "flomap-stats.rkt")
(provide flomap-lift flomap-lift2 flomap-lift-helper flomap-lift-helper2
  fmsq fmsq3 fmsin fmsos fmsin fmsos fmsin fmsos fmsin
  fmsq fmsq3 fmsin fmsos fmsin fmsos fmsin fmsos fmsin
  fmsq fmsq3 fmsin fmsos fmsin fmsos fmsin fmsos fmsin
  flomap-normalize flomap-multiply-alpha flomap-divide-alpha)
;
; Unary
; ~~~~~
(define (flomap-lift-helper f)
  (lambda ([fn : flomap])
    (match-define (flomap vs c w h) fm)
    (flomap (inline-build-flvector (* c w h) (lambda (i) (if (unsafe-flvector-ref vs i))))
      c w h)))
; ~~~~~
(define (flomap-lift op)
  (lambda ([flomap : flomap] (x) (real->double-flonum (op x))))
  (define fmsq (flomap-lift-helper 'sq))
  (define fmsq3 (flomap-lift-helper 'sq3))
  (define fmsin (flomap-lift-helper 'sin))
  (define fmsos (flomap-lift-helper 'cos))
  (define fmsin (flomap-lift-helper 'tan))
  (define flomap (flomap-lift-helper 'flomap))
  (define fmsos (flomap-lift-helper 'asin))
  (define fmsqrt (flomap-lift-helper 'sqrt))
  (define fmsin (flomap-lift-helper 'asin))
  (define fmsos (flomap-lift-helper 'acos))
  (define fmsin (flomap-lift-helper 'atan))
  (define fmsos (flomap-lift-helper 'cot))
  (define flomap (flomap-lift-helper 'floor))
  (define flomap (flomap-lift-helper 'ceil))
  (define flomap (flomap-lift-helper 'truncate))
  (define flomap (flomap-lift-helper (lambda (x) (if (x = . 0.0) 1.0 0.0))))
;
; Binary
; ~~~~~
; flomap-lift-helper2: (flomap (U Real flomap) (U Real flomap) -> (U Real flomap) (U Real flomap))
(define (flomap-lift-helper2 name f)
  (let [(fml : (U Real flomap)) (fm2 : (U Real flomap))]
    (cond
      [(and (real? fml) (real? fm2))
       (error name "expected at least one flomap argument; given ~e and ~e" fml fm2)]
      [(real? fml) (let [(fml (real->double-flonum fml))]
        ((flomap-lift-helper (lambda (v) (f v)) fml)))]
      [(real? fm2) (let [(fm2 (real->double-flonum fm2))]
        ((flomap-lift-helper (lambda (v) (f v)) fm2)))]
      [else
       (match-define (flomap vs1 c1 w1 h1) fml)
       (match-define (flomap vs2 c2 w2 h2) fm2)
       (cond
         [(not (and (= w1 w2) (= h1 h2)))
          (error name "expected same-size flomaps; given sizes ~e-e and ~e-e" w1 w2 h1 h2)]
         [(= c1 c2) (define rns-vs (make-flomap h))
          (flomap (inline-build-flvector n (lambda (i) (if (unsafe-flvector-ref vs1 i))
            (unsafe-flvector-ref vs2 i))))
          c1 w1 h1]
         [(= c2 1) (inline-build-flomap
          c2 w h
          (lambda (k x y z) (if (unsafe-flvector-ref vs1 (coords->index 1 w 0 x y))
            (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y)))))]
         [(= c2 1) (inline-build-flomap
          c1 w h
          (lambda (k x y z) (if (unsafe-flvector-ref vs1 i)
            (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y)))))]
         [else
          (error name (string-append "expected flomaps with the same number of components, "
            "or a flomap with 1 component and any same-size flomap;"
            "given flomaps with ~e and ~e components")
          (list c1 c2)))]))]]
; ~~~~~
(define (flomap-lift2 name f)
  (flomap-lift-helper2 name (lambda (x y) (real->double-flonum (f x y))))
  (define fmsq (flomap-lift-helper2 'sq))
  (define fmsq3 (flomap-lift-helper2 'sq3))
  (define fmsin (flomap-lift-helper2 'sin))
  (define fmsos (flomap-lift-helper2 'cos))
  (define fm (flomap-lift-helper2 'tan))
  (define fmsin (flomap-lift-helper2 'asin))
  (define fmsos (flomap-lift-helper2 'acos))
  (define fmsin (flomap-lift-helper2 'atan))
  (define fmsos (flomap-lift-helper2 'cot))
  (define flomap (flomap-normalize (flomap -> flomap)))
  (define (flomap-normalize fm)
    (define-values (v-min v-max) (flomap-extreme-values fm))
    (define v-size (+ v-max v-min))
    (let* ([fn (fm -/ v-min)]
           [fn (if (v-size = . 0.0) fm (fn / v-size))]]
      fn))
  (define fndiv/zero
    (lambda (flomap) (flomap-normalize (flomap-divide-alpha flomap -> flomap)))
  (define (flomap-divide-alpha flomap c w h)
  (match-define (flomap _ c w h) fm)
  (cond [(c = . 1) fm]
        [else
         (define alpha-fm (flomap-ref-component fm 0))
         (flomap-append-components alpha-fm (fndiv/zero (flomap-drop-components fm 1) alpha-fm))]]])
; ~~~~~
```

```
#lang typed/racket/base
(require racket/flonum
  (except-in racket/fixnum fx->fl fl->fx)
  racket/match racket/math
  "flonum.rkt"
  "flomap.rkt")
(provide deep-flomap-deep-flomap? deep-flomap-argb deep-flomap-z
  deep-flomap-width deep-flomap-height deep-flomap-z-min deep-flomap-z-max
  deep-flomap-size deep-flomap-alpha deep-flomap-rgb
  flomap->deep-flomap
  ; Sizing
  deep-flomap-inset deep-flomap-trim deep-flomap-scale deep-flomap-resize
  ; Z-adjusting
  deep-flomap-scale-z deep-flomap-smooth-z deep-flomap-raise deep-flomap-tilt
  deep-flomap-emboss
  deep-flomap-bulge deep-flomap-bulge-round deep-flomap-bulge-round-rect
  deep-flomap-bulge-spheroid deep-flomap-bulge-horizontal deep-flomap-bulge-vertical
  deep-flomap-bulge-ripple
  ; Compositing
  deep-flomap-pin deep-flomap-pin*
  deep-flomap-lt-superimpose deep-flomap-lc-superimpose deep-flomap-lb-superimpose
  deep-flomap-ct-superimpose deep-flomap-cc-superimpose deep-flomap-cb-superimpose
  deep-flomap-rt-superimpose deep-flomap-rc-superimpose deep-flomap-rb-superimpose
  deep-flomap-vc-append deep-flomap-vc-spread deep-flomap-vr-append
  deep-flomap-hb-append deep-flomap-hc-append deep-flomap-hb-append)
(struct: deep-flomap ([argb : flomap] [z : flomap])
  #:transparent
  #:guard
  (lambda (argb-fn z-fn name)
    (match-define (flomap _ 4 w h) argb-fm)
    (match-define (flomap _ 1 w h) z-fm)
    (unless (and (= w zw) (= h zh))
      (error "deep-flomap
        \"expected flomaps of equal dimension; given dimensions ~e-e and ~e-e\" w h zw zh))
    (values argb-fn z-fn)))
(: flomap->deep-flomap (flomap -> deep-flomap))
(define (flomap->deep-flomap argb-fm)
  (match-define (flomap _ 4 w h) argb-fm)
  (deep-flomap-argb-fm (make-flomap 1 w h)))
(: deep-flomap-width (deep-flomap -> Nonnegative-Fixnum))
(define (deep-flomap-width dfm)
  (define w (flomap-width (deep-flomap-argb dfm)))
  (with-asserts (w nonnegative-fixnum?) w))
(: deep-flomap-height (deep-flomap -> Nonnegative-Fixnum))
(define (deep-flomap-height dfm)
  (with-asserts (h nonnegative-fixnum?) h))
(: deep-flomap-z-min (deep-flomap -> flonum))
(define (deep-flomap-z-min dfm)
  (flomap-min-value (deep-flomap-z dfm)))
(: deep-flomap-z-max (deep-flomap -> flonum))
(define (deep-flomap-z-max dfm)
  (flomap-max-value (deep-flomap-z dfm)))
(: deep-flomap-size (deep-flomap -> (values Nonnegative-Fixnum Nonnegative-Fixnum)))
(define (deep-flomap-size dfm)
  (values (deep-flomap-width dfm) (deep-flomap-height dfm)))
(: deep-flomap-alpha (deep-flomap -> flomap))
(define (deep-flomap-alpha dfm)
  (flomap-ref-component (deep-flomap-argb dfm) 0))
(: deep-flomap-rgb (deep-flomap -> flomap))
(define (deep-flomap-rgb dfm)
  (flomap-drop-components (deep-flomap-argb dfm) 1))
; ~~~~~
; Z adjusters
; ~~~~~
(deep-flomap-scale-z (deep-flomap (U Real flomap) -> deep-flomap))
(define (deep-flomap-scale-z dfm z)
  (match-define (deep-flomap argb-fm z-fm) dfm)
  (deep-flomap-argb-fm (fn* z-fm z)))
(: deep-flomap-smooth-z (deep-flomap Real -> deep-flomap))
(define (deep-flomap-smooth-z dfm o)
  (let [(o (exact->inexact o))]
    (match-define (deep-flomap argb-fm z-fm) dfm)
    (define new-z-fm (flomap-blur z-fm o))
    (deep-flomap-argb-fm new-z-fm)))
; ~~~~~
; deep-flomap-raise and everything derived from it observe an invariant:
; when z is added, added z must be 0.0 everywhere alpha is 0.0
(: deep-flomap-raise (deep-flomap (U Real flomap) -> deep-flomap))
(define (deep-flomap-raise dfm z)
  (match-define (deep-flomap argb-fm z-fm) dfm)
  (define alpha-fm (deep-flomap-alpha dfm))
  (deep-flomap-argb-fm (fn* z-fm (fn* alpha-fm z)))
  (: deep-flomap-emboss (deep-flomap Real (U Real flomap) -> deep-flomap))
  (define (deep-flomap-emboss dfm xy-ant z-ant)
    (let [(o (/ xy-ant 3.0))]
      (define z-fm (flomap-normalize (deep-flomap-alpha dfm)))
      (define new-z-fm (fn* (flomap-blur z-fm o) z-ant))
      (deep-flomap-raise dfm new-z-fm))))
; ~~~~~
(define (deep-flomap-bulge-helper dfm f)
  (let ()
    (define-values (w h) (deep-flomap-size dfm))
    (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
    (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
    (define z-fm
      (inline-build-flomap
        3 w h
        (lambda (k x y z)
          (f (- (/ (fx->fl x) half-x-size) 1.0)
            (- (/ (fx->fl y) half-y-size) 1.0))))))
    (deep-flomap-raise dfm z-fm))))
; ~~~~~
; deep-flomap-blur (deep-flomap (U Real flomap) -> flomap) -> deep-flomap))
(define (deep-flomap-blur dfm)
  (deep-flomap-bulge-helper dfm (lambda (cx cy) (real->double-flonum (f cx cy)))))
```



```

#lang typed/racket/base
(require racket/match racket/math racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         "flonum.rkt"
         "flomap-struct.rkt")
(provide flomap-flip-horizontal flomap-flip-vertical flomap-transpose
         flomap-cw-rotate flomap-ccw-rotate
         (struct-out invertible-2d-function) Flomap-Transform
         transform-compose rotate-transform whirl-and-pinch-transform
         flomap-transform)
(: flomap-flip-horizontal (flomap -> flomap))
(define (flomap-flip-horizontal fm)
  (match-define (flomap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flomap c w h (lambda (k x y)
    (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))
  (define (flomap-flip-vertical fm)
    (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))
  (match-define (flomap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flomap c w h (lambda (k x y)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))
  (define (flomap-transpose fm)
    (match-define (flomap vs c w h) fm)
    (inline-build-flomap c h w (lambda (k x y)
      (unsafe-flvector-ref vs (coords->index c w k x y))))
    (define (flomap-cw-rotate fm)
      (match-define (flomap vs c w h) fm)
      (define h-1 (fx- h 1))
      (inline-build-flomap c h w (lambda (k x y)
        (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))
      (define (flomap-ccw-rotate fm)
        (match-define (flomap vs c w h) fm)
        (define w-1 (fx- w 1))
        (inline-build-flomap c h w (lambda (k x y)
          (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))
        (struct: invertible-2d-function ((f : (Flonum Flonum -> (values Flonum Flonum)))
          (f : (Flonum Flonum -> (values Flonum Flonum))))))

```

```

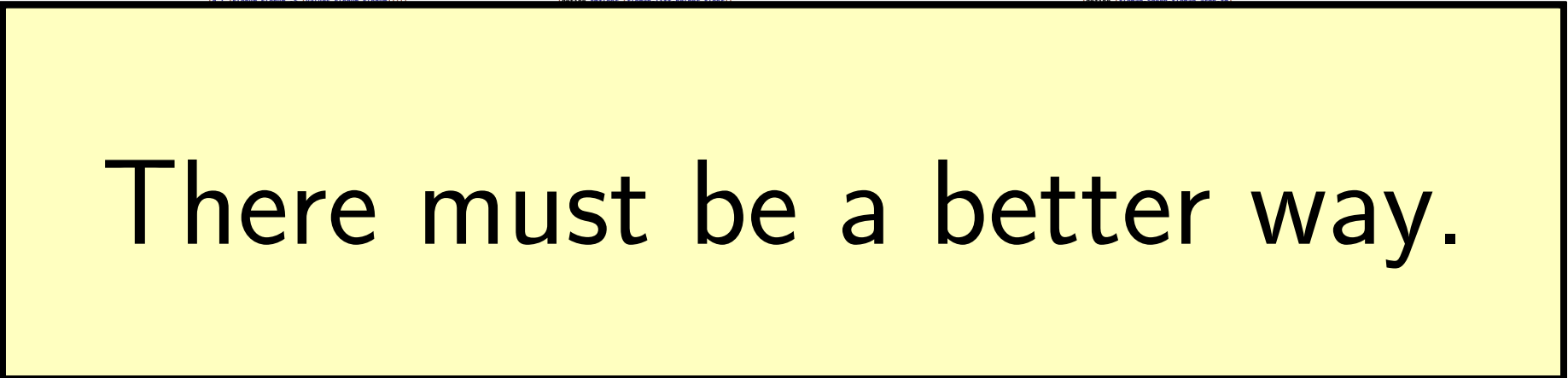
#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         racket/match racket/math
         "flonum.rkt"
         "flomap-struct.rkt"
         "flomap-stats.rkt")
(provide flomap-lift flomap-lift2 flomap-lift-helper flomap-lift-helper2
         fmsq fmsbs fmsqr fmsin fmsos fmsin fmsos fmsin fmsos fmsin
         fmsqr fmsin fmsos fmsin fmsos fmsin fmsos fmsin fmsos fmsin
         fmsqr fmsin fmsos fmsin fmsos fmsin fmsos fmsin fmsos fmsin
         fmsqr fmsin fmsos fmsin fmsos fmsin fmsos fmsin fmsos fmsin
         flomap-normalize flomap-multiply-alpha flomap-divide-alpha)
; =====
; Unary
(: flomap-lift-helper : (Float -> Float) -> (flomap -> flomap))
(define (flomap-lift-helper f)
  (lambda ([fn : flomap])
    (match-define (flomap vs c w h) fm)
    (flomap (inline-build-flvector (* c w h) (lambda (i) (f (unsafe-flvector-ref vs i))))
            c w h)))
(: flomap-lift ((Flonum -> Real) -> (flomap -> flomap)))
(define (flomap-lift op)
  (flomap-lift-helper (lambda (x) (real->double-flonum (op x))))
  (define fmsbs (flomap-lift-helper -))
  (define fmsqr (flomap-lift-helper abs))
  (define fmsin (flomap-lift-helper sin))
  (define fmsos (flomap-lift-helper cos))
  (define fmsin (flomap-lift-helper tan))
  (define fmslog (flomap-lift-helper log))
  (define fmscos (flomap-lift-helper exp))
  (define fmsqrt (flomap-lift-helper flsqrt))
  (define fmsin (flomap-lift-helper asin))
  (define fmscos (flomap-lift-helper acos))
  (define fmsatan (flomap-lift-helper atan))
  (define fmsround (flomap-lift-helper round))
  (define fmsfloor (flomap-lift-helper floor))

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fx->fl fl->fx)
         racket/match racket/math
         "flonum.rkt"
         "flomap.rkt")
(provide deep-flomap deep-flomap? deep-flomap-argb deep-flomap-z
         deep-flomap-width deep-flomap-height deep-flomap-z-min deep-flomap-z-max
         deep-flomap-size deep-flomap-alpha deep-flomap-rgb
         flomap->deep-flomap
         ; Sizing
         deep-flomap-inset deep-flomap-trim deep-flomap-scale deep-flomap-resize
         ; Z-adjusting
         deep-flomap-scale-z deep-flomap-smooth-z deep-flomap-raise deep-flomap-tilt
         deep-flomap-emboss
         deep-flomap-bulge deep-flomap-bulge-round deep-flomap-bulge-round-rect
         deep-flomap-bulge-spheroid deep-flomap-bulge-horizontal deep-flomap-bulge-vertical
         deep-flomap-bulge-ripple
         ; Compositing
         deep-flomap-pin deep-flomap-pin*
         deep-flomap-lt-superimpose deep-flomap-lc-superimpose deep-flomap-lb-superimpose
         deep-flomap-ct-superimpose deep-flomap-cc-superimpose deep-flomap-cb-superimpose
         deep-flomap-rt-superimpose deep-flomap-rc-superimpose deep-flomap-rb-superimpose
         deep-flomap-vc-append deep-flomap-vc-append deep-flomap-vr-append
         deep-flomap-ht-append deep-flomap-ht-append deep-flomap-hb-append)
(struct: deep-flomap ([argb : flomap] [z : flomap])
  #:transparent
  #:guard
  (lambda (argb-fn z-fn name)
    (match-define (flomap _ 4 w h) argb-fm)
    (match-define (flomap _ 1 zw zh) z-fm)
    (unless (and (= w zw) (= h zh))
      (error "deep-flomap
              expected flomaps of equal dimension; given dimensions -e-e-e and -e-e-e" w h zw zh))
    (values argb-fm z-fm)))
(: deep-flomap->deep-flomap (flomap -> deep-flomap))
(define (flomap->deep-flomap flomap)

```



```

[else
  (y-loop (fx- y 1)))]))
(flomap-transform fm t x-min x-max y-min y-max)
[[[fm t x-min x-max y-min y-max]
  (let ([x-min (real->double-flonum x-min)]
        [x-max (real->double-flonum x-max)]
        [y-min (real->double-flonum y-min)]
        [y-max (real->double-flonum y-max)])
    (match-define (flomap vs c w h) fm)
    (match-define (invertible-2d-function f g) (t w h))
    (define int-x-min (fl->fx (floor x-min)))
    (define int-x-max (fl->fx (ceiling x-max)))
    (define int-y-min (fl->fx (floor y-min)))
    (define int-y-max (fl->fx (ceiling y-max)))
    (define new-w (- int-x-max int-x-min))
    (define new-h (- int-y-max int-y-min))
    (define x-offset (+ 0.5 (fx->fl int-x-min)))
    (define y-offset (+ 0.5 (fx->fl int-y-min)))
    (inline-build-flomap
     c new-w new-h
     (lambda (k x y)
       (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                                         (+ (fx->fl y) y-offset)))
       (flomap-bilinear-ref fm k old-x old-y)))))]))

```

```

c1 w h
(lambda (k x y) (f (unsafe-flvector-ref vs1 i)
                  (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y)))))]))
[else
  (error name (string-append "expected flomaps with the same number of components, "
                              "or a flomap with 1 component and any same-size flomap: "
                              "given flomaps with -e and -e components")
         c1 c2)])))]))
(: flomap-lift2 (Symbol (Flonum Flonum -> Real) -> ((U Real flomap) (U Real flomap) -> flomap)))
(define (flomap-lift2 name f)
  (flomap-lift-helper2 name (lambda (x y) (real->double-flonum (f x y))))
  (define fm+ (flomap-lift-helper2 'fm+))
  (define fm- (flomap-lift-helper2 'fm-))
  (define fm* (flomap-lift-helper2 'fm*))
  (define fm/ (flomap-lift-helper2 'fm/))
  (define fmin (flomap-lift-helper2 'fmin min))
  (define fmax (flomap-lift-helper2 'fmax max))
  (: flomap-normalize (flomap -> flomap))
  (define (flomap-normalize fm)
    (define-values (v-min v-max) (flomap-extreme-values fm))
    (define v-size (- v-max v-min))
    (let* ([fm (fm- fm v-min)]
           [fm (if (v-size = . 0.0) fm (fm / fm v-size))])
      fm))
  (define fndiv/zero
    (flomap-lift-helper2 'fndiv/zero (lambda (x y) (if (y = . 0.0) 0.0 (/ x y))))
  (: flomap-divide-alpha (flomap -> flomap))
  (define (flomap-divide-alpha fm)
    (match-define (flomap _ c w h) fm)
    (cond [(c . => . 1) fm]
          [else
           (define alpha-fm (flomap-ref-component fm 0))
           (flomap-append-components alpha-fm (fndiv/zero (flomap-drop-components fm 1) alpha-fm))]]))

```

```

(deep-flomap-argb-fm (fm* z-fm z)))
(: deep-flomap-smooth-z (deep-flomap Real -> deep-flomap))
(define (deep-flomap-smooth-z dfm o)
  (let ([o (exact->inexact o)])
    (match-define (deep-flomap-argb-fm z-fm dfm)
      (define new-z-fm (deep-flomap-blur z-fm o))
      (deep-flomap-argb-fm new-z-fm)))
; deep-flomap-raise and everything derived from it observe an invariant:
; when z is added, added z must be 0.0 everywhere alpha is 0.0
(: deep-flomap-raise (deep-flomap (U Real flomap) -> deep-flomap))
(define (deep-flomap-raise dfm z)
  (match-define (deep-flomap-argb-fm z-fm dfm)
    (define alpha-fm (deep-flomap-alpha dfm))
    (deep-flomap-argb-fm (fx- z-fm (fx* alpha-fm z))))
  (: deep-flomap-emboss (deep-flomap Real (U Real flomap) -> deep-flomap))
  (define (deep-flomap-emboss dfm xy-ant z-ant)
    (let ([o (/ xy-ant 3.0)])
      (define z-fm (flomap-normalize (deep-flomap-alpha dfm)))
      (define new-z-fm (fm* (flomap-blur z-fm o) z-ant))
      (deep-flomap-raise dfm new-z-fm)))
  (: deep-flomap-bulge-helper (deep-flomap (Flonum Flonum -> Flonum) -> deep-flomap))
  (define (deep-flomap-bulge-helper dfm f)
    (let ()
      (define-values (w h) (deep-flomap-size dfm))
      (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
      (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
      (define z-fm
        (inline-build-flomap
         1 w h
         (lambda (x y)
           (f (- (/ (fx->fl x) half-x-size) 1.0)
              (- (/ (fx->fl y) half-y-size) 1.0))))))
      (deep-flomap-raise dfm z-fm)))
  (: deep-flomap-bulge (deep-flomap (Flonum Flonum -> Real) -> deep-flomap))
  (define (deep-flomap-bulge dfm f)
    (deep-flomap-bulge-helper dfm (lambda (cx cy) (real->double-flonum (f cx cy)))))

```



```
#lang typed/racket/base
(require racket/math racket/flonum
         (except-in racket/fixnum fl->fx fl->fl)
         "flonum.rkt"
         "flonap-struct.rkt")
(provide flonap-flip-horizontal flonap-flip-vertical flonap-transpose
         flonap-cw-rotate flonap-ccw-rotate
         (struct-out invertible-2d-function) Flonap-Transform
         transform-compose rotate-transform whirl-and-pinch-transform
         flonap-transform)
(: flonap-flip-horizontal (flonap -> flonap))
(define (flonap-flip-horizontal fn)
  (match-define (flonap vs c w h) fn)
  (define w-1 (fx- w 1))
  (inline-build-flonap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x))))))
(define (flonap-flip-vertical fn)
  (match-define (flonap vs c w h) fn)
  (define h-1 (fx- h 1))
  (inline-build-flonap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y))))))
(define (flonap-transpose fn)
  (match-define (flonap vs c w h) fn)
  (inline-build-flonap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k y x))))))
(define (flonap-cw-rotate fn)
  (match-define (flonap vs c w h) fn)
  (define h-1 (fx- h 1))
  (inline-build-flonap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))))
(define (flonap-ccw-rotate fn)
  (match-define (flonap vs c w h) fn)
  (define w-1 (fx- w 1))
  (inline-build-flonap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k y (fx- w-1 x))))))
(struct invertible-2d-function ([f : (Flonum Flonum -> (values Flonum Flonum))])
  #:guard
  (λ (f) (Flonum Flonum -> (values Flonum Flonum))))
(define-type Flonap-Transform (Invertible-2d-function))
(: transform-compose (Flonap-Transform Flonap-Transform -> Flonap-Transform))
(define ((transform-compose t1 t2) w h)
  (match-define (invertible-2d-function f1 g1) (t1 w h))
  (match-define (invertible-2d-function f2 g2) (t2 w h))
  (invertible-2d-function (λ: [(x : Flonum) (y : Flonum)]
    (let-values [(fx y) (f1 x y)]
      (let-values [(fx2 y2) (f2 fx y)]
        (let-values [(fx3 y3) (g1 fx2 y2)]
          (g2 x y))))))
(: flonap-transform (case-> (flonap Flonap-Transform -> flonap)
  (flonap Flonap-Transform Real Real Real -> flonap)))
(define flonap-transform
  (case-lambda
    [(t)
     (match-define (flonap vs c w h) fn)
     (match-define (invertible-2d-function f g) (t w h))
     (define x-min +inf.0)
     (define x-max -inf.0)
     (define y-min +inf.0)
     (define y-max -inf.0)
     (let [y-loop : Void [(y : Integer) 0]]
       (when (y = fxc . h)
         (let [x-loop : Void [(x : Integer) 0]]
           (cond [(x = fxc . w)
                  (define-values (new-x new-y) (f (+ 0.5 (fx->fl x)) (+ 0.5 (fx->fl y))))
                  (when (new-x < . x-min) (set! x-min new-x))
                  (when (new-x > . x-max) (set! x-max new-x))
                  (when (new-y < . y-min) (set! y-min new-y))
                  (when (new-y > . y-max) (set! y-max new-y))
                  (x-loop (fx+ x 1))]
                 [else
                  (y-loop (fx+ y 1))])])
         (flonap-transform fn t x-min x-max y-min y-max))
       [(fn t x-min x-max y-min y-max)
        (let [(x-min (real->double-flonum x-min))
              (x-max (real->double-flonum x-max))
              (y-min (real->double-flonum y-min))
              (y-max (real->double-flonum y-max))]
          (match-define (flonap vs c w h) fn)
          (match-define (invertible-2d-function f g) (t w h))
          (define int-x-min (fl->fx (ceiling x-min)))
          (define int-y-min (fl->fx (ceiling y-min)))
          (define int-x-max (fl->fx (ceiling x-max)))
          (define int-y-max (fl->fx (ceiling y-max)))
          (define new-w (- int-x-max int-x-min))
          (define new-h (- int-y-max int-y-min))
          (define x-offset (+ 0.5 (fx->fl int-x-min)))
          (define y-offset (+ 0.5 (fx->fl int-y-min)))
          (inline-build-flonap
            c new-w new-h
            (λ (k x y z)
              (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                (+ (fx->fl y) y-offset)))
              (flonap-bilinear-ref fn k old-x old-y))))))
    ])
```

```
#lang typed/racket/base
(require racket/fixnum fl->fx fl->fl)
 racket/math racket/math
 "flonum.rkt"
 "flonap-struct.rkt"
 "flonap-stats.rkt")
(provide flonap-lift flonap-lift2 flonap-lift-helper flonap-lift-helper2
         flonap-fling flonap-fling2 flonap-fling3 flonap-fling4 flonap-fling5 flonap-fling6 flonap-fling7
         flonap-fling8 flonap-fling9 flonap-fling10 flonap-fling11 flonap-fling12 flonap-fling13 flonap-fling14
         flonap-fling15 flonap-fling16 flonap-fling17 flonap-fling18 flonap-fling19 flonap-fling20 flonap-fling21
         flonap-fling22 flonap-fling23 flonap-fling24 flonap-fling25 flonap-fling26 flonap-fling27 flonap-fling28
         flonap-fling29 flonap-fling30 flonap-fling31 flonap-fling32 flonap-fling33 flonap-fling34 flonap-fling35
         flonap-fling36 flonap-fling37 flonap-fling38 flonap-fling39 flonap-fling40 flonap-fling41 flonap-fling42
         flonap-fling43 flonap-fling44 flonap-fling45 flonap-fling46 flonap-fling47 flonap-fling48 flonap-fling49
         flonap-fling50 flonap-fling51 flonap-fling52 flonap-fling53 flonap-fling54 flonap-fling55 flonap-fling56
         flonap-fling57 flonap-fling58 flonap-fling59 flonap-fling60 flonap-fling61 flonap-fling62 flonap-fling63
         flonap-fling64 flonap-fling65 flonap-fling66 flonap-fling67 flonap-fling68 flonap-fling69 flonap-fling70
         flonap-fling71 flonap-fling72 flonap-fling73 flonap-fling74 flonap-fling75 flonap-fling76 flonap-fling77
         flonap-fling78 flonap-fling79 flonap-fling80 flonap-fling81 flonap-fling82 flonap-fling83 flonap-fling84
         flonap-fling85 flonap-fling86 flonap-fling87 flonap-fling88 flonap-fling89 flonap-fling90 flonap-fling91
         flonap-fling92 flonap-fling93 flonap-fling94 flonap-fling95 flonap-fling96 flonap-fling97 flonap-fling98
         flonap-fling99 flonap-fling100)
; =====
; Unary
(: flonap-lift-helper (Float -> Float) -> (flonap -> flonap))
(define (flonap-lift-helper f)
  (λ: [(fn : flonap)]
    (match-define (flonap vs c w h) fn)
    (flonap (inline-build-flvector (* c w h) (λ (i) (f (unsafe-flvector-ref vs i))))
      c w h)))
(: flonap-lift ((Flonum -> Real) -> (flonap -> flonap)))
(define (flonap-lift op)
  (flonap-lift-helper (λ (x) (real->double-flonum (op x))))
  (define fmax (flonap-lift-helper +))
  (define fmin (flonap-lift-helper -))
  (define fmult (flonap-lift-helper *))
  (define fdiv (flonap-lift-helper /))
  (define fsqr (flonap-lift-helper sq))
  (define fsin (flonap-lift-helper sin))
  (define fcos (flonap-lift-helper cos))
  (define ftan (flonap-lift-helper tan))
  (define flng (flonap-lift-helper lng))
  (define flng2 (flonap-lift-helper lng2))
  (define flng3 (flonap-lift-helper lng3))
  (define flng4 (flonap-lift-helper lng4))
  (define flng5 (flonap-lift-helper lng5))
  (define flng6 (flonap-lift-helper lng6))
  (define flng7 (flonap-lift-helper lng7))
  (define flng8 (flonap-lift-helper lng8))
  (define flng9 (flonap-lift-helper lng9))
  (define flng10 (flonap-lift-helper lng10))
  (define flng11 (flonap-lift-helper lng11))
  (define flng12 (flonap-lift-helper lng12))
  (define flng13 (flonap-lift-helper lng13))
  (define flng14 (flonap-lift-helper lng14))
  (define flng15 (flonap-lift-helper lng15))
  (define flng16 (flonap-lift-helper lng16))
  (define flng17 (flonap-lift-helper lng17))
  (define flng18 (flonap-lift-helper lng18))
  (define flng19 (flonap-lift-helper lng19))
  (define flng20 (flonap-lift-helper lng20))
  (define flng21 (flonap-lift-helper lng21))
  (define flng22 (flonap-lift-helper lng22))
  (define flng23 (flonap-lift-helper lng23))
  (define flng24 (flonap-lift-helper lng24))
  (define flng25 (flonap-lift-helper lng25))
  (define flng26 (flonap-lift-helper lng26))
  (define flng27 (flonap-lift-helper lng27))
  (define flng28 (flonap-lift-helper lng28))
  (define flng29 (flonap-lift-helper lng29))
  (define flng30 (flonap-lift-helper lng30))
  (define flng31 (flonap-lift-helper lng31))
  (define flng32 (flonap-lift-helper lng32))
  (define flng33 (flonap-lift-helper lng33))
  (define flng34 (flonap-lift-helper lng34))
  (define flng35 (flonap-lift-helper lng35))
  (define flng36 (flonap-lift-helper lng36))
  (define flng37 (flonap-lift-helper lng37))
  (define flng38 (flonap-lift-helper lng38))
  (define flng39 (flonap-lift-helper lng39))
  (define flng40 (flonap-lift-helper lng40))
  (define flng41 (flonap-lift-helper lng41))
  (define flng42 (flonap-lift-helper lng42))
  (define flng43 (flonap-lift-helper lng43))
  (define flng44 (flonap-lift-helper lng44))
  (define flng45 (flonap-lift-helper lng45))
  (define flng46 (flonap-lift-helper lng46))
  (define flng47 (flonap-lift-helper lng47))
  (define flng48 (flonap-lift-helper lng48))
  (define flng49 (flonap-lift-helper lng49))
  (define flng50 (flonap-lift-helper lng50))
  (define flng51 (flonap-lift-helper lng51))
  (define flng52 (flonap-lift-helper lng52))
  (define flng53 (flonap-lift-helper lng53))
  (define flng54 (flonap-lift-helper lng54))
  (define flng55 (flonap-lift-helper lng55))
  (define flng56 (flonap-lift-helper lng56))
  (define flng57 (flonap-lift-helper lng57))
  (define flng58 (flonap-lift-helper lng58))
  (define flng59 (flonap-lift-helper lng59))
  (define flng60 (flonap-lift-helper lng60))
  (define flng61 (flonap-lift-helper lng61))
  (define flng62 (flonap-lift-helper lng62))
  (define flng63 (flonap-lift-helper lng63))
  (define flng64 (flonap-lift-helper lng64))
  (define flng65 (flonap-lift-helper lng65))
  (define flng66 (flonap-lift-helper lng66))
  (define flng67 (flonap-lift-helper lng67))
  (define flng68 (flonap-lift-helper lng68))
  (define flng69 (flonap-lift-helper lng69))
  (define flng70 (flonap-lift-helper lng70))
  (define flng71 (flonap-lift-helper lng71))
  (define flng72 (flonap-lift-helper lng72))
  (define flng73 (flonap-lift-helper lng73))
  (define flng74 (flonap-lift-helper lng74))
  (define flng75 (flonap-lift-helper lng75))
  (define flng76 (flonap-lift-helper lng76))
  (define flng77 (flonap-lift-helper lng77))
  (define flng78 (flonap-lift-helper lng78))
  (define flng79 (flonap-lift-helper lng79))
  (define flng80 (flonap-lift-helper lng80))
  (define flng81 (flonap-lift-helper lng81))
  (define flng82 (flonap-lift-helper lng82))
  (define flng83 (flonap-lift-helper lng83))
  (define flng84 (flonap-lift-helper lng84))
  (define flng85 (flonap-lift-helper lng85))
  (define flng86 (flonap-lift-helper lng86))
  (define flng87 (flonap-lift-helper lng87))
  (define flng88 (flonap-lift-helper lng88))
  (define flng89 (flonap-lift-helper lng89))
  (define flng90 (flonap-lift-helper lng90))
  (define flng91 (flonap-lift-helper lng91))
  (define flng92 (flonap-lift-helper lng92))
  (define flng93 (flonap-lift-helper lng93))
  (define flng94 (flonap-lift-helper lng94))
  (define flng95 (flonap-lift-helper lng95))
  (define flng96 (flonap-lift-helper lng96))
  (define flng97 (flonap-lift-helper lng97))
  (define flng98 (flonap-lift-helper lng98))
  (define flng99 (flonap-lift-helper lng99))
  (define flng100 (flonap-lift-helper lng100)))
  (let [(fn1 : (U Real flonap)) (fn2 : (U Real flonap)) (U Real flonap) -> flonap)]
    (define (flonap-lift-helper2 name f)
      (let [(fn1 : (U Real flonap)) (fn2 : (U Real flonap))]
        (cond
          [(and (real? fn1) (real? fn2))
           (error name "expected at least one flonap argument; given ~e and ~e" fn1 fn2)]
          [(real? fn1) (let [(fn1 (real->double-flonum fn1))]
            ((flonap-lift-helper (λ (v) (f fn1 v)) fn2))]
            [(real? fn2) (let [(fn2 (real->double-flonum fn2))]
              ((flonap-lift-helper (λ (v) (f v fn2)) fn1))]
            [else
             (match-define (flonap vs1 c1 w h) fn1)
             (match-define (flonap vs2 c2 w2 h2) fn2)
             (cond
               [(not (and (= w w2) (= h h2)))
                (error name "expected same-size flonaps; given sizes ~e-e and ~e-e" w h2 h2)]
               [(= c1 c2) (define n (+ c1 w h))
                (define res-vs (make-flvector n))
                (flonap (inline-build-flvector n (λ (i) (f (unsafe-flvector-ref vs1 i)
                  (unsafe-flvector-ref vs2 i))))
                  c1 w h)]
               [(= c2 1) (inline-build-flonap
                c2 w h
                (λ (k x y i) (f (unsafe-flvector-ref vs1 (coords->index 1 w 0 x y))
                  (unsafe-flvector-ref vs2 i)))]
               [(= c2 1) (inline-build-flonap
                c1 w h
                (λ (k x y i) (f (unsafe-flvector-ref vs1 i)
                  (unsafe-flvector-ref vs2 (coords->index 1 w 0 x y)))]
               [else
                (error name (string-append "expected flonaps with the same number of components, "
                  "or a flonap with 1 component and any same-size flonap;"
                  "given flonaps with ~e and ~e components"
                  c1 c2)))]
            ]))
          ]))
  (flonap-lift2 (Symbol (Flonum Flonum -> Real) -> ((U Real flonap) (U Real flonap) -> flonap)))
  (define (flonap-lift2 name f)
    (flonap-lift-helper2 name (λ (x y) (real->double-flonum (f x y))))
    (define f+ (flonap-lift-helper2 +))
    (define f- (flonap-lift-helper2 -))
    (define f* (flonap-lift-helper2 *))
    (define f/ (flonap-lift-helper2 /))
    (define fm (flonap-lift-helper2 'fn/))
    (define fmin (flonap-lift-helper2 'fmin min))
    (define fmax (flonap-lift-helper2 'fmax max))
    (flonap-normalize (flonap -> flonap))
    (define (flonap-normalize fn)
      (define-values (v-min v-max) (flonap-extreme-values fn))
      (define v-size (+ v-max v-min))
      (let* [(fn (fn v-m-in)
              (fn (if (v-size = . 0.0) fn (fn v-m-size))))
            (fn)]
        (define fndiv/zero
          (flonap-lift-helper2 'fndiv/zero (λ (x y) (if (y = . 0.0) 0.0 (/ x y))))
          (flonap-divide-alpha (flonap -> flonap))
          (define (flonap-divide-alpha fn)
            (match-define (flonap _ c w h) fn)
            (cond [(c = . 1) (fn)
                   [else
                    (define alpha-fn (flonap-ref-component fn 0))
                    (flonap-append-components alpha-fn (fndiv/zero (flonap-drop-components fn 1) alpha-fn))]]
            ])
```

```
#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fx->fl fl->fx)
         racket/math racket/math
         "flonum.rkt"
         "flonap.rkt")
(provide deep-flonap-deep-flonap? deep-flonap-argb deep-flonap-z
         deep-flonap-width deep-flonap-height deep-flonap-z-min deep-flonap-z-max
         deep-flonap-size deep-flonap-alpha deep-flonap-rgb
         flonap->deep-flonap
         ; Sizing
         deep-flonap-inset deep-flonap-trim deep-flonap-scale deep-flonap-resize
         ; Z-adjusting
         deep-flonap-scale-z deep-flonap-smooth-z deep-flonap-raise deep-flonap-tilt
         deep-flonap-emboss
         deep-flonap-bulge deep-flonap-bulge-round deep-flonap-bulge-round-rect
         deep-flonap-bulge-spheroid deep-flonap-bulge-horizontal deep-flonap-bulge-vertical
         deep-flonap-bulge-ripple
         ; Compositing
         deep-flonap-pin deep-flonap-pin*
         deep-flonap-lt-superimpose deep-flonap-lc-superimpose deep-flonap-lb-superimpose
         deep-flonap-ct-superimpose deep-flonap-cc-superimpose deep-flonap-rb-superimpose
         deep-flonap-rt-superimpose deep-flonap-rc-superimpose deep-flonap-rb-superimpose
         deep-flonap-vc-append deep-flonap-vc-append
         deep-flonap-hc-append deep-flonap-hc-append deep-flonap-hb-append)
(struct: deep-flonap ([argb : flonap] [z : flonap])
  #:transparent
  #:guard
  (λ (argb-fn z-fn name)
    (match-define (flonap _ 4 w h) argb-fn)
    (match-define (flonap _ 1 w zh) z-fn)
    (unless (and (= w zw) (= h zh))
      (error "deep-flonap
        "expected flonaps of equal dimension; given dimensions ~e-e and ~e-e" w h zw zh))
    (values argb-fn z-fn)))
(: flonap->deep-flonap (flonap -> deep-flonap))
(define (flonap->deep-flonap argb-fn)
  (match-define (flonap _ 4 w h) argb-fn)
  (deep-flonap-argb-fn (make-flonap 1 w h))
  (deep-flonap-width (deep-flonap -> Nonnegative-Fixnum))
  (define (deep-flonap-width dfn)
    (define w (flonap-width (deep-flonap-argb dfn)))
    (with-asserts [(w nonnegative-fixnum?)
      w])
    (deep-flonap-height (deep-flonap -> Nonnegative-Fixnum))
    (define h (flonap-height (deep-flonap-argb dfn)))
    (with-asserts [(h nonnegative-fixnum?)
      h])
    (deep-flonap-z-min (deep-flonap -> flonap))
    (define (deep-flonap-z-min dfn)
      (flonap-min-value (deep-flonap-z dfn)))
    (deep-flonap-z-max (deep-flonap -> flonap))
    (define (deep-flonap-z-max dfn)
      (flonap-max-value (deep-flonap-z dfn)))
    (deep-flonap-size (deep-flonap -> (values Nonnegative-Fixnum Nonnegative-Fixnum)))
    (define (deep-flonap-size dfn)
      (values (deep-flonap-width dfn) (deep-flonap-height dfn)))
    (deep-flonap-alpha (deep-flonap -> flonap))
    (define (deep-flonap-alpha dfn)
      (flonap-ref-component (deep-flonap-argb dfn) 0))
    (deep-flonap-rgb (deep-flonap -> flonap))
    (define (deep-flonap-rgb dfn)
      (flonap-drop-components (deep-flonap-argb dfn) 1))
    ; =====
    ; Z adjusters
    (deep-flonap-scale-z (deep-flonap (U Real flonap) -> deep-flonap))
    (define (deep-flonap-scale-z dfn z)
      (match-define (deep-flonap argb-fn z-fn) dfn)
      (deep-flonap-argb-fn (fn z-fn z) z))
    (deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
    (define (deep-flonap-smooth-z dfn o)
      (let [(o (exact->inexact o))]
        (match-define (deep-flonap argb-fn z-fn) dfn)
        (define new-z-fn (flonap-blur z-fn o) z-ant)
        (deep-flonap-argb-fn new-z-fn)))
    (deep-flonap-raise and everything derived from it observe an invariant:
      when z is added, added z must be 0.0 everywhere alpha is 0.0
    (deep-flonap-raise (deep-flonap (U Real flonap) -> deep-flonap))
    (define (deep-flonap-raise dfn z)
      (match-define (deep-flonap argb-fn z-fn) dfn)
      (define alpha-fn (deep-flonap-alpha dfn))
      (deep-flonap-argb-fn (fn z-fn (fn alpha-fn z)))
      (deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
      (define (deep-flonap-emboss dfn xy-ant z-ant)
        (let [(o (/ xy-ant 3.0))]
          (define z-fn (flonap-normalize (deep-flonap-alpha dfn)))
          (define new-z-fn (fn* (flonap-blur z-fn o) z-ant)
            (deep-flonap-raise dfn new-z-fn)))
        (deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> flonap) -> deep-flonap))
        (define (deep-flonap-bulge-helper dfn f)
          (let ()
            (define-values (w h) (deep-flonap-size dfn))
            (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
              (define half-y-size (- (* 0.5 (fx->fl h) 0.5))
            (define z-fn
              (inline-build-flonap
                3 w h
                (λ (x y z)
                  (f (- (/ (fx->fl x) half-x-size) 1.0)
                    (- (/ (fx->fl y) half-y-size) 1.0))))
                (deep-flonap-raise dfn z-fn)))
            (deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
            (define (deep-flonap-bulge dfn f)
              (deep-flonap-bulge-helper dfn (λ (cx cy) (real->double-flonum (f cx cy))))))
```

```

#lang typed/racket/base
(require racket/match racket/math racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         "flonum.rkt"
         "flomap-struct.rkt")
(provide flomap-flip-horizontal flomap-flip-vertical flomap-transpose
         flomap-cw-rotate flomap-ccw-rotate
         (struct-out invertible-2d-function) Flomap-Transform
         transform-compose rotate-transform whirl-and-pinch-transform
         flomap-transform)
(: flomap-flip-horizontal (flomap -> flomap))
(define (flomap-flip-horizontal fm)
  (match-define (flomap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flomap c w h (λ (k x y z)
    (define (flomap-flip-vertical fm)
      (match-define (flomap vs c w h) fm)
      (define h-1 (fx- h 1))
      (inline-build-flomap c w h (λ (k x y z)
        (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y)))))))))
      (unsafe-flvector-ref vs (coords->index c w k x (fx- w-1 x) y))))))
(define (flomap-transpose fm)
  (match-define (flomap vs c w h) fm)
  (inline-build-flomap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k y x))))))
(define (flomap-cw-rotate fm)
  (match-define (flomap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flomap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))))
(define (flomap-ccw-rotate fm)
  (match-define (flomap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flomap c h w (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x))))))
(define (flomap-flip-horiz c w h)
  (match-define (flomap-struct c w h) s)
  (define w-1 (fx- w 1))
  (inline-build-flomap c w h (λ (k x y z)
    (unsafe-flvector-ref vs (coords->index c w k x (fx- w-1 x) y))))))
(struct invertible-2d-function
  (transform-composition))
(define-type Flomap (invertible-2d-function))
(: transform-composition (Flomap -> Flomap))
(define ((transform-composition f) g)
  (match-define (invertible-2d-function f) f)
  (match-define (invertible-2d-function g) g)
  (invertible-2d-function
    (compose (invertible-2d-function f) (invertible-2d-function g))))
(: flomap-transpose (flomap -> flomap))
(define flomap-transpose
  (case-lambda
    [(fm t)
     (match-define (invertible-2d-function fm) fm)
     (define x-min (fl->fx (floor x-min)))
     (define x-max (fl->fx (ceiling x-max)))
     (define int-x-min (fl->fx (floor x-min)))
     (define int-x-max (fl->fx (ceiling x-max)))
     (define int-y-min (fl->fx (floor y-min)))
     (define int-y-max (fl->fx (ceiling y-max)))
     (define new-w (- int-x-max int-x-min))
     (define new-h (- int-y-max int-y-min))
     (define x-offset (+ 0.5 (fx->fl int-x-min)))
     (define y-offset (+ 0.5 (fx->fl int-y-min)))
     (inline-build-flomap
      c new-w new-h
      (λ (k x y z)
        (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                                         (+ (fx->fl y) y-offset)))
          (flomap-bilinear-ref fm k old-x old-y))))))])

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fl->fx fx->fl)
         racket/match racket/math
         "flonum.rkt"
         "flomap-struct.rkt"
         "flomap-stats.rkt")
(provide flomap-lift flomap-lift2 flomap-lift-helper flomap-lift-helper2
         fmax fmax2 fmax3 fmax4 fmax5 fmax6 fmax7 fmax8 fmax9 fmax10
         fmin fmin2 fmin3 fmin4 fmin5 fmin6 fmin7 fmin8 fmin9 fmin10
         flomap-normalize flomap-multiply-alpha flomap-divide-alpha)
; =====
; Unary
(: flomap-lift-helper (Float -> Float) -> (flomap -> flomap))
(define (flomap-lift-helper f)
  (λ (fm : flomap)
    (match-define (flomap vs c w h) fm)
    (flomap (inline-build-flvector (* c w h) (λ (i) (if (unsafe-flvector-ref vs i))))
            c w h)))
(: flomap-lift ((Flonum -> Real) -> (flomap -> flomap))
  (define (flomap-lift op)
    (flomap-lift-helper (λ (x) (real->double-flonum (op x)))))
  (define fmax (flomap-lift-helper +))
  (define fmax2 (flomap-lift-helper *))
  (define fmax3 (flomap-lift-helper *)*)
  (define fmax4 (flomap-lift-helper *)*)*)
  (define fmax5 (flomap-lift-helper *)*)*)*)
  (define fmax6 (flomap-lift-helper *)*)*)*)*)
  (define fmax7 (flomap-lift-helper *)*)*)*)*)*)
  (define fmax8 (flomap-lift-helper *)*)*)*)*)*)*)
  (define fmax9 (flomap-lift-helper *)*)*)*)*)*)*)*)
  (define fmax10 (flomap-lift-helper *)*)*)*)*)*)*)*)*)
  (define fmin (flomap-lift-helper -))
  (define fmin2 (flomap-lift-helper *)*)
  (define fmin3 (flomap-lift-helper *)*)*)
  (define fmin4 (flomap-lift-helper *)*)*)*)
  (define fmin5 (flomap-lift-helper *)*)*)*)*)
  (define fmin6 (flomap-lift-helper *)*)*)*)*)*)
  (define fmin7 (flomap-lift-helper *)*)*)*)*)*)*)
  (define fmin8 (flomap-lift-helper *)*)*)*)*)*)*)*)
  (define fmin9 (flomap-lift-helper *)*)*)*)*)*)*)*)*)
  (define fmin10 (flomap-lift-helper *)*)*)*)*)*)*)*)*)*)

```

```

#lang typed/racket/base
(require racket/flonum
         (except-in racket/fixnum fx->fl fl->fx)
         racket/match racket/math
         "flonum.rkt"
         "flomap.rkt")
(provide deep-flomap-deep-flomap? deep-flomap-argb deep-flomap-z
         deep-flomap-width deep-flomap-height deep-flomap-z-max deep-flomap-z-min
         deep-flomap-size deep-flomap-alpha deep-flomap-rgb flomap->deep-flomap
         ; Sizing
         deep-flomap-inset deep-flomap-trim deep-flomap-scale deep-flomap-resize
         ; Z-adjusting
         deep-flomap-scale-z deep-flomap-smooth-z deep-flomap-raise deep-flomap-tilt
         deep-flomap-emboss
         deep-flomap-bulge deep-flomap-bulge-round deep-flomap-bulge-round-rect
         deep-flomap-bulge-spheroid deep-flomap-bulge-horizontal deep-flomap-bulge-vertical
         deep-flomap-bulge-ripple
         ; Compositing
         deep-flomap-pin deep-flomap-pin*
         deep-flomap-lt-superimpose deep-flomap-ic-superimpose deep-flomap-lb-superimpose
         deep-flomap-ct-superimpose deep-flomap-cc-superimpose deep-flomap-cb-superimpose
         deep-flomap-rt-superimpose deep-flomap-rc-superimpose deep-flomap-rb-superimpose
         deep-flomap-vc-append deep-flomap-vc-append deep-flomap-vr-append
         deep-flomap-ht-append deep-flomap-ht-append deep-flomap-hb-append)
(struct: deep-flomap ([argb : flomap] [z : flomap])
  #transparent
  #guard
  (λ (argb-z-fm (flomap-argb z-fm) argb)
    (match-define (flomap _ 4 w h) argb-z-fm)
    (match-define (flomap-argb-fm 1 w h) argb)
    (inline-and (w h) z-fm)))

```

Optimization Coach
w h zw zh

20:0:

flomap-lift-helper

✘

Missed Inlining (0 success out of 46)

Consider turning this function into a macro to force inlining.

"or a flomap with 1 component and any same-size flomap,"
"given flomaps with -e and -o components")

(λ c2))]]]]]]))

```

(define flomap-lift2 (symbol (flonum flonum -> real) -> ((U Real flomap) (U Real flomap) -> flomap)))
(define (flomap-lift2 name f)
  (flomap-lift-helper2 name (λ (x y) (real->double-flonum (f x y))))
  (define f+ (flomap-lift-helper2 +))
  (define f* (flomap-lift-helper2 *))
  (define f- (flomap-lift-helper2 -))
  (define f/ (flomap-lift-helper2 /))
  (define fmin (flomap-lift-helper2 fmin min))
  (define fmax (flomap-lift-helper2 fmax max))
  (: flomap-normalize (flomap -> flomap))
  (define (flomap-normalize fm)
    (define-values (v-min v-max) (flomap-extreme-values fm))
    (define v-size (- v-max v-min))
    (let* ([fm (fm- fm v-min)]
           [fm (if (v-size . = . 0.0) fm (fm- fm v-size))])
      fm))
  (define flomap-divide-zero 'flomap-divide-zero (λ (x y) (if (y . = . 0.0) 0.0 (/ x y))))
  (: flomap-divide-alpha (flomap -> flomap))
  (define (flomap-divide-alpha fm)
    (match-define (flomap-struct c w h) fm)
    (cond [(c . = . 1) fm]
          [else
           (define alpha-fm (flomap-ref-component fm 0))
           (flomap-append-components alpha-fm (flomap-drop-components fm 1) alpha-fm)])))

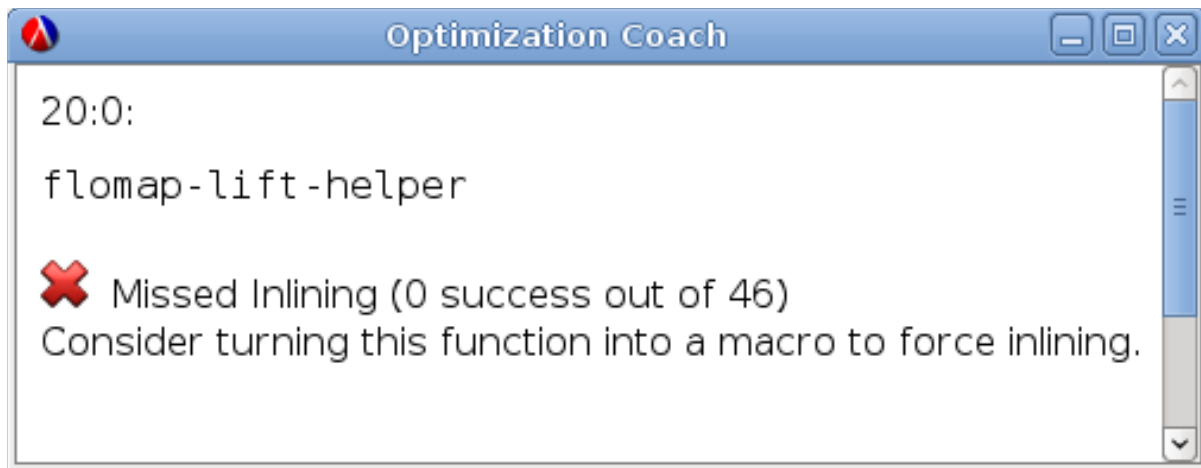
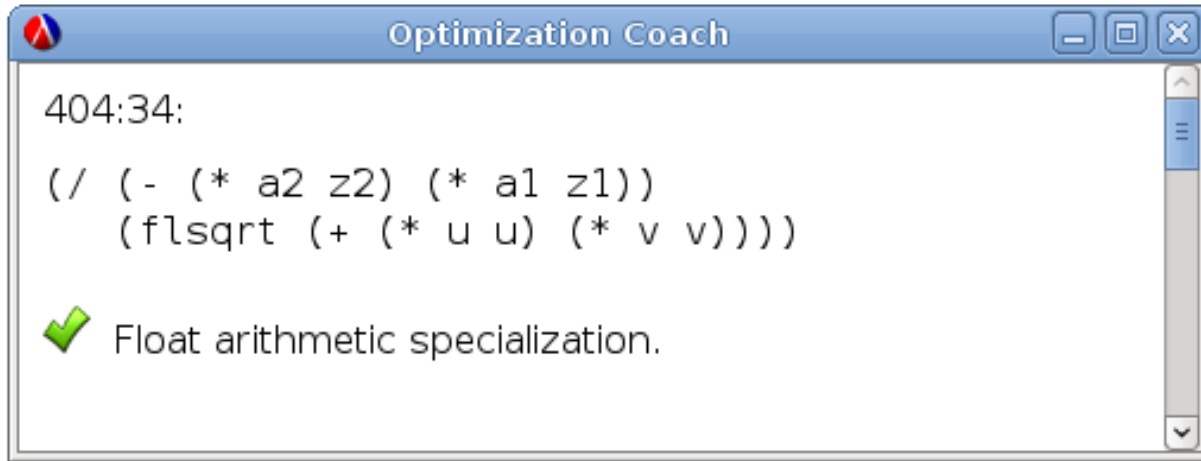
```

```

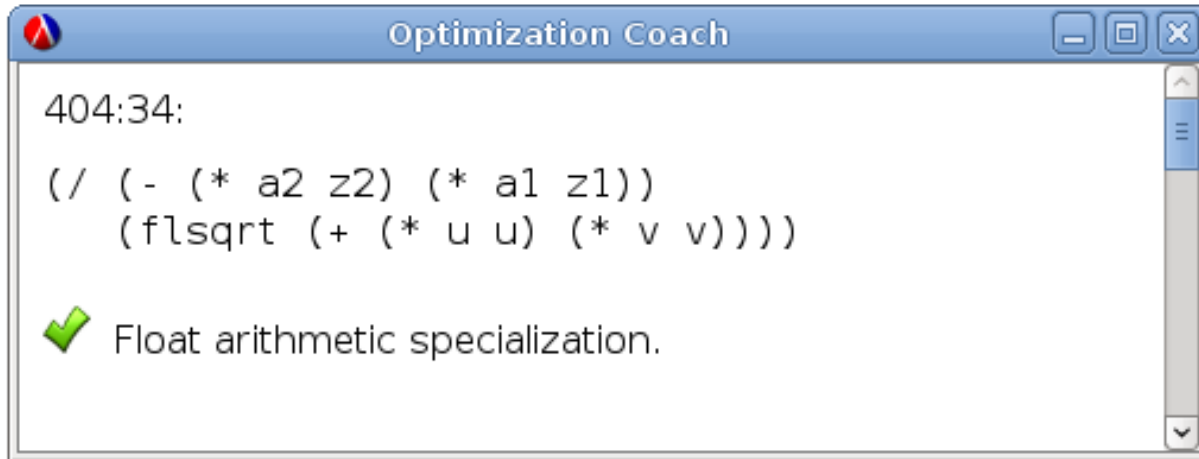
(define new-z-fm (flomap-blur z-fm o))
(define (deep-flomap-raise (deep-flomap (U Real flomap) -> deep-flomap))
  (define (deep-flomap-raise dfm z)
    (match-define (deep-flomap-argb argb-z-fm) dfm)
    (define alpha-fm (deep-flomap-alpha dfm))
    (define argb-fm (fx- z-fm (fx+ alpha-fm z))))
  (: deep-flomap-emboss (deep-flomap Real (U Real flomap) -> deep-flomap))
  (define (deep-flomap-emboss dfm xy-ant z-ant)
    (let ([o (/ xy-ant 3.0)])
      (define z-fm (flomap-normalize (deep-flomap-alpha dfm)))
      (define new-z-fm (fx+ (flomap-blur z-fm o) z-ant))
      (deep-flomap-raise dfm new-z-fm)))
  (: deep-flomap-bulge-helper (deep-flomap (flonum flonum -> flonum) -> deep-flomap))
  (define (deep-flomap-bulge-helper dfm f)
    (let ()
      (define-values (w h) (deep-flomap-size dfm))
      (define half-x-size (- (* 0.5 (fx->fl w) 0.5))
      (define half-y-size (- (* 0.5 (fx->fl h) 0.5)))
      (define z-fm
        (inline-build-flomap
         z w h
         (λ (x y z)
           (f (/ (fx->fl x) half-x-size) 1.0)
              (- (/ (fx->fl y) half-y-size) 1.0))))))
      (deep-flomap-raise dfm z-fm)))
  (: deep-flomap-bulge (deep-flomap (flonum flonum -> real) -> deep-flomap))
  (define (deep-flomap-bulge dfm f)
    (deep-flomap-bulge-helper dfm (λ (cx cy) (real->double-flonum (f cx cy)))))

```

Dialog between compilers and programmers



Dialog between compilers and programmers



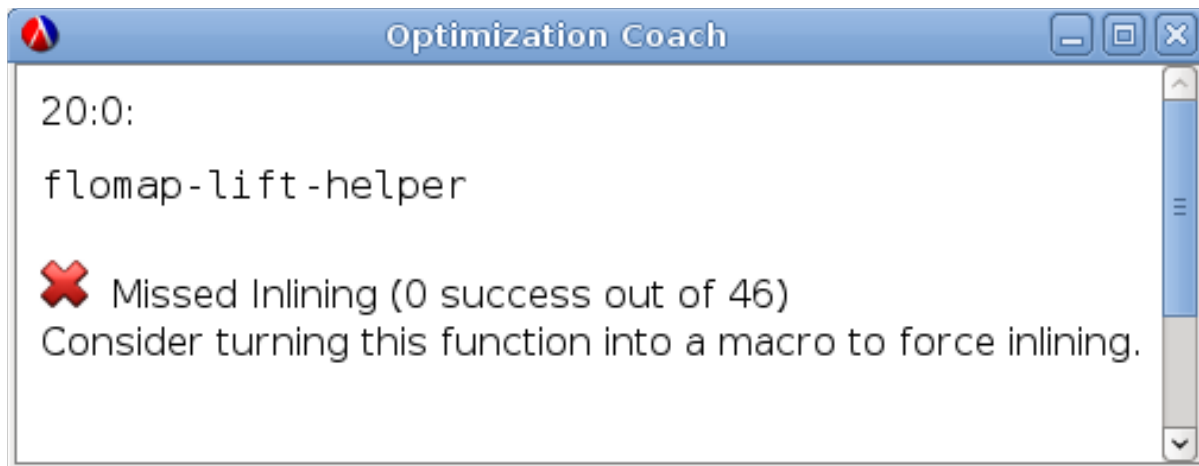
Optimization Coach

404:34:

```
(/ (- (* a2 z2) (* a1 z1))  
  (flsqrt (+ (* u u) (* v v))))
```

✓ Float arithmetic specialization.

Successes



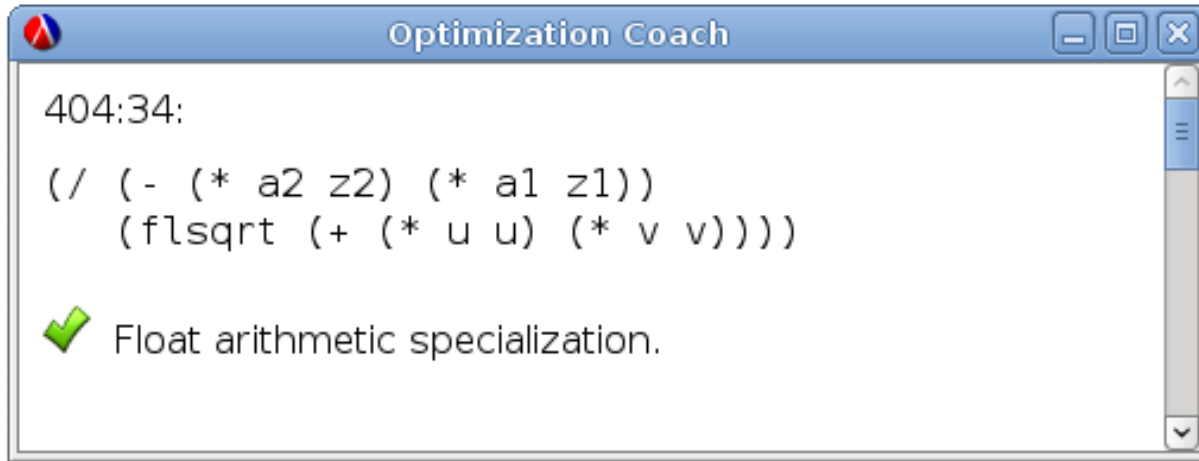
Optimization Coach

20:0:

flomap-lift-helper

✗ Missed Inlining (0 success out of 46)
Consider turning this function into a macro to force inlining.

Dialog between compilers and programmers

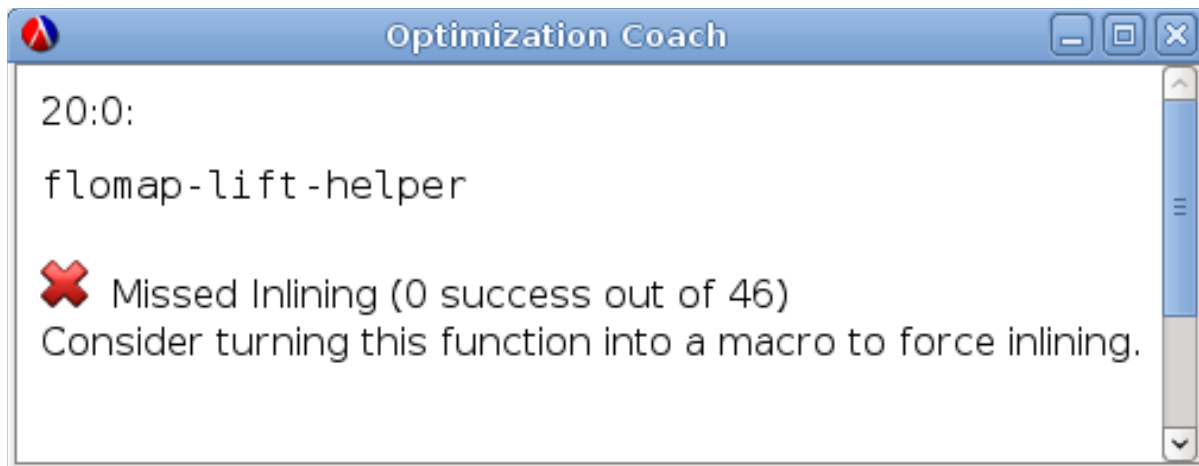


Optimization Coach

404:34:

```
(/ (- (* a2 z2) (* a1 z1))  
  (flsqrt (+ (* u u) (* v v))))
```

✓ Float arithmetic specialization.



Optimization Coach

20:0:

flomap-lift-helper

✗ Missed Inlining (0 success out of 46)
Consider turning this function into a macro to force inlining.

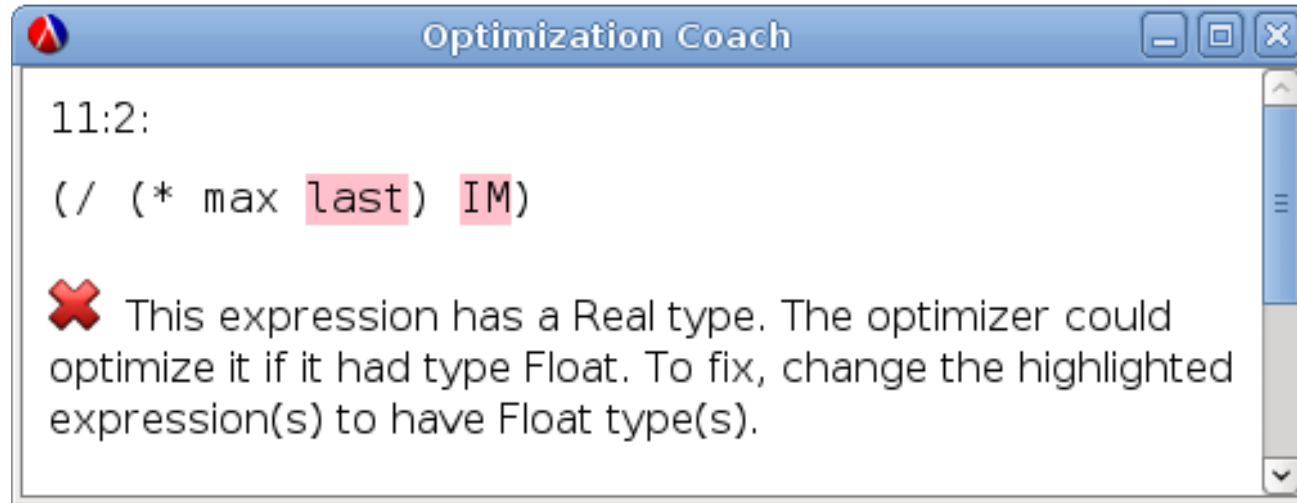
Successes

Near misses

+

Recommendations

Programmers can do more than compilers



Recommendations can **change semantics!**

`(/ 1 3)` → `1/3`

`(/ 1.0 3.0)` → `0.3333333333333333`

How does it work?

Overview

Compiler Instrumentation



Optimization Analysis



Recommendation Generation



Programmer Response

A day in the life of a near miss

```
#lang typed/racket/base
```

```
(define IM 139968)
```

```
(define IA 3877)
```

```
(define IC 29573)
```

```
(define last 42)
```

```
(define min 35.3)
```

```
(define max 156.8)
```

```
(define (gen-random)
```

```
  (set! last (modulo (+ (* last IA) IC) IM))
```

```
  (+ (/ (* (- max min) last) IM) min))
```


A day in the life of a near miss

```
#lang typed/racket/base
```

```
(define IM  
(define IA f1-  
(define IC (/ <Float> <Float>))  
  
(define last 42)  
(define min 35.3)  
(define max 156.8)  
(define (gen-random)  
  (set! last (modulo (+ (* last IA) IC) IM))  
  (+ (/ (* (- max min) last) IM) min))
```

A day in the life of a near miss

```
#lang typed/racket
```

```
(define IM
```

```
(define IA
```

```
(define IC
```

```
(define last
```

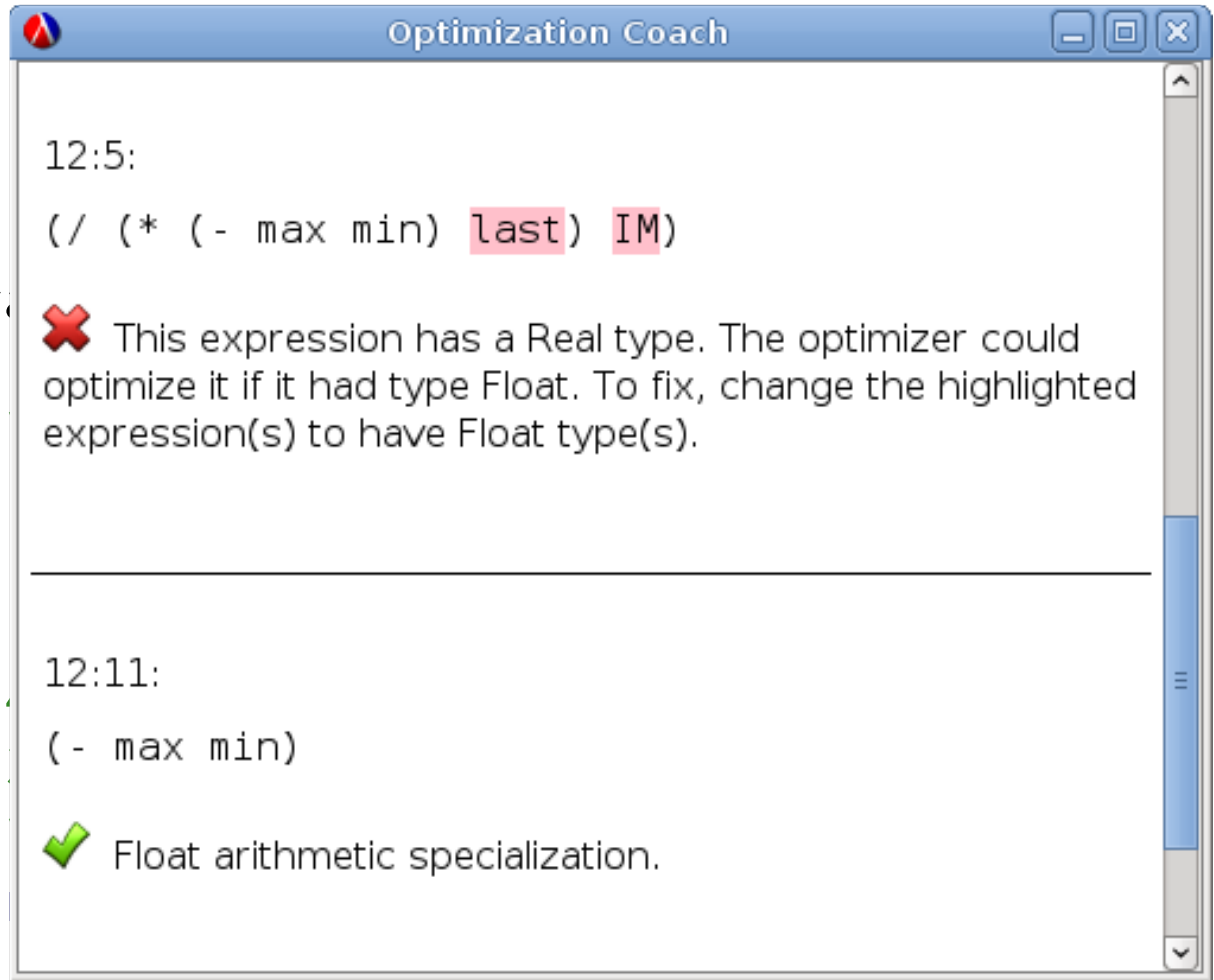
```
(define min
```

```
(define max
```

```
(define (gen-
```

```
(set! last
```

```
(+ (/ (* (- max min) last) IM) min))
```



Compiler Instrumentation

Float Float
`(- max min)`



`f1-`
`(- <Float> <Float>)`



`(f1- max min)`



TR opt: prng-example.rkt 12:11
`(- max min)`
Float Float
binary float subtraction

Compiler Instrumentation

Float Integer
`(* (- max min) last)`



`(* <Number> <Number>) ; no change`



`(* (- max min) last)`



TR opt failure: prng-example.rkt 12:8
`(* (- max min) last)`
Float Integer
generic multiplication

Optimization Analysis

Optimization proximity

Incomprehensible failure pruning

Irrelevant failure pruning

Harmless failure pruning

Irritant analysis

Causality merging

Locality merging

Optimization Analysis

Optimization proximity

Incomprehensible failure pruning

Irrelevant failure pruning

Harmless failure pruning

Irritant analysis

Causality merging

Locality merging

Optimization Analysis

Optimization proximity

Float Integer
`(* (- max min) last)`

Float
Float ~~Integer~~
Irritant $\Delta = 1$

Optimization Analysis

Optimization proximity

Near miss, report

Float
Float ~~Integer~~
Irritant $\Delta = 1$

Optimization Analysis

Optimization proximity

Integer

Integer

(* last IA)

Float

Float

~~Integer~~

~~Integer~~

Irritant

Irritant

$$\Delta = 2$$

Optimization Analysis

Optimization proximity

Too far, don't report

Float	Float	$\Delta = 2$
Integer	Integer	
Irritant	Irritant	

Recommendation Generation

Float Integer

```
(* (- max min) last)
```

Float
~~Integer~~
Irritant

12:5:

```
(/ (* (- max min) last) IM)
```


✘ This expression has a Real type. The optimizer could optimize it if it had type Float. To fix, change the highlighted expression(s) to have Float type(s).

Programmer Response

```
(->f1 last) (->f1 IM)
(+ (/ (* (- max min) last) IM) min)
```

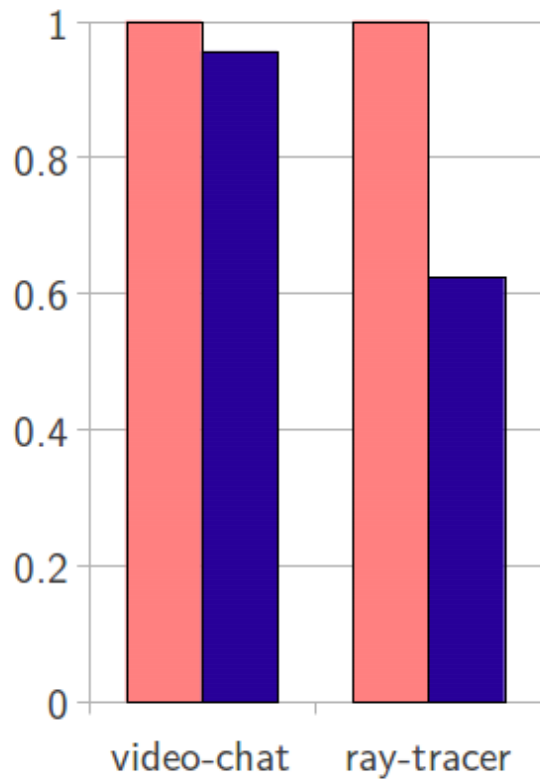
12:5:

```
(/ (* (- max min) last) IM)
```

 This expression has a Real type. The optimizer could optimize it if it had type Float. To fix, change the highlighted expression(s) to have Float type(s).

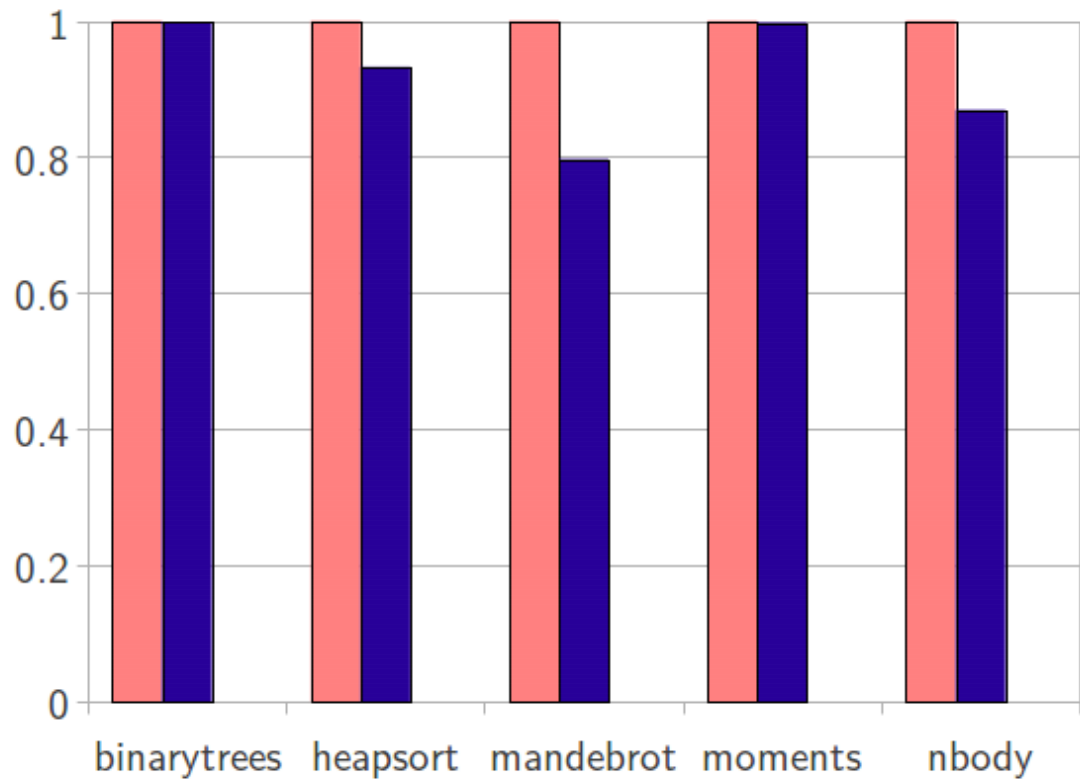
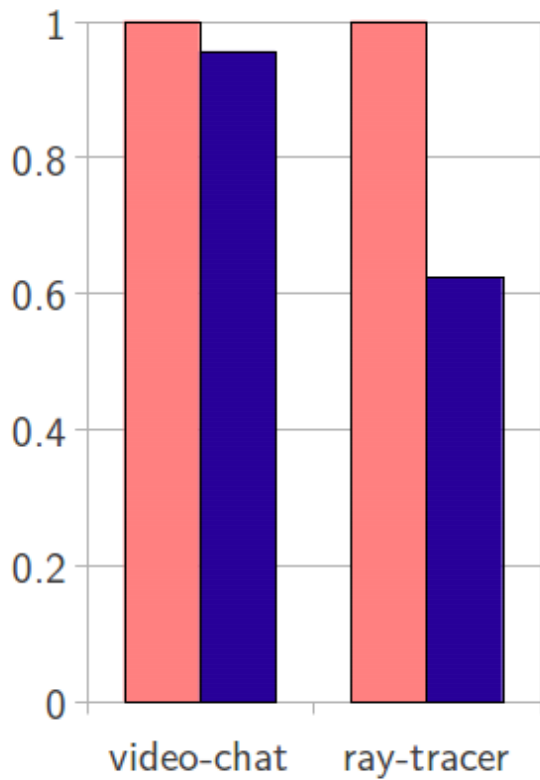
How well does it work?

- Baseline: Non-optimized
- Coached: Followed recommendations (Minutes of work)



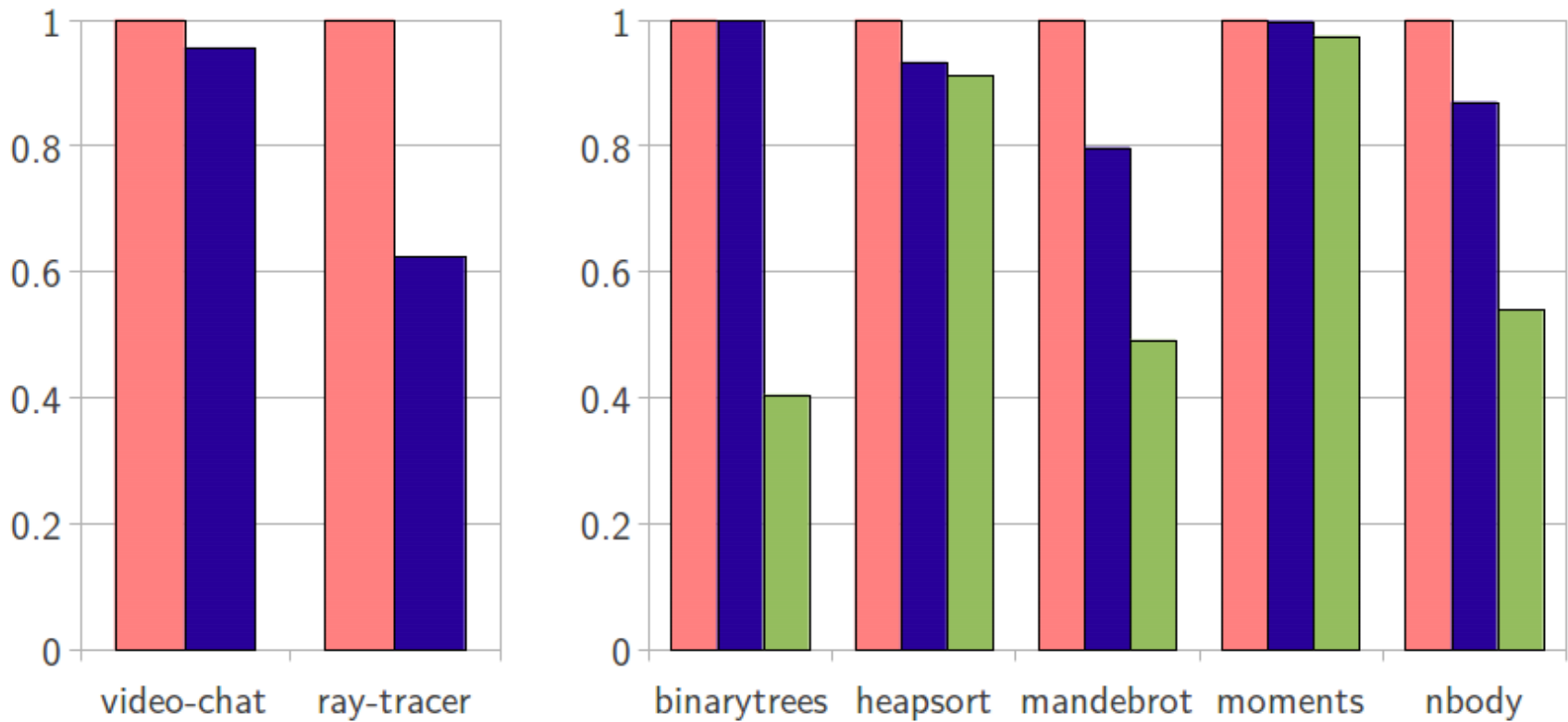
Lower is better

■ Baseline: Non-optimized
■ Coached: Followed recommendations (Minutes of work)



Lower is better

- Baseline: Non-optimized
- Coached: Followed recommendations (Minutes of work)
- Gold standard: Hand-optimized by experts (Days of work)



Lower is better



The take-away

Key idea: The compiler talks back

General optimization analysis techniques

+ *Optimization-specific heuristics*

Targeted recommendations



The take-away

Key idea: The compiler talks back

General optimization analysis techniques

+ *Optimization-specific heuristics*

Targeted recommendations

racket-lang.org