

Optimization Coaching

Vincent St-Amour

Sam Tobin-Hochstadt

Matthias Felleisen

PLT

Harvard - October 31st, 2012

```

enum typed/racket/base
(require racket/float)

(except-in racket/flonum fl->fx fx->fl)

racket/match racket/math

"flonum.rkt"
"flonop-struct.rkt"
"flonop-stats.rkt")

(provide (flonop-lift flonop-lift2 flonop-lift-helper flonop-lift-helper2
  fmax fmaxs fmsqr fmsin facos fatan flnlog fmaxp fmsqrt fmaxsin fmaxcos fmaxtan
  fround fxfloor fceiling ftruncate fzazero
  fx- fa- for fx/ fmin fmax
  flonop-normalize flonop-multiply-alpha flonop-divide-alpha)

; =====
; Unary
(: flonop-lift-helper : (Float -> Float) -> (flonop -> flonop))
(define (flonop-lift-helper f)
  (λ: (fml : flonop)
    (match-define (flonop v c w h) fml)
    (flonop (inline-build-fvector (* c w h) (λ (i) (f (unsafe-flvector-ref v i))))
      c w h)))
(: flonop-lift (flonum -> Real) -> (flonop -> flonop))
(define (flonop-lift op)
  (flonop-lift-helper (λ (x) (real->double-flonum (op x)))))
(define fmax (flonop-lift-helper -))
(define fmaxs (flonop-lift-helper abs))
(define fmsqr (flonop-lift-helper sq))
(define fmsin (flonop-lift-helper sin))
(define facos (flonop-lift-helper cos))
(define fatan (flonop-lift-helper tan))
(define flnlog (flonop-lift-helper fllog))
(define fmaxp (flonop-lift-helper exp))
(define fmsqrt (flonop-lift-helper fsqrt))
(define fmaxsin (flonop-lift-helper asin))
(define fmaxcos (flonop-lift-helper acos))
(define fmaxtan (flonop-lift-helper atan))
(define fround (flonop-lift-helper round))
(define fxfloor (flonop-lift-helper floor))
(define fceiling (flonop-lift-helper ceiling))
(define ftruncate (flonop-lift-helper truncate))
(define fzazero (flonop-lift-helper (λ (x) (if (x = . 0.0) 1.0 0.0))))
; =====
; Binary
(: flonop-lift-helper2 : Symbol (Float Float -> Float) -> ((U Real flonop) (U Real flonop) -> flonop))
(define (flonop-lift-helper2 name f)
  (let: ()
    (λ: (fml1 : (U Real flonop)) (fml2 : (U Real flonop))
      (cond
        [(and (real? fml1) (real? fml2))
          (error name "expected at least one flonop argument; given ~e and ~e" fml1 fml2)]
        [(real? fml1) (let ([fml1 (real->double-flonum fml1)])
          ((flonop-lift-helper (λ (v) (f fml1 v))) fml2))]
        [(real? fml2) (let ([fml2 (real->double-flonum fml2)])
          ((flonop-lift-helper (λ (v) (f v fml2))) fml1))]
        [else
          (match-define (flonop v1 c1 w1 h1) fml1)
          (match-define (flonop v2 c2 w2 h2) fml2)
          (cond
            [(not (and (= w1 w2) (= h1 h2)))
              (error name "expected same-size flonops; given sizes ~e-e and ~e-e w h w2 h2)]
            [(= c1 c2) (define n (* c1 w1 h1))
              (define res-vs (make-fvector n))
              (flonop (inline-build-fvector n (λ (i) (f (unsafe-flvector-ref v1 i)
                (unsafe-flvector-ref v2 i))))
                c1 w1 h1)]
            [(= c1 1) (inline-build-flonop
              c2 w h
              (λ (k x y i) (f (unsafe-flvector-ref v1 (coords->index 1 w 0 x y))
                (unsafe-flvector-ref v2 i))))]
            [(= c2 1) (inline-build-flonop
              c1 w h
              (λ (k x y i) (f (unsafe-flvector-ref v1 i)
                (unsafe-flvector-ref v2 (coords->index 1 w 0 x y))))]
            [else
              (error name (string-append "expected flonops with the same number of components, "
                "or a flonop with 1 component and any same-size flonop; "
                "given flonops with ~e and ~e components")
                c1 c2)]))])])])
(: flonop-lift2 (Symbol) (flonum flonum -> Real) -> ((U Real flonop) (U Real flonop) -> flonop))
(define (flonop-lift2 name f)
  (flonop-lift-helper2 name (λ (x y) (real->double-flonum (f x y))))
(define fm+ (flonop-lift-helper2 'f+))
(define fm- (flonop-lift-helper2 'f-))
(define fm (flonop-lift-helper2 'f*))
(define fm/ (flonop-lift-helper2 'f/))
(define fmin (flonop-lift-helper2 'fmin min))
(define fmax (flonop-lift-helper2 'fmax max))
(: flonop-normalize (flonop -> flonop))
(define (flonop-normalize fm)
  (define-values (v-min v-max) (flonop-extreme-values fm))
  (define v-size (- v-max v-min))
  (let ([fm (f- fm v-min)])
    (f- (if (v-size = . 0.0) fm (fm/ fm v-size)))
    fm))
(define fndiv/zero
  (flonop-lift-helper2 'fndiv/zero (λ (x y) (if (y = . 0.0) 0.0 (/ x y)))))
(: flonop-divide-alpha (flonop -> flonop))
(define (flonop-divide-alpha fm)
  (match-define (flonop _c w h) fm)
  (cond [(= (. <= . 1) fm)
    [else
      (define alpha-fm (flonop-ref-component fm 0))
      (flonop-append-components alpha-fm (fndiv/zero (flonop-drop-components fm 1) alpha-fm))]]])

```

```
(: deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
(define (deep-flonap-bulge dfm f)
  (deep-flonap-bulge-helper dfm (λ (cx cy) (real->double-flonum (f cx cy)))))
```



```

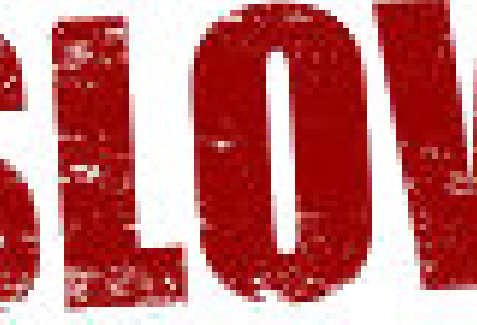
#ang typed/racket/base
(require racket/flonum

(racket-match racket/math

    "flonum.rkt"
    "flomap-struct.rkt"
    "flomap-stats.rkt")

(provide flomap-lift flomap-lift2 flomap-lift-helper flomap-lift-helper2
fmapy fnabs fmsqr fmsin fmcos ftan fnlog fmxp fsqrt fmsin fmacos fnsatan
fsround fmfloor facelling fnturccate fzero
fm+ fm- fm* fm/ fmin fmax
flomap-normalize flomap-multiply-alpha flomap-divide-alpha)

; =====
; Unary
(: flomap-lift-helper : (Float -> Float) -> (Flomap -> Flomap))
(define (flomap-lift-helper f)
  (λ (fmap : (fn : Flomap))
    (match-define (flomap vs c w h) fmap)
    (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 fmapy (flomap-lift-helper *))
(define fnabs (flomap-lift-helper abs))
(define fmsqr (flomap-lift-helper sqr))
(define fmsin (flomap-lift-helper sin))
(define fmcos (flomap-lift-helper cos))
(define ftan (flomap-lift-helper tan))
(define fmlog (flomap-lift-helper log))
(define fmxp (flomap-lift-helper exp))
(define fsqrt (flomap-lift-helper sqrt))
(define fmsin (flomap-lift-helper asin))
(define fmacos (flomap-lift-helper acos))
(define fnsatan (flomap-lift-helper atan))
(define fsround (flomap-lift-helper round))
(define fmfloor (flomap-lift-helper floor))
(define facelling (flomap-lift-helper ceiling))



(unsafe-flvector-ref vs2 i))))

((= c2 1) (inline-build-flomap
  c1 w h
  (λ (k x y i) (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"))
c1 c2))))))
(: 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 fm+ (flomap-lift-helper2 '+))
  (define fm- (flomap-lift-helper2 '-))
  (define fm* (flomap-lift-helper2 '*'))
  (define fm/ (flomap-lift-helper2 '/'))
  (define fmsin (flomap-lift-helper2 'sin))
  (define fmaxx (flomap-lift-helper2 'max))
  (: flomap-normalize (flomap -> flomap))
  (define (values (v-min v-max) (flomap-extreme-values fm))
    (define v-size (- v-max v-min))
    (let ([fm (flomap-lift-helper2 'div v-size)])
      (fm (if (= v-size . 0.0) fm (fm/fm v-size))))))
  (define fmdiv/zero
    (flomap-append-components alpha-fm (fmdiv/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 _ c w h) fm)
    (cond [(= (. <=. 1) fm)
      [else
        (define alpha-fm (flomap-drop-components fm 1) alpha-fm)]])
    (flomap-append-components alpha-fm (fmdiv/zero (flomap-drop-components fm 1) alpha-fm)))

```

```

(lang typed/racket/base
(require racket/flonum
 (except-in racket/flonum fx->fl fx->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-rbg
 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-!t-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-vl-append deep-flonap-vr-append deep-flonap-vr-append
 deep-flonap-hl-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 ~x~w and ~x~h 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)
 ~-fn (make-flonap 1 w h))
  h (deep-flonap -> Nonnegative-Fixnum))
  p-width dfm)
  -width (deep-flonap-argb dfm))
  nonnegative-fixnum?))

  ht (deep-flonap -> Nonnegative-Fixnum))
  p-height dfm)
  -height (deep-flonap-argb dfm))
  nonnegative-fixnum?))

  n (deep-flonap -> Flonum))
  p-z-min dfm)
  (deep-flonap-z dfm))
  x (deep-flonap -> Flonum))
  p-z-max dfm)
  (deep-flonap-z dfm))
  (deep-flonap -> (values Nonnegative-Fixnum Nonnegative-Fixnum)))
  p-size dfm)
  map-width dfm) (deep-flonap-height dfm))
  a (deep-flonap -> flonap))
  p-alpha dfm)
  nest (deep-flonap-argb dfm) 0))
  (deep-flonap -> flonap))
  p-rgb dfm)
  onents (deep-flonap-argb dfm) 1])

=====

(e-z (deep-flonap (U Real flonap) -> deep-flonap))
(define (deep-flonap-scale-z dfm z)
 (match-define (deep-flonap argb-fn z-fn) dfm)
 (deep-flonap argb-fn (fn* z-fn z)))
(: deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
(define (deep-flonap-smooth-z dfm α)
 (let ([o (exact->inexact 0)])
 (match-define (deep-flonap argb-fn z-fn) dfm)
 (define new-z-fn (flonap-blur z-fn α))
 (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 dfm z)
 (match-define (deep-flonap argb-fn z-fn) dfm)
 (define alpha-fn (deep-flonap-alpha-fn z))
 (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 dfm xy-ant z-ant)
 (let ([o (/ xy-ant 3.0)])
 (define z-fn (flonap-normalize (deep-flonap-alpha dfm)))
 (define new-z-fn (fn* (flonap-raise z-fn o) z-ant))
 (deep-flonap-raise dfm new-z-fn)))
(: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> Flonum) -> deep-flonap))
(define (deep-flonap-bulge-helper dfm f)
 (let ()
 (define-values (w h) (deep-flonap-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-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 dfm z-fn)))
(: deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
(define (deep-flonap-bulge dfm f)
 (deep-flonap-bulge-helper dfm (λ (x cy) (real->double-flonum (f x cy)))))

```

building results			
total open data observed: 10000 (out of 10000)			
number of samples taken: 146 (out of 146)			
Building Functions used: 146 (in and 146 out of 146)			
(1)	total	total	100.0%
(2)	total	total	100.0%
(3)	total	total	100.0%
(4)	total	total	100.0%
(5)	total	total	100.0%
(6)	total	total	100.0%
(7)	total	total	100.0%
(8)	total	total	100.0%
(9)	total	total	100.0%
(10)	total	total	100.0%
(11)	total	total	100.0%
(12)	total	total	100.0%
(13)	total	total	100.0%
(14)	total	total	100.0%
(15)	total	total	100.0%
(16)	total	total	100.0%
(17)	total	total	100.0%
(18)	total	total	100.0%
(19)	total	total	100.0%
(20)	total	total	100.0%
(21)	total	total	100.0%
(22)	total	total	100.0%
(23)	total	total	100.0%
(24)	total	total	100.0%
(25)	total	total	100.0%
(26)	total	total	100.0%
(27)	total	total	100.0%
(28)	total	total	100.0%
(29)	total	total	100.0%
(30)	total	total	100.0%
(31)	total	total	100.0%
(32)	total	total	100.0%
(33)	total	total	100.0%
(34)	total	total	100.0%
(35)	total	total	100.0%
(36)	total	total	100.0%
(37)	total	total	100.0%
(38)	total	total	100.0%
(39)	total	total	100.0%
(40)	total	total	100.0%
(41)	total	total	100.0%
(42)	total	total	100.0%
(43)	total	total	100.0%
(44)	total	total	100.0%
(45)	total	total	100.0%
(46)	total	total	100.0%
(47)	total	total	100.0%
(48)	total	total	100.0%
(49)	total	total	100.0%
(50)	total	total	100.0%
(51)	total	total	100.0%
(52)	total	total	100.0%
(53)	total	total	100.0%
(54)	total	total	100.0%
(55)	total	total	100.0%
(56)	total	total	100.0%
(57)	total	total	100.0%
(58)	total	total	100.0%
(59)	total	total	100.0%
(60)	total	total	100.0%
(61)	total	total	100.0%
(62)	total	total	100.0%
(63)	total	total	100.0%
(64)	total	total	100.0%
(65)	total	total	100.0%
(66)	total	total	100.0%
(67)	total	total	100.0%
(68)	total	total	100.0%
(69)	total	total	100.0%
(70)	total	total	100.0%
(71)	total	total	100.0%
(72)	total	total	100.0%
(73)	total	total	100.0%
(74)	total	total	100.0%
(75)	total	total	100.0%
(76)	total	total	100.0%
(77)	total	total	100.0%
(78)	total	total	100.0%
(79)	total	total	100.0%
(80)	total	total	100.0%
(81)	total	total	100.0%
(82)	total	total	100.0%
(83)	total	total	100.0%
(84)	total	total	100.0%
(85)	total	total	100.0%
(86)	total	total	100.0%
(87)	total	total	100.0%
(88)	total	total	100.0%
(89)	total	total	100.0%
(90)	total	total	100.0%
(91)	total	total	100.0%
(92)	total	total	100.0%
(93)	total	total	100.0%
(94)	total	total	100.0%
(95)	total	total	100.0%
(96)	total	total	100.0%
(97)	total	total	100.0%
(98)	total	total	100.0%
(99)	total	total	100.0%
(1			

```

(unsafe-flvector-ref vs2 i))))

[ (= c2 1) (inline-build-flomap
  c1 w h
  (λ (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 (Flomum Flomum) -> Real) -> ((U Real flomap) (U Real flomap) -> flomap))
(define (flomap-lift2 name fn)
  (flomap-lift-helper2 name (λ (x y) (real->double (flomun (f x y)))))
(define fn= (flomap-lift-helper2 'fn =))
(define fn- (flomap-lift-helper2 'fn -))
(define fn+ (flomap-lift-helper2 'fn +))
(define fn/ (flomap-lift-helper2 'fn /))
(define fmin (flomap-lift-helper2 'fmin min))
(define fmax (flomap-lift-helper2 'fmax max))
(: flomap-normalize (flomap -> flomap))
(define (flomap-normalize fn)
  (define-values (v-min v-max) (flomap-extreme-values fn))
  (define v-size (- v-max v-min))
  (let* ((fn= (fn- fn v-min))
    (fn+ (fn (if (v-size = . 0.0) fn (fn/ fn v-size))))
    fn)
  (define fndiv/zero
    (flomap-lift-helper2 'fndiv/zero (λ (x y) (if (y = . 0.0) 0.0 (/ x y))))
  (: flomap-divide-alpha (flomap -> flomap))
  (define (flomap-divide-alpha fn)
    (match-define (flomap _ c w h) fn)
    (cond [(c = - . 1) fn]
      [else
       (define alpha-fn (flomap-ref-component fn 0))
       (flomap-append-components alpha-fn (fndiv/zero (flomap-drop-components fn 1) alpha-fn))]))

```

```

(e-z (deep-flonmap (U Real flonmap) -> deep-flonmap))

(define (deep-flonmap-scale-z dfm z)
  (match-define (deep-flonmap arg-b fm z-fm) dfm)
  (deep-flonmap arg-b-fm (fa* z-fm z)))

(: deep-flonmap-smooth-z (deep-flonmap Real -> deep-flonmap))
(define (deep-flonmap-smooth-z dfm)
  (let ([o (exact->inexact o)])
    (match-define (deep-flonmap arg-b fm z-fm) dfm)
    (define new-z-fm (deep-flonmap-blur z-fm o))
    (deep-flonmap arg-b-fm new-z-fm)))

; deep-flonmap-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-flonmap-raise (deep-flonmap (U Real flonmap) -> deep-flonmap))
(define (deep-flonmap-raise dfm z)
  (match-define (deep-flonmap arg-b fm z-fm) dfm)
  (define alpha-fm (deep-flonmap-alpha dfm))
  (deep-flonmap arg-b-fm (fa+ z-fm (fa* alpha-fm z)))))

(: deep-flonmap-emboss (deep-flonmap Real (U Real flonmap) -> deep-flonmap))
(define (deep-flonmap-emboss dfm xy-ant z-ant)
  (let ([o (/ xy-ant 3.0)])
    (define z-fm (flonmap-normalize (deep-flonmap-alpha dfm)))
    (define new-z-fm (fa* (flonmap-blur z-fm o) z-ant))
    (deep-flonmap arg-b-fm (deep-new-z-fm)))

; deep-flonmap-bulge-helper (deep-flonmap (Flonmap Flonmap -> Flonmap) -> deep-flonmap))
(define (deep-flonmap-bulge-helper dfm f)
  (let ()
    (define-values (w h) (deep-flonmap-size dfm))
    (define half-x-size (- (* 0.5 (fx<=) w) 0.5))
    (define half-y-size (- (* 0.5 (fx<=) h) 0.5))
    (define z-fm
      (inline-build-flonmap
        1 w h
        (lambda (x y z)
          (f (- (/ (fx<=) x) half-x-size) 1.0)
            (- (/ (fx<=) y) half-y-size) 1.0))))
    (deep-flonmap-raise dfm z-fm)))

(: deep-flonmap-bulge (deep-flonmap (Flonmap Flonmap -> Real) -> deep-flonmap))
(define (deep-flonmap-bulge dfm f)
  (deep-flonmap-bulge-helper dfm (lambda (x cy cy) (real->double-flonmap (f cx cy)))))

```

2:35. This is

```
(: 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 arg-hf z-fn f) dfm)
    (define new-z-fn (flomap-blur z-fn o))
    (deep-flomap arg-hf new-z-fn f)))

; 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 dz)
  (match-define (deep-flomap arg-hf z-fn f) dfm)
  (define alpha-fn (deep-flomap-alpha dfm))
  (define flomap-arg-hf (fn (z x-fn) (fn* (alpha-fn z))))
  (: deep-flomap-emboss (deep-flomap Real (U Real flomap) => deep-flomap))
  (define (deep-flomap-emboss dfr xy-ant z-ant z)
    (let ([o (/ xy-ant 3.0)])
      (define z-fn (flomap-normalize (deep-flomap-alpha dfm)))
      (define new-z-fn (fn* (flomap-blur z-fn o) z-ant))
      (deep-flomap-raise dfm new-z-fn f)))
  (: deep-flomap-helpz-helper (deep-flomap (Flonum Flonum => Flonum) => deep-flomap))
  (define (deep-flomap-helpz-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-fn
            (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-fn f)))))
  (: deep-flomap-hlpr (deep-flomap (Flonum Flonum => Real) => deep-flomap))
  (define (deep-flomap-hlpr dfm f)
    (deep-flomap-helpz-helper dfm f))
    (define (deep-flomap-helpz-helper dfm f)
      (let ([x cy cy] (real-double-flonum (f cx cy))))
```



```

#lang typed/racket/base
(require racket/match racket/math racket/
  (except-in racket/fixnum fl->fx
    "flonum.rkt"
    "flonap-struct.rkt"))
(provide flonap-flip-horizontal flonap-fl
  flonap-cw-rotate
  (struct-out invertible-2d-func
    transform-compose
    flonap-transform
    (: flonap-flip-horizontal
      (define (flonap-flip-hor
        (match-define (flonap
          (define w-1 (fx- w 1))
          (inline-build-flonap
            (define (flonap-flip-ver
              (match-define (flonap
                (define h-1 (fx- h 1))
                (inline-build-flonap
                  (define (flonap-transpo
                    (match-define (flonap
                      (inline-build-flonap
                        (define (flonap-cw-rotat
                          (match-define (flonap
                            (define h-1 (fx- h 1))
                            (inline-build-flonap
                              (define (flonap-ccw-rotat
                                (match-define (flonap
                                  (define w-1 (fx- w 1))
                                  (inline-build-flonap
                                    (struct: invertible-2d-func
                                      (define-type Flonap-Trans
                                        (: transform-compose (Fl
                                          (define ((transform-comp
                                            (match-define (invert
                                              (match-define (invert
                                                (invertible-2d-functio

```

```

(: flonap-transform (case
  (define flonap-transform
    (case-lambda
      ([fn t]
        (match-define (flonap
          (match-define (invertible-2d-func
            (define x-min +inf
              (define x-max -inf
                (define y-min +inf
                  (define y-max -inf
                    (let: y-loop : Void
                      (when (y - fr- .
                        (let: x-loop :
                          (cond [(x - .
                            (define
                              (when
                                (when
                                  (when
                                    (when
                                      (when
                                        (when
                                          (when
                                            (when
                                              (when
                                                (when
                                                  (when
                                                    (when
                                                      (when
                                                        (when

```

20 hours later...

```

()
#f)
#6=(module-rename
  1
  normal
  4026
  (#s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 0 1 () #f ())
    #s((all-from-module zo 0) #<module-path-index> 1 0 () #f ())
  #7=#s((all-from-module zo 0)
    #<module-path-index>
    1
    0
    ()
    #f
    ()))

```

```

.
#s((phased-module-binding module-binding 0 zo 0)
  #<module-path-index>
  1
  parameter-name->arg-name
  #8#
  parameter-name->arg-name))
(<lifted>
.
#s((phased-module-binding module-binding 0 zo 0)
  #<module-path-index>
  1
  <lifted>
  #8#
  <lifted>))
(argument-spec
.
#s((phased-module-binding module-binding 0 zo 0)
  #<module-path-index>
  1

```

```

piled' -exec rm
l images
compiled' -exec rm
l images/private
l images
. This is

```

```

an invariant:
0.0
(flonap))
deep-flonap))
)
(flonum) -> deep-flonap))
> deep-flonap))
onum (f cx cy))))

```



```

(flang Typed/racket/base
(require racket/flonum
 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-vl-append deep-flonap-vc-append deep-flonap-vr-append
 deep-flonap-hl-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 -ex-e and -ex-e w 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)

```

```
(define (flomap-lift-helper f) ...)
```

```
(define (deep-flonmap-scale-z dfm z)
  (match-define (deep-flonmap-arg-fn f-z-fn) dfm)
  (deep-flonmap-arg-fn (fn* z-fm z)))
(: deep-flonmap-smooth-z (deep-flonmap Real -> deep-flonmap))
(define (deep-flonmap-smooth-z dfm o)
  (let ([o] (exact->inexact o)])
    (match-define (deep-flonmap-arg-fn f-z-fn) dfm)
    (define new-z-fm (flonmap-blur z-fm o))
    (deep-flonmap-arg-fn f-new-z-fm)))

; deep-flonmap-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-flonmap-raise (deep-flonmap (U Real flonmap) -> deep-flonmap))
(define (deep-flonmap-raise dfm z)
  (match-define (deep-flonmap-arg-fn f-z-fn) dfm)
  (define alpha-fn (deep-flonmap-alpha dfm))
  (deep-flonmap-arg-fn (fn* z-fm (fn* [alpha-fn] alpha-fn z))))
(: deep-flonmap-emboss (deep-flonmap (U Real flonmap) -> deep-flonmap))
(define (deep-flonmap-emboss dfm xy-ant z-ant)
  (let ([o] (xy-ant 3.0)])
    (define z-fm (flonmap-normalize (deep-flonmap-alpha dfm)))
    (define new-z-fm (fn* (flonmap-blur z-fm o) z-ant))
    (deep-flonmap-raise dfm new-z-fm)))

; deep-flonmap-bulge-helper (deep-flonmap (Flonmap Flonmap -> Flonmap) -> deep-flonmap))
(define (deep-flonmap-bulge-helper dfm f)
  (let ()
    (define-values (w h) (deep-flonmap-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-flonmap
        1 w h
        (lambda (_ x y)
          ((f (- (/ (fx->fl x) half-x-size) 1.0)
              (- (/ (fx->fl y) half-y-size) 1.0)))))))
    (deep-flonmap-raise dfm z-fm)))

; deep-flonmap-double-flonmap (deep-flonmap (Flonmap -> Real) -> deep-flonmap)
(define (deep-flonmap-double-flonmap dfm)
  (deep-flonmap-bulge-helper dfm (lambda (x cy) (real->double-flonmap (if (cx cy) 1))))
```

```
#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-cw-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 fm)
  (match-define (flonap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flonap c w h (λ (k x y i)
    (inline-build-flonap c w h (λ (k x y i)
      (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))))
  (define (flonap-flip-vertical fm)
    (match-define (flonap vs c w h) fm)
    (define h-1 (fx- h 1))
    (inline-build-flonap c w h (λ (k x y i)
      (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y) i))))))
  (define (flonap-transpose fm)
    (match-define (flonap vs c w h) fm)
    (inline-build-flonap c h w (λ (k x y i)
      (unsafe-flvector-ref vs (coords->index c w k x y i))))))
  (define (flonap-cw-rotate fm)
    (match-define (flonap vs c w h) fm)
    (define h-1 (fx- h 1))
    (inline-build-flonap c h w (λ (k x y i)
      (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) x i))))))
  (define (flonap-cw-rotate fm)
    (match-define (flonap vs c w h) fm)
    (define w-1 (fx- w 1))
    (inline-build-flonap c h w (λ (k x y i)
      (unsafe-flvector-ref vs (coords->index c w k x (fx- w-1 x) i))))))
  (struct: invertible-2d-function [(f : (Flonum Flonum -> (values Flonum Flonum)))]
    (transform-compose flonap-flip-horizontal flonap-flip-vertical flonap-transpose)))
```

```
#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
         flonap flnabs flnsqr flnsin flncos flntan flnlog flnexp flnsqrt flnsin flncos flntan
         flround flfloor flceiling fltruncate flzero
         fm- fm- fm- fm/ flmin flmax
         flonap-normalize flonap-multiply-alpha flonap-divide-alpha)
; =====
; Unary
(: flonap-lift-helper : (Float -> Float) -> (flonap -> flonap))
(define (flonap-lift-helper f)
  (λ (f : (fl : flonap))
    (match-define (flonap vs c w h) fm)
    (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 flnabs (flonap-lift-helper abs))
(define flnsqr (flonap-lift-helper sqr))
(define flnsin (flonap-lift-helper sin))
(define flncos (flonap-lift-helper cos))
(define flntan (flonap-lift-helper tan))
(define flnlog (flonap-lift-helper fllog))
(define flnexp (flonap-lift-helper exp))
(define flnsqrt (flonap-lift-helper flsqrt))
(define flnsin (flonap-lift-helper asin))
(define flncos (flonap-lift-helper acos))
(define flntan (flonap-lift-helper atan))
(define flround (flonap-lift-helper round))
(define flfloor (flonap-lift-helper floor))
(define flceiling (flonap-lift-helper ceiling))
(define fltruncate (flonap-lift-helper truncate))
(define flzero (flonap-lift-helper zero))
```

```
#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-vl-append deep-flonap-vr-append deep-flonap-vr-append
         deep-flonap-hr-append deep-flonap-hc-append deep-flonap-hb-append)
(struct: deep-flonap ([argb : flonap] [z : flonap])
  #:transparent
  #:guard
  (λ (argb-fm z-fm name)
    (match-define (flonap _ 4 w h) argb-fm)
    (match-define (flonap _ 1 zw zh) z-fm)
    (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-fm z-fm)))
(: flonap->deep-flonap (flonap -> deep-flonap))
(define (flonap->deep-flonap flonap)
  (deep-flonap argb-fm (fm- z-fm z)))
```

There must be a better way.

```
[else
  (y-loop (fx+ y 1)))]))
(flonap-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 (flonap 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-flonap
      c new-w new-h
      (λ (k x y i)
        (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
          (+ (fx->fl y) y-offset))))
      (flonap-bilinear-ref fm k old-x old-y))))))
```

```
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 fm+ (flonap-lift-helper2 'fm+ +))
(define fm- (flonap-lift-helper2 'fm- -))
(define fm* (flonap-lift-helper2 'fm* *))
(define fm/ (flonap-lift-helper2 'fm/ /))
(define flmin (flonap-lift-helper2 'flmin min))
(define flmax (flonap-lift-helper2 'flmax max))
(: flonap-normalize (flonap -> flonap))
(define (flonap-normalize fm)
  (define-values (v-min v-max) (flonap-extreme-values fm))
  (define v-size (- v-max v-min))
  (let* ([fm (fm- fm v-min)]
        [fm (if (v-size = 0 0.0) fm (fm / fm v-size))])
    fm))
(define flndiv/zero
  (flonap-lift-helper2 'flndiv/zero (λ (x y) (if (y = 0 0.0) 0.0 (/ x y)))))
(: flonap-divide-alpha (flonap -> flonap))
(define (flonap-divide-alpha fm)
  (match-define (flonap _ c w h) fm)
  (cond [(c = 0 1) fm]
    [else
      (define alpha-fm (flonap-ref-component fm 0))
      (flonap-append-components alpha-fm (flndiv/zero (flonap-drop-components fm 1) alpha-fm))]))
```

```
(deep-flonap argb-fm (fm- z-fm z)))
(: deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
(define (deep-flonap-smooth-z dfm o)
  (let ([o (exact->inexact o)])
    (match-define (deep-flonap argb-fm z-fm) dfm)
    (define new-z-fm (flonap-blur z-fm o))
    (deep-flonap argb-fm new-z-fm)))
; 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 dfm z)
  (match-define (deep-flonap argb-fm z-fm) dfm)
  (define alpha-fm (deep-flonap-alpha dfm))
  (deep-flonap argb-fm (fm- z-fm (fm+ alpha-fm z))))
(: deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
(define (deep-flonap-emboss dfm xy-ant z-ant)
  (let ([o (/ xy-ant 3.0)])
    (define z-fm (flonap-normalize (deep-flonap-alpha dfm)))
    (define new-z-fm (fm+ (flonap-blur z-fm o) z-ant))
    (deep-flonap-raise dfm new-z-fm)))
(: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> Flonum) -> deep-flonap))
(define (deep-flonap-bulge-helper dfm f)
  (let ()
    (define-values (w h) (deep-flonap-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-flonap
        1 w h
        (λ (x y i)
          (f (- (/ (fx->fl x) half-x-size) 1.0)
            (- (/ (fx->fl y) half-y-size) 1.0)))))
    (deep-flonap-raise dfm z-fm)))
(: deep-flonap-bulge (deep-flonap (Flonum Flonum -> Real) -> deep-flonap))
(define (deep-flonap-bulge dfm f)
  (deep-flonap-bulge-helper dfm (λ (cx cy) (real->double-flonum (f cx cy)))))
```



```

lang typed/racket/basis
(require racket/racket/racket/math racket/flonum
         (except-in racket/flonum 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 fm)
  (match-define (flonap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flonap c w h (λ (x y z)
                               (unsafe-flvector-ref vs (coords->index c w k (fx- w-1 x) y))))))

(define (flonap-flip-vertical fm)
  (match-define (flonap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flonap c w h (λ (x k y)
                               (unsafe-flvector-ref vs (coords->index c w k x (fx- h-1 y) k))))))

(define (flonap-transpose fm)
  (match-define (flonap vs c w h) fm)
  (inline-build-flonap c h w (λ (k x y)
                               (unsafe-flvector-ref vs (coords->index c w k x y))))))

(define (flonap-cw-rotate fm)
  (match-define (flonap vs c w h) fm)
  (define h-1 (fx- h 1))
  (inline-build-flonap c w h (λ (k x y)
                               (unsafe-flvector-ref vs (coords->index c w k (fx- h-1 y) k))))))

(define (flonap-ccw-rotate fm)
  (match-define (flonap vs c w h) fm)
  (define w-1 (fx- w 1))
  (inline-build-flonap c h w (λ (k x y)
                               (unsafe-flvector-ref vs (coords->index c w k y (fx- w-1 x) k))))))

(struct: invertible-2d-function ([f : (Flonum Flonum -> (values Flonum Flonum))]
                                [g : (Flonum Flonum -> (values Flonum Flonum))]))

(define-type Flonap-Transform (Integer Integer -> invertible-2d-function))
(: (transform-compose (Flonap-Transform Flonap-Transform -> Flonap-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 (λ (k: (Flonum) [y : Flonum])
                           (let-values (((x y) (f1 x y)))
                             (f2 x y)))
                          (λ (k: (Flonum) [y : Flonum])
                           (let-values (((x y) (g1 x y)))
                             (g2 x y))))))

(: (flonap-transform (case-> (Flonap Flonap-Transform -> flonap)
                             (Flonap Flonap-Transform Real Real Real -> flonap)))
(define flonap-transform
  (case-lambda
   [(fn t)
    (match-define (flonap vs c w h) fm)
    (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= 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)))])))
    (flonap-transform fm 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) 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-x (- int-x-max int-x-min))
      (define new-y (- 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)
        (define-values (old-x old-y) (g (+ (fx->fl x) x-offset)
                                           (+ (fx->fl y) y-offset)))
        (flonap-bilinear-ref fm k old-x old-y))))))])

```

```

range typed/racket/base
(require racket/flonum)

(define (racket/flonum fl)
  (except-in racket/flonum 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
  fmgg fmgas fmgqr fasin fncos ftan flog fneap fnsqr fmsin fmcos fntan
  fmround fmfloor fmceiling ftruncate fzzero
  fsw fw fw' fmsin fmxax
  flonap-normalize flonap-multiply-alpha flonap-divide-alpha)

; =====
; Unary
;
; (flonap-lift-helper : (Float -> Float) -> (flonap -> flonap))
(define (flonap-lift-helper f)
  (λ (x) (fml : (flonap
    (match-define (flonap vs c w h) fml)
    (flonap (inline-build-flvector (* c w h)
      (λ (unsafe-flvector-ref vs i))
      (c w h)))))

; (flonap-lift2 (Flonum -> Real) -> (flonap -> flonap))
(define (flonap-lift2 op)
  (flonap-lift-helper (λ (x) (real-double-flonum (op x)))))
(define fmgg (flonap-lift-helper -))
(define fmgas (flonap-lift-helper abs))
(define fmgqr (flonap-lift-helper sqrt))
(define fmsin (flonap-lift-helper sin))
(define fmcos (flonap-lift-helper cos))
(define ftan (flonap-lift-helper tan))
(define flog (flonap-lift-helper log))
(define fneap (flonap-lift-helper exp))
(define fnsqr (flonap-lift-helper sqrt))
(define fmsin (flonap-lift-helper asin))
(define fmcos (flonap-lift-helper acos))
(define fntan (flonap-lift-helper atan))
(define fmround (flonap-lift-helper round))
(define fmfloor (flonap-lift-helper floor))
(define fmceiling (flonap-lift-helper ceiling))
(define ftruncate (flonap-lift-helper truncate))
(define fzzero (flonap-lift-helper (λ (x) (if (x . = . 0.0) 1.0 0.0))))
; =====
; Binary
;
; (flonap-lift-helper2 : Symbol (Float Float -> Float) -> ((U Real flonap) -> flonap))
(define (flonap-lift-helper2 name f)
  (let ()
    (λ (fml : (U Real flonap)) (fml2 : (U Real flonap))
      [cond
        (and (real? fml) (real? fml2))
          (error name "expected at least one flonap argument; given ~e and ~e" fml fml2)
        [real? fml] (let ([fml (real-double-flonum fml)])
          ((flonap-lift-helper (λ (v) (f fml v))) fml2))
        [real? fml2] (let ([fml2 (real-double-flonum fml2)])
          ((flonap-lift-helper (λ (v) (f v fml2))) fml))
        [else
          (match-define (flonap vs1 c1 w1 h1) fml)
          (match-define (flonap vs2 c2 w2 h2) fml2)
          [cond
            (not (and (= w1) (= h1)))
              (error name "expected same-size flonaps; given sizes ~e-e and ~e-e" w1 w2 h1)
            (= c1 c2) (define n (* c1 w1 h1))
              (define res-vs (make-flvector n))
              (flonap (inline-build-flvector n (λ (i) (f (unsafe-flvector-ref vs1 i)
                (unsafe-flvector-ref vs2 i))))
                (unsafe-flvector-ref vs2 i))))
            (else
              (error name (string-append "expected flonaps with the same number of components, ~e"
                "or a flonap with 1 component and any same-size flonap; ~e"
                "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 fm+ (flonap-lift-helper2 '+))
(define fm- (flonap-lift-helper2 '-))
(define fm* (flonap-lift-helper2 '*'))
(define fm/ (flonap-lift-helper2 '/))
(define fmin (flonap-lift-helper2 'min))
(define fmax (flonap-lift-helper2 'max'))
; (flonap-normalize (flonap -> flonap))
(define (flonap-normalize fm)
  (define-values (v-min v-max) (flonap-extreme-values fm))
  (define v-size (- v-max v-min))
  (let* ([fm (fm -v-min)]
    [fm (if (v-size . = . 0.0) fm (fm / v-size))])
    fm)

  (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 fm)
  (match-define (flonap _ c w h) fm)
  (let ([c (< . 1) fm]
    [else
      (define alpha-fm (flonap-ref-component fm 0))
      (flonap-append-components alpha-fm (fndiv/zero (flonap-drop-components fm 1) alpha-fm))]]))

```

```

range typed/racket/base
(require racket/fixnum fx => 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-lt-append deep-flonap-lc-append deep-flonap-lb-append
deep-flonap-rt-append deep-flonap-rc-append deep-flonap-rb-append
deep-flonap-lt-append deep-flonap-lc-append deep-flonap-lb-append
deep-flonap-rt-append deep-flonap-rc-append deep-flonap-rb-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 --w-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)
  (define (deep-flonap-fn (make-flonap 'z w h))
    (: deep-flonap-width (deep-flonap -> Nonnegative-Fixnum))
    (define (deep-flonap-width dfm)
      (define w (flonap-width (deep-flonap-argb dfm)))
      (with-asserts (λw nonnegative-fixnum?)
        w))
    (: deep-flonap-height (deep-flonap -> Nonnegative-Fixnum))
    (define (deep-flonap-height dfm)
      (define h (flonap-height (deep-flonap-argb dfm)))
      (with-asserts (λh nonnegative-fixnum?)
        h))
    (: deep-flonap-z-min (deep-flonap -> Flonum))
    (define (deep-flonap-z-min dfm)
      (flonap-min-value (deep-flonap-z dfm)))
    (: deep-flonap-z-max (deep-flonap -> Flonum))
    (define (deep-flonap-z-max dfm)
      (flonap-max-value (deep-flonap-z dfm)))
    (: flonap-max-value (deep-flonap-z dfm))
    (: deep-flonap-size (deep-flonap -> (values Nonnegative-Fixnum Nonnegative-Fixnum)))
    (define (deep-flonap-size dfm)
      (values (deep-flonap-width dfm) (deep-flonap-height dfm)))
    (: deep-flonap-alpha (deep-flonap -> flonap))
    (define (deep-flonap-alpha dfm)
      (flonap-rtf-component (deep-flonap-argb dfm) 0))
    (: deep-flonap-rgb (deep-flonap -> flonap))
    (define (deep-flonap-rgb dfm)
      (flonap-drop-components (deep-flonap-argb dfm) 1))
    )
  )

; =====
; Z adjusters
(: deep-flonap-scale-z (deep-flonap (U Real flonap) -> deep-flonap))
(define (deep-flonap-scale-z deep-flonap z-fn)
  (match-define (deep-flonap argb-fn z-fn) dfm)
  (deep-flonap-argb-fn (fn* z-fn z-fn))
  (: deep-flonap-smooth-z (deep-flonap Real -> deep-flonap))
  (define (deep-flonap-smooth-z dfm)
    (let ([o (exact-inexact 0)])
      (define (deep-flonap-smooth-z-fn z-fn) dfm)
      (define new-z-fn (flonap-blur z-fn o) dfm)
      (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 deep-flonap z-fn)
  (match-define (deep-flonap argb-fn z-fn) dfm)
  (define alpha-fn (deep-flonap-alpha dfm))
  (deep-flonap-argb-fn (fn* z-fn (fn* alpha-fn z-fn)))
  (: deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
  (define (deep-flonap-emboss dfm xy-ant z-ant)
    (let ([o (/ xy-ant 3.0)])
      (define z-fn (flonap-normalize (deep-flonap-alpha dfm)))
      (define new-z-fn (fn* (flonap-blur z-fn o) z-ant))
      (deep-flonap-raise deep-flonap new-z-fn))
    )
  (: deep-flonap-bulge-helper (deep-flonap (flonum Flonum -> flonum) -> deep-flonap))
  (define (deep-flonap-bulge-helper dfm f)
    (let ()
      (define-values (w h) (deep-flonap-size dfm))
      (define half-z-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
          λ w h
          (λ (x y z)
            (f (- (/ (fx->fl x) half-z-size) 2.0)
              (- (/ (fx->fl y) half-y-size) 2.0))))))
      (deep-flonap-raise dfm z-fn))
    )
  )
(: deep-flonap-bulge (deep-flonap (U Real flonap) -> Real) -> deep-flonap)
(define (deep-flonap-bulge dfm f)
  (deep-flonap-bulge-helper dfm (λ (cx cy) (real-double-flonum (f cx cy)))))

```

[illegible]

Optimization Coach

per

(0 success out of 46
his function into a ma

```

                                "or a flopmap with 1 component and any same-size flopmap;"
                                "given flopmaps with -e and -e components")

(c2 c2)))))))))

(: flopmap-lift2 (Symbol (Flopsum Flopsum -> Real) -> ((U Real Flopmap) (U Real Flopmap) -> Flopmap)))
(define (flopmap-lift2 name v)
  (flopmap-lift-helper2 name (lambda (x y) (real->double-flopsum (f x y))))
  (define f#m (flopmap-lift-helper2 'f#m ->))
  (define f#n (flopmap-lift-helper2 'f#n ->))
  (define f#m' (flopmap-lift-helper2 'f#m' ->))
  (define f#n' (flopmap-lift-helper2 'f#n' ->))
  (define f#mmin (flopmap-lift-helper2 'f#mmin min))
  (define f#mmax (flopmap-lift-helper2 'f#mmax max))
  (: flopmap-normalize (flopmap -> Flopmap))
  (define (flopmap-normalize fm)
    (define-values (v-min v-max) (flopmap-extreme-values fm))
    (define v-size (- v-max v-min))
    (let* ((fn (fm (fm- fm v-min)))
           (fn' (if (v-size . = . 0.0) fm (fm/ fm v-size))))
      fn))
  (define f#mdiv/zero
    (flopmap-lift-helper2 'f#mdiv/zero (lambda (x y) (if (y . = . 0.0) 0.0 (/ x y))))
  (: flopmap-divide-alpha (flopmap -> Flopmap))
  (define (flopmap-divide-alpha fm)
    (match-define (flopmap _ c w h) fm)
    (cond [(c . = . 1) fn]
          [else
           (define alpha-fm (flopmap-ref-component fm 0))
           (flopmap-append-components alpha-fm (f#mdiv/zero (flopmap-drop-components fm 1) alpha-fm))))))

```

```

#lang Typed/racket/base
(require racket/flonum

  racket/math
  racket/match racket/math
  "flonum.rkt"
  "flonum.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-rbg
  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-rc-superimpose deep-flonap-rc-superimpose deep-flonap-rb-superimpose
  deep-flonap-vl-append deep-flonap-vc-append deep-flonap-vr-append
  deep-flonap-hl-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 "incompatible dimensions"))))

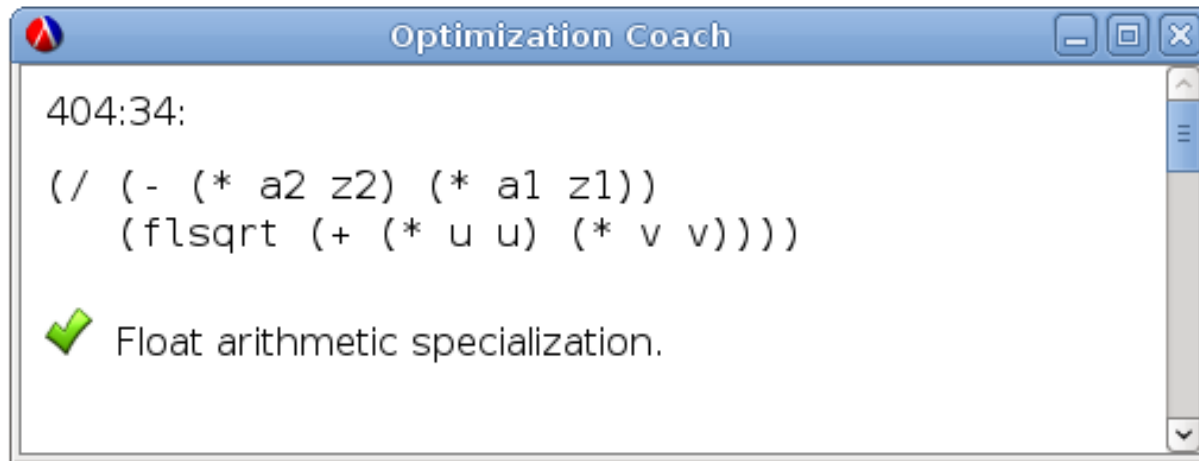
; force inlining.

(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 dfm z)
  (match-define (deep-flonap argb-fn z-fn) dfm)
  (define alpha-fn (deep-flonap-alpha dfm))
  (deep-flonap argb-fn (for z-fn (for* (alpha-fn z)))
    (: deep-flonap-emboss (deep-flonap Real (U Real flonap) -> deep-flonap))
    (define (deep-flonap-emboss dfm xy-ant z-ant)
      (let ([o (/ xy-ant 3.0)])
        (define z-fn (flonap-normalize (deep-flonap-alpha dfm)))
        (define new-z-fn (for* (flonap-blur z-fn o) z-ant))
        (deep-flonap-raise dfm new-z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum -> Flonum) -> deep-flonap))
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ 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 dfm z-fn)))
    (: deep-flonap-bulge-helper (deep-flonap (Flonum Flonum) -> Real) -> deep-flonap)
    (define (deep-flonap-bulge-helper dfm f)
      (let ()
        (define-values (w h) (deep-flonap-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-fn
          (inline-build-flonap
            λ w h
            (λ (
```

Dialog between compilers and programmers



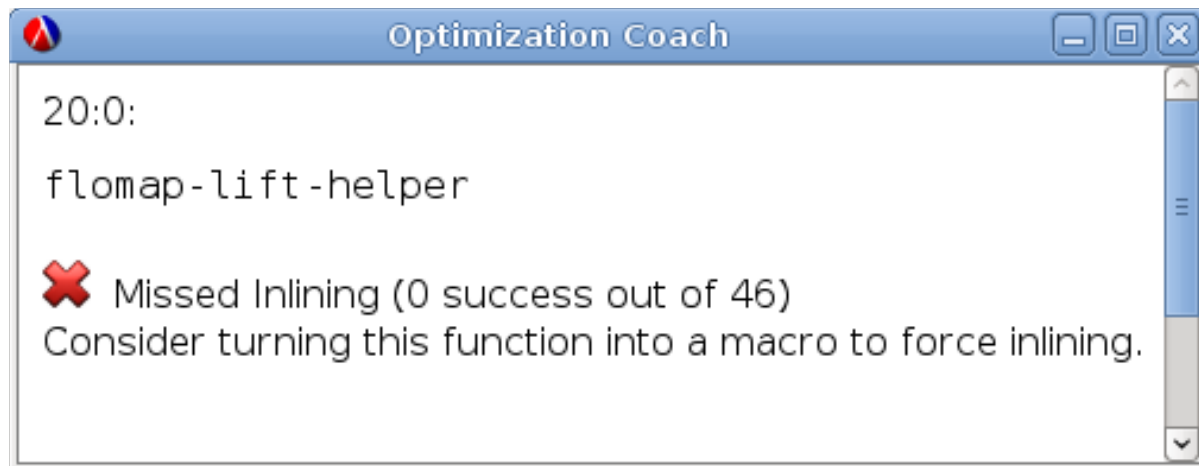
Optimization Coach

404:34:

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

✓ Float arithmetic specialization.

This screenshot shows a window titled "Optimization Coach" with a blue header bar. The main content area is white and contains the text "404:34:" followed by a Scheme-like expression: `(/ (- (* a2 z2) (* a1 z1)) (flsqrt (+ (* u u) (* v v))))`. Below the code, there is a green checkmark icon followed by the text "Float arithmetic specialization." The window has standard OS controls (minimize, maximize, close) in the top right corner.



Optimization Coach

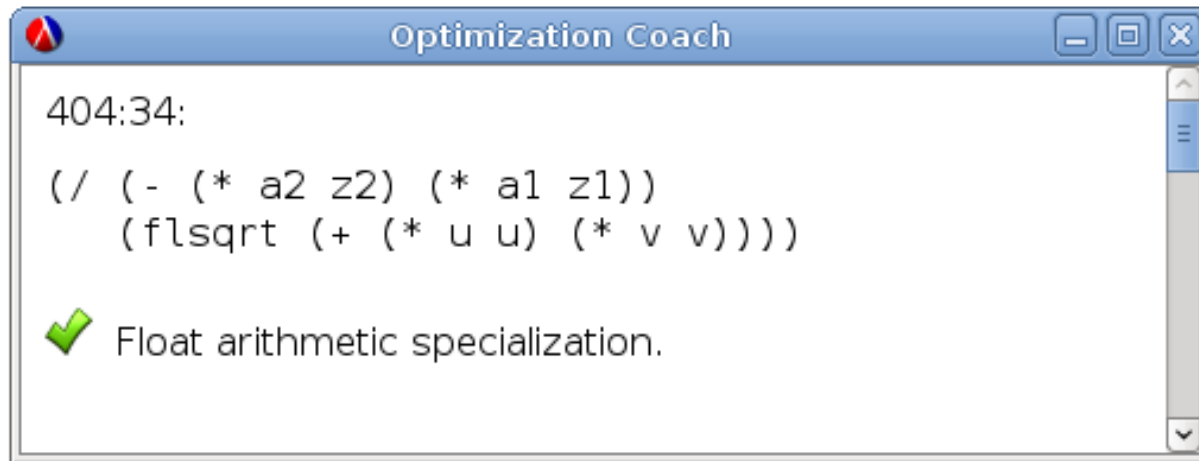
20:0:

flomap-lift-helper

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

This screenshot shows another "Optimization Coach" window. The main content area is white and contains the text "20:0:" followed by the function name "flomap-lift-helper". Below this, there is a red 'X' icon followed by the text "Missed Inlining (0 success out of 46)" and "Consider turning this function into a macro to force inlining." The window has standard OS controls in the top right corner.

Dialog between compilers and programmers



Optimization Coach

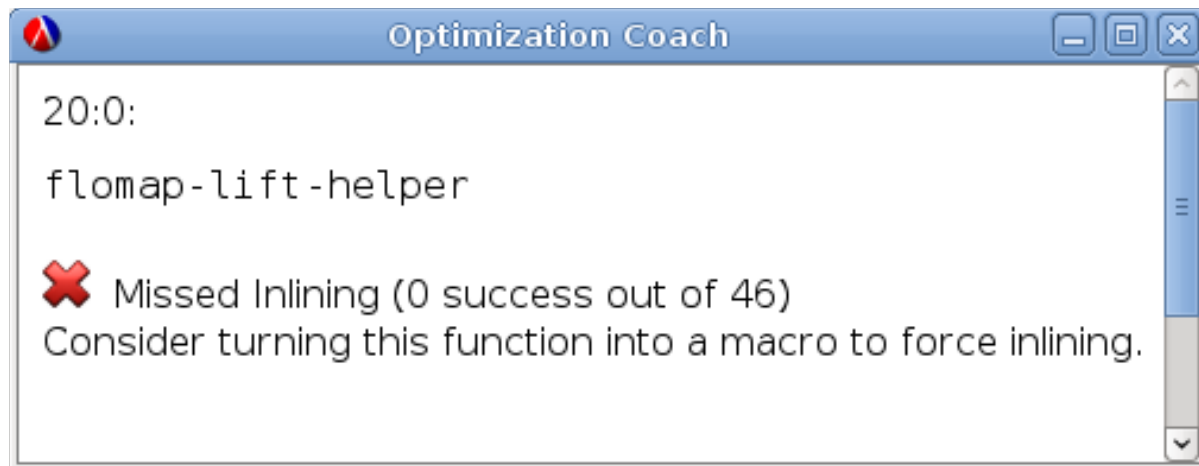
404:34:

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

✓ Float arithmetic specialization.

This screenshot shows a window titled "Optimization Coach" with a blue header bar. The main text area displays a code snippet at line 404:34, which is a division of two expressions. Below the code, a green checkmark icon is followed by the text "Float arithmetic specialization.", indicating a successful optimization.

Successes



Optimization Coach

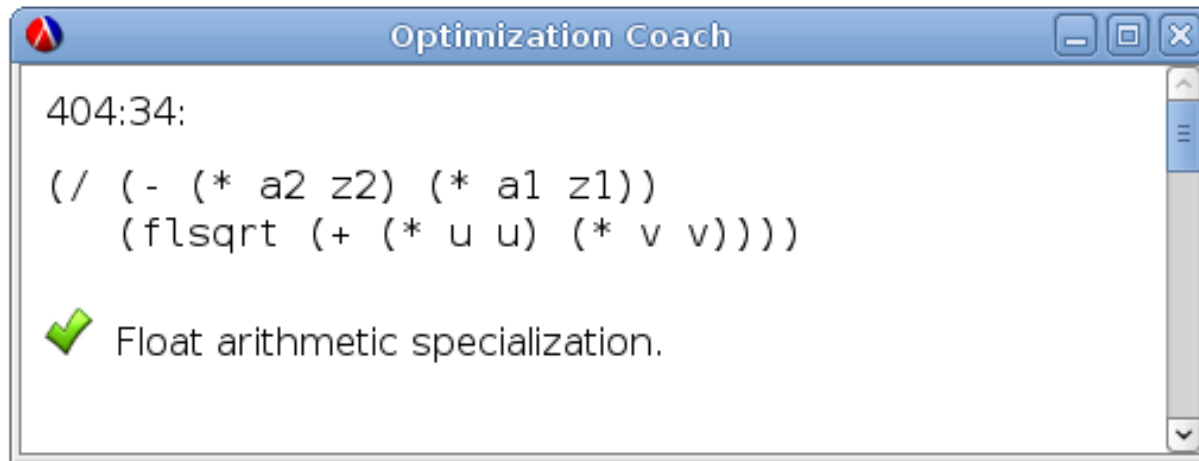
20:0:

flomap-lift-helper

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

This screenshot shows another "Optimization Coach" window. It displays the code snippet "flomap-lift-helper" at line 20:0. Below this, a red X icon is followed by the text "Missed Inlining (0 success out of 46)" and a suggestion: "Consider turning this function into a macro to force inlining.", indicating a failed optimization attempt.

Dialog between compilers and programmers



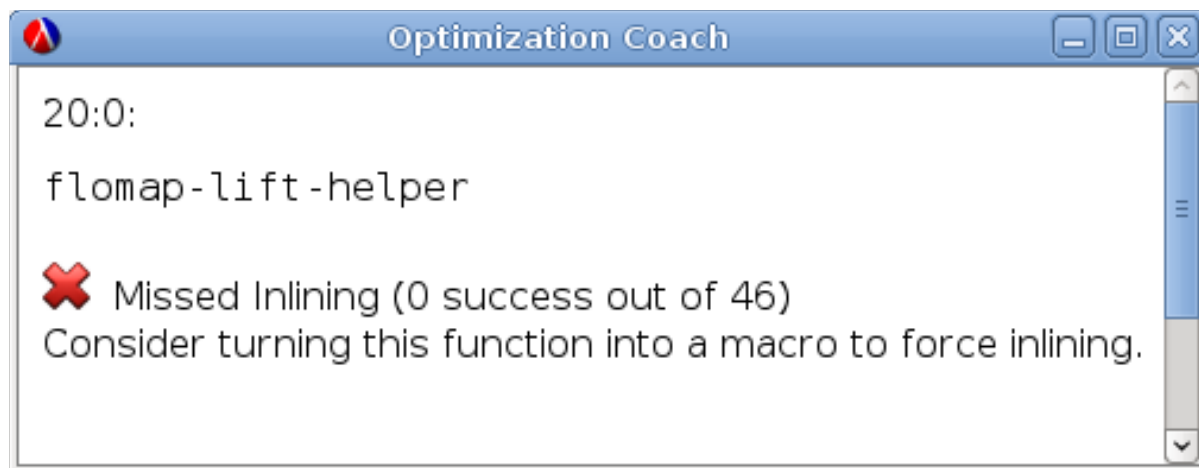
Optimization Coach

404:34:

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

✓ Float arithmetic specialization.

Successes



Optimization Coach

20:0:

flomap-lift-helper

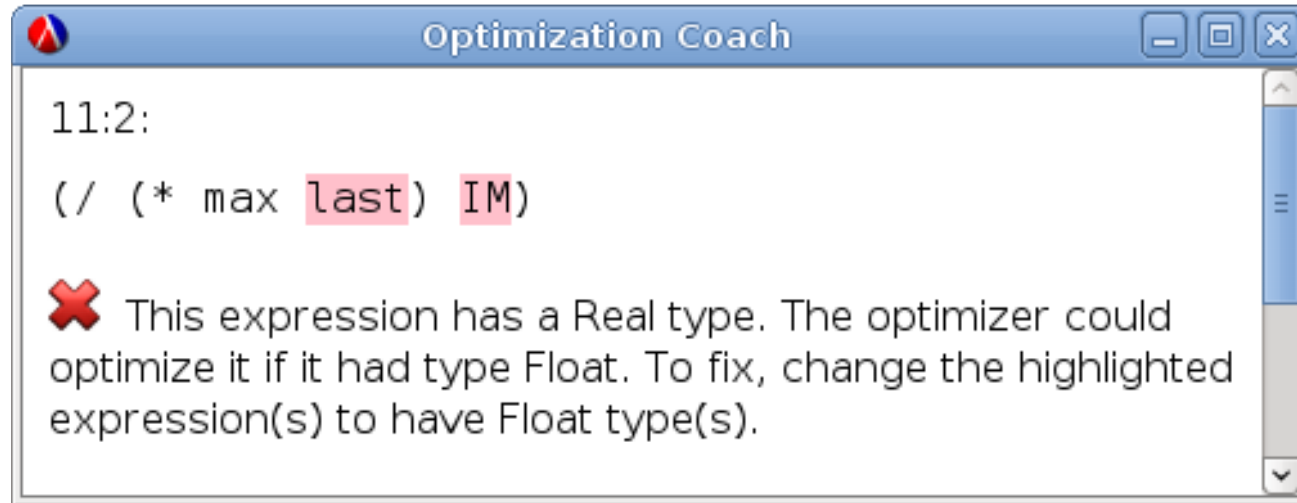
✗ Missed Inlining (0 success out of 46)
Consider turning this function into a macro to force inlining.

Near misses

+

Recommendations

Programmers can do more than compilers



Recommendations can **change semantics!**

`(/ 1 3)` $\rightarrow$ `1/3`

`(/ 1.0 3.0)` $\rightarrow$ `0.3333333333333333`

How does it work?

How well does it work?

How to extend it?

How does it work?

Overview

Compiler Instrumentation



Optimization Analysis



Recommendation Generation



Programmer Response

Type-Driven Specialization

```
#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))
```


Type-Driven Specialization

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

```
(define IM f1-  
(define IA  
(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))
```

Type-Driven Specialization

```
#lang typed/racket
```

```
(define IM
```

```
(define IA
```

```
(define IC
```

```
(define last
```

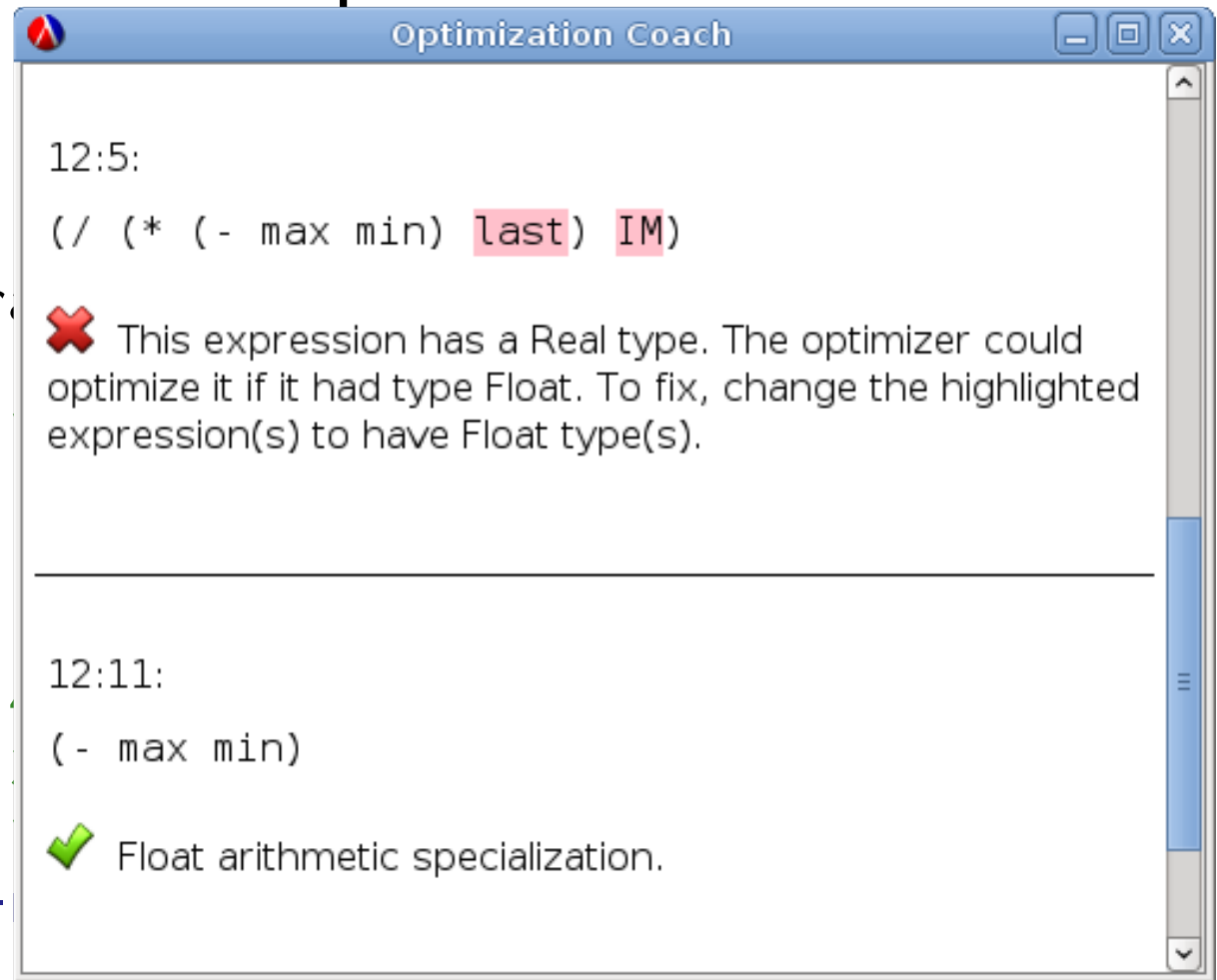
```
(define min
```

```
(define max
```

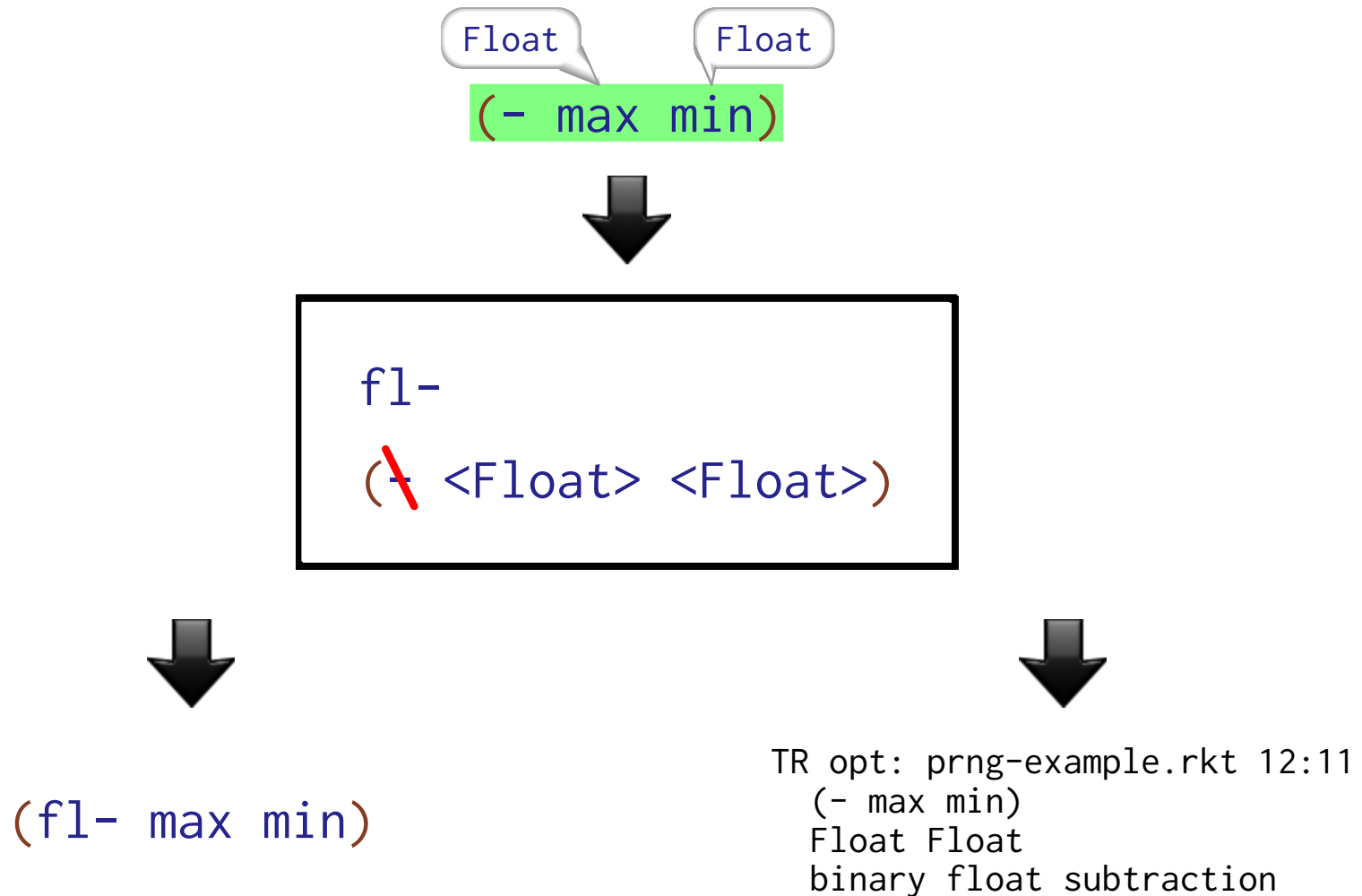
```
(define (gen-
```

```
(set! last
```

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



Compiler Instrumentation



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

Incomprehensible failure pruning

Irrelevant failure pruning

Harmless failure pruning

Optimization proximity

Causality merging

Locality merging

Optimization Analysis

Incomprehensible failure pruning

Irrelevant failure pruning

Harmless failure pruning

Optimization proximity

Causality merging

Locality merging

Optimization Analysis

Incomprehensible failure pruning

Irrelevant failure pruning

Harmless failure pruning

Optimization proximity

Causality merging

Locality merging

Optimization Analysis

Incomprehensible failure pruning

Irrelevant failure pruning

Can't trace back to user program.

Optimization proximity

Causality merging

Locality merging

Optimization Analysis

Incomprehensible failure pruning

Irrelevant failure pruning

Non-optimized semantics is desirable.

Optimization proximity

Causality merging

Locality merging

Optimization Analysis

Optimization proximity

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

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

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
Integer	Integer
Irritant	Irritant

$$\Delta = 2$$

Optimization Analysis

Causality merging

Real Integer

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

Irritant Irritant

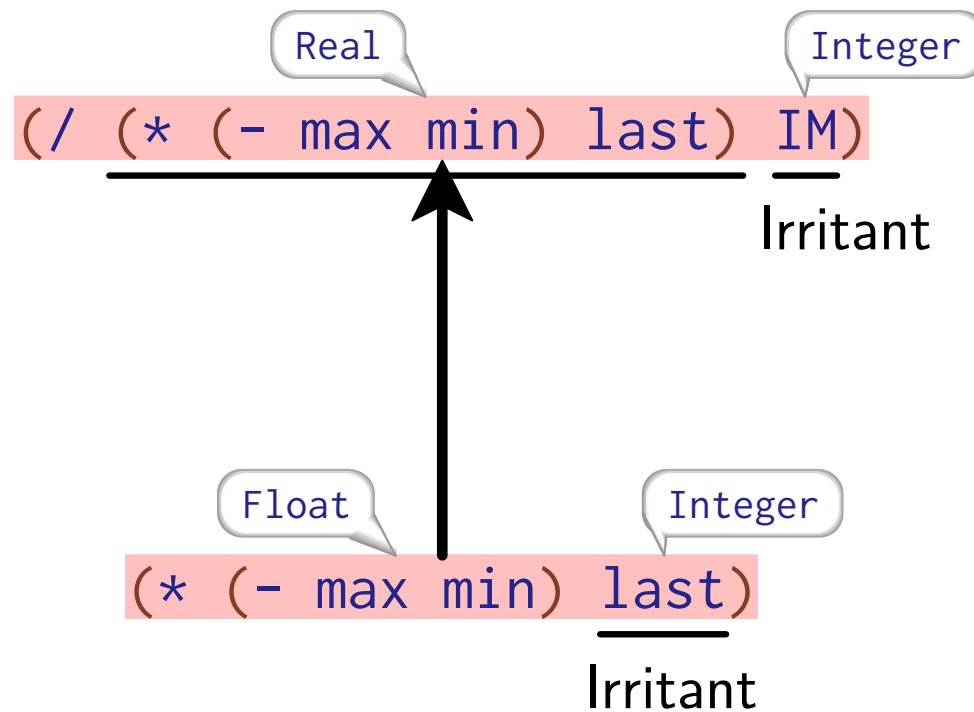
Float Integer

(* (- max min) last)

Irritant

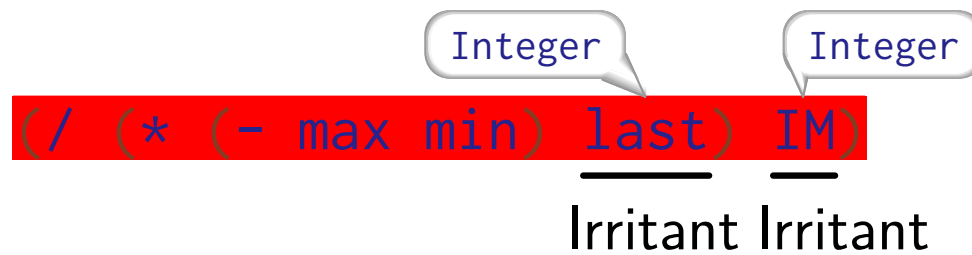
Optimization Analysis

Causality merging



Optimization Analysis

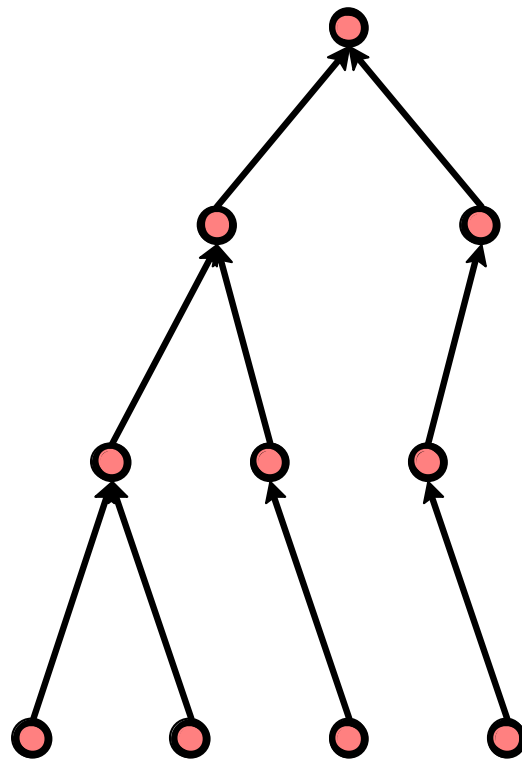
Causality merging


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

Badness = 2

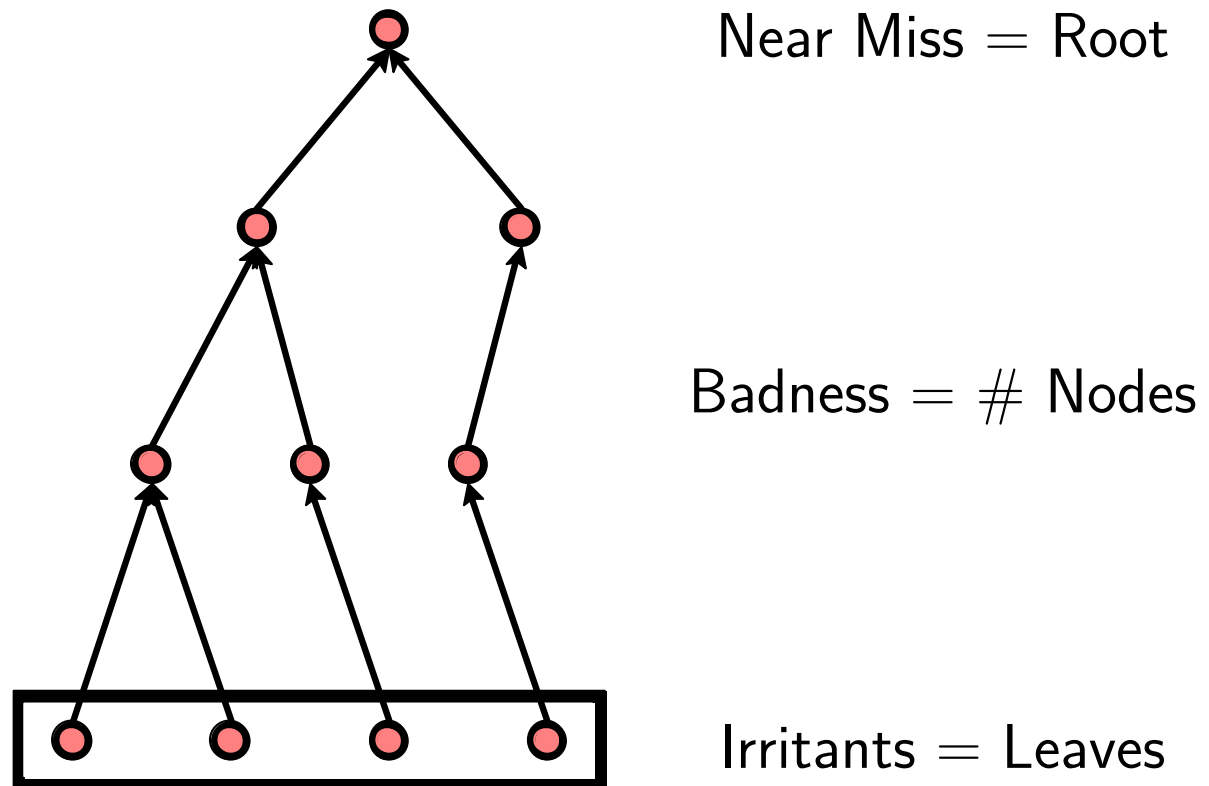
Optimization Analysis

Causality merging

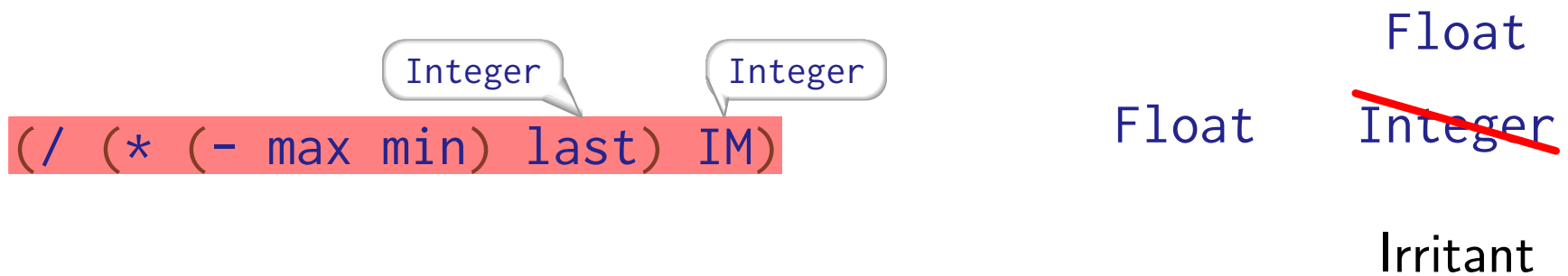


Optimization Analysis

Causality merging



Recommendation Generation



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).

The Racket inliner in one slide

Based on a design by Serrano

Fuel

- Inlining cost $\propto$ Function size
- Inlining stops when out of fuel

Loop unrolling for free!

```
(define (negate-pixel pixel) ; small function
  (+ #x010101 (bitwise-xor pixel #x3f3f3f)))
```

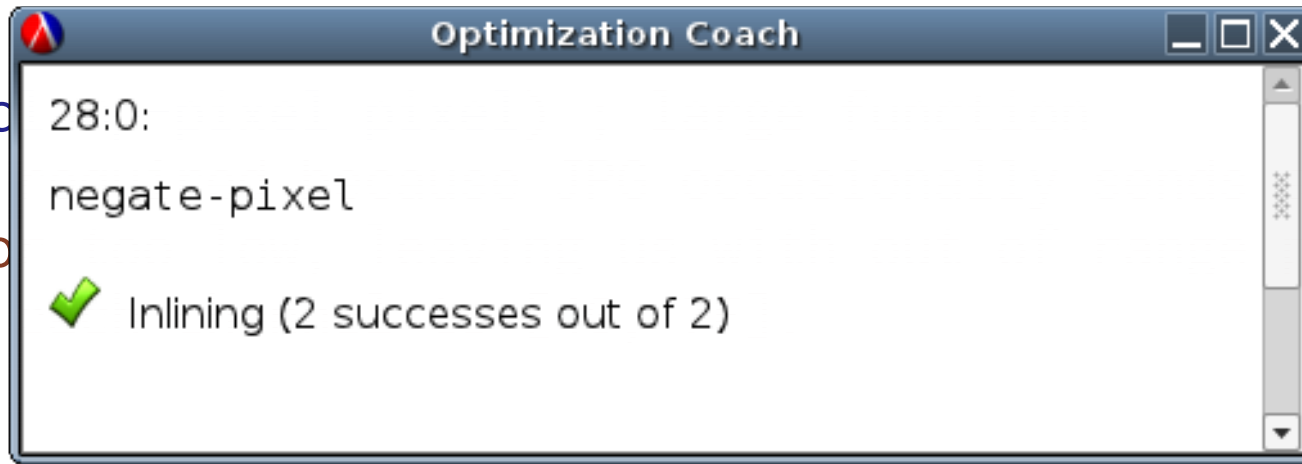
```
(define (clamp-pixel pixel) ; large function
  ; Clamp required because JPG occasionally sends a delta too
  ; high or too low, leaving us with out-of-range pixels.
  ; Clamp each channel to [40, 7F].
  ...)
```

```
(define (kernel-decode base-pixel delta-pixel)
  ... (negate-pixel ...) ...
  ... (clamp-pixel clampable) ...)
```

```
... (kernel-decode ...) ...
```

```
(define (negate-pixel pixel) ; small function
  (+ #x010101 (bitwise-xor pixel #x3f3f3f)))
```

```
(define (clamp-pixel pixel)
  ; Clamp
  ; high
  ; Clamp
  ...)
```



delta too
pixels.

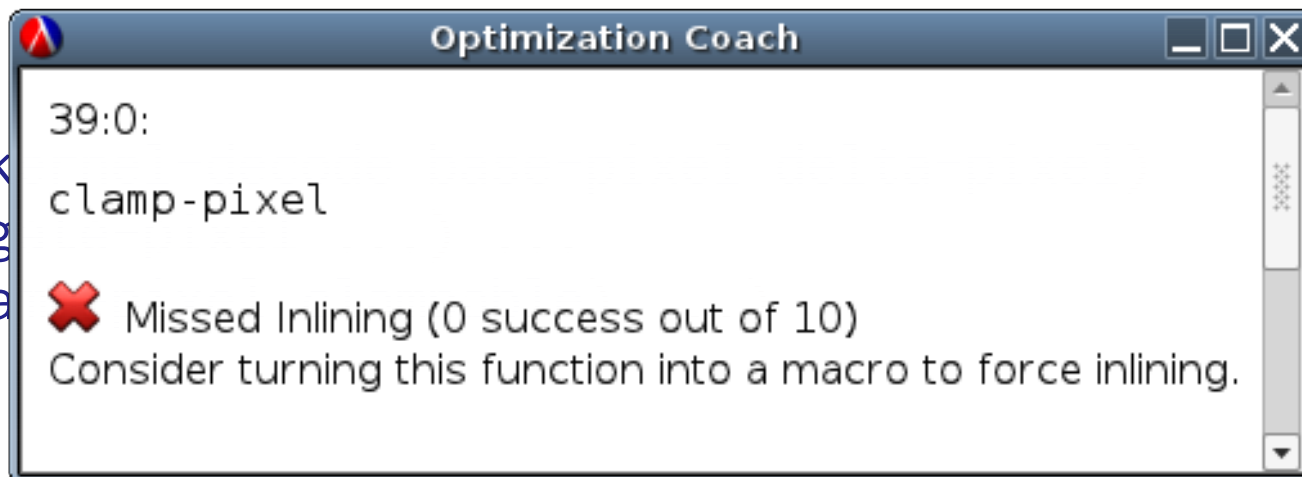
```
(define (kernel-decode base-pixel delta-pixel)
  ... (negate-pixel ...) ...
  ... (clamp-pixel clampable) ...)

... (kernel-decode ...) ...
```

```
(define (negate-pixel pixel) ; small function
  (+ #x010101 (bitwise-xor pixel #x3f3f3f)))
```

```
(define (clamp-pixel pixel) ; large function
  ; Clamp required because JPG occasionally sends a delta too
  ; high or too low, leaving us with out-of-range pixels.
  ; Clamp each channel to [40, 7F].
  ...)
```

```
(define (kernel-decode ...)
  ... (negate-pixel ...)
  ... (clamp-pixel ...))
```

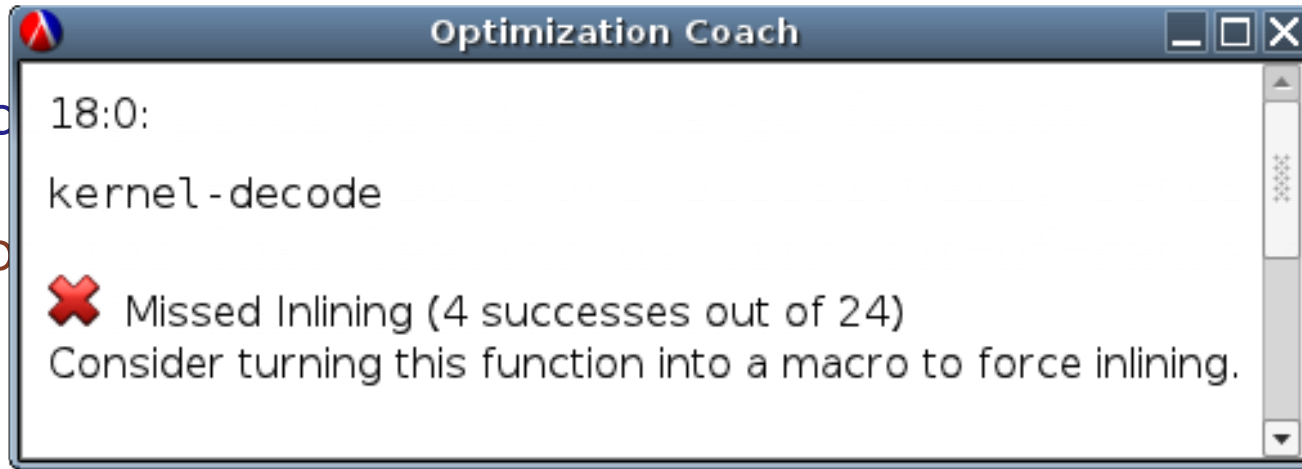


```
... (kernel-decode ...) ...
```



```
(define (negate-pixel pixel) ; small function
  (+ #x010101 (bitwise-xor pixel #x3f3f3f)))
```

```
(define (clamp-pixel pixel)
  ; Clamp
  ; high o
  ; Clamp
  ...)
```



delta too
pixels.

```
(define (kernel-decode base-pixel delta-pixel)
  ... (negate-pixel ...) ...
  ... (clamp-pixel clampable) ...)
```

```
... (kernel-decode ...) ...
```

Compiler Instrumentation

inlining:

**#(negate-pixel #<path:video.rkt> 28 0 793 73 #f)
in: video.rkt:18:0: kernel-decode
size: 1 fuel: 96**

no inlining, out of fuel:

**#(clamp-pixel #<path:video.rkt> 39 0 1228 534 #f)
in: video.rkt:18:0: kernel-decode
size: 99 fuel: 96**

Compiler Instrumentation

inlining:

```
$(negate-pixel #<path:video.rkt> 28 0 793 73 #f)  
in: video.rkt:18:0: kernel-decode  
size: 1 fuel: 96
```

no inlining, out of fuel:

```
$(clamp-pixel #<path:video.rkt> 39 0 1228 534 #f)  
in: video.rkt:18:0: kernel-decode  
size: 99 fuel: 96
```

Compiler Instrumentation

inlining:

```
$(negate-pixel #<path:video.rkt> 28 0 793 73 #f)  
in: video.rkt:18:0: kernel-decode  
size: 1 fuel: 96
```

no inlining, out of fuel:

```
$(clamp-pixel #<path:video.rkt> 39 0 1228 534 #f)  
in: video.rkt:18:0: kernel-decode  
size: 99 fuel: 96
```

Compiler Instrumentation

inlining:

```
$(negate-pixel #<path:video.rkt> 28 0 793 73 #f)  
in: video.rkt:18:0: kernel-decode  
size: 1 fuel: 96
```

no inlining, out of fuel:

```
$(clamp-pixel #<path:video.rkt> 39 0 1228 534 #f)  
in: video.rkt:18:0: kernel-decode  
size: 99 fuel: 96
```

Optimization Analysis

Harmless Failure Pruning

```
no inlining, out of fuel:  
#(for-loop #<path:video.rkt> 63 2 2353 281 #f)  
in: video.rkt:63:2: for-loop  
size: 52 fuel: 8
```

Unrolling has to stop at some point

Optimization Analysis

Locality Merging

4×

```
inlining:  
#(kernel-decode #<path:video.rkt> 18 0 427 276 #f)
```

20×

```
no inlining, out of fuel:  
#(kernel-decode #<path:video.rkt> 18 0 427 276 #f)
```

$4 < 20 \rightarrow$ Optimization failure

$$\Delta = \overline{(\text{size} - \text{fuel})}$$

Recommendation Generation

No unrollings: function $\rightarrow$ macro

Otherwise: make smaller / break into pieces

Just over limit: **begin-encourage-inline**

Fast/slow path: break off slow path

Opt./kw. args: multiple specialized functions

Programmer Response

define-syntax-rule

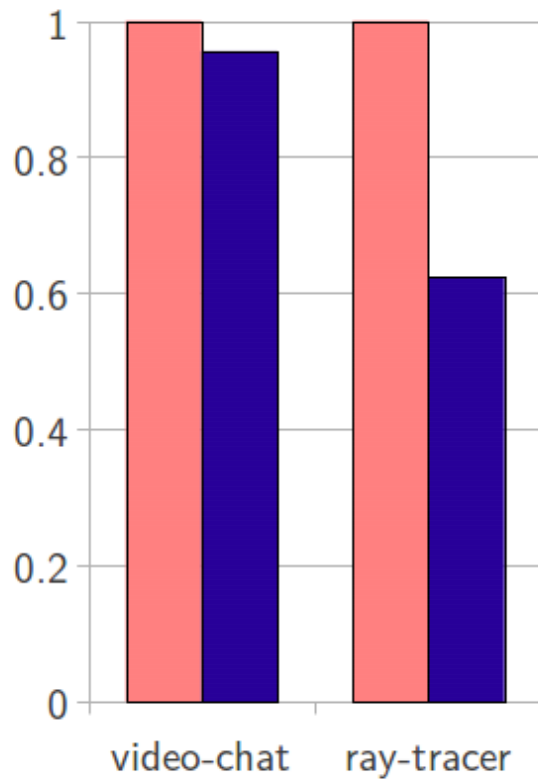
```
(define (clamp-pixel pixel) ; large function  
  ; Clamp required because JPG occasionally sends a delta too  
  ; high or too low, leaving us with out-of-range pixels.  
  ; Clamp each channel to [40, 7F].  
  ...)
```

define-syntax-rule

```
(define (kernel-decode base-pixel delta-pixel)  
  ... (negate-pixel ...) ...  
  ... (clamp-pixel clampable) ...)
```

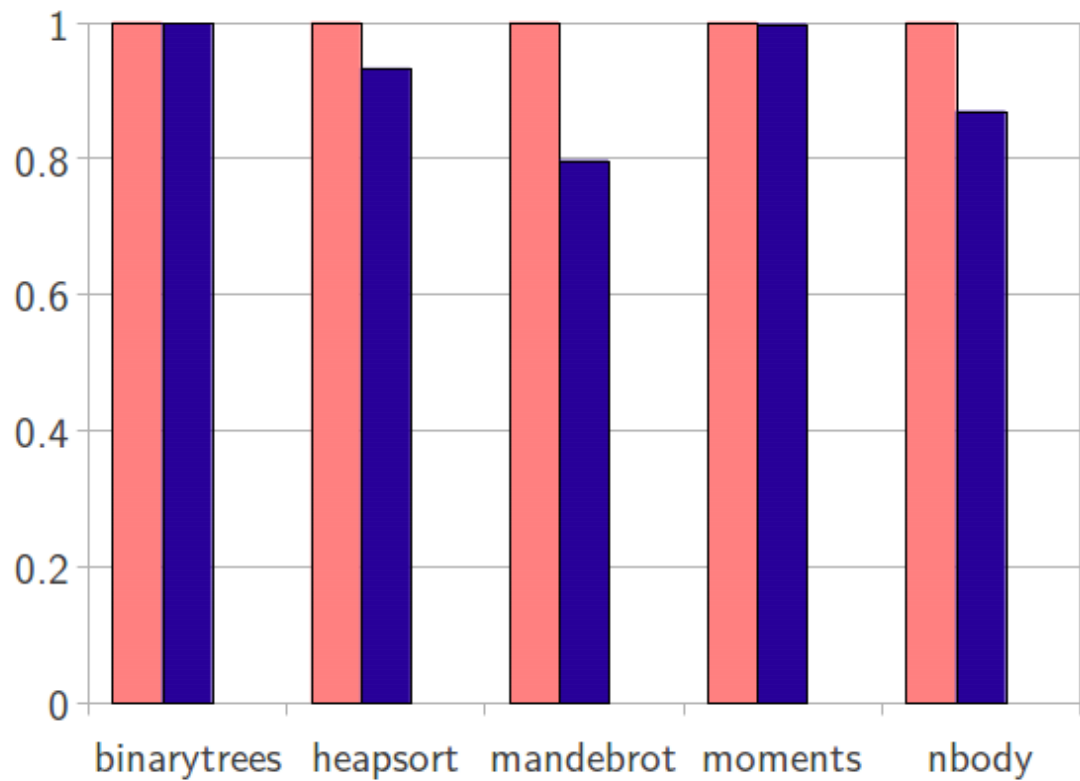
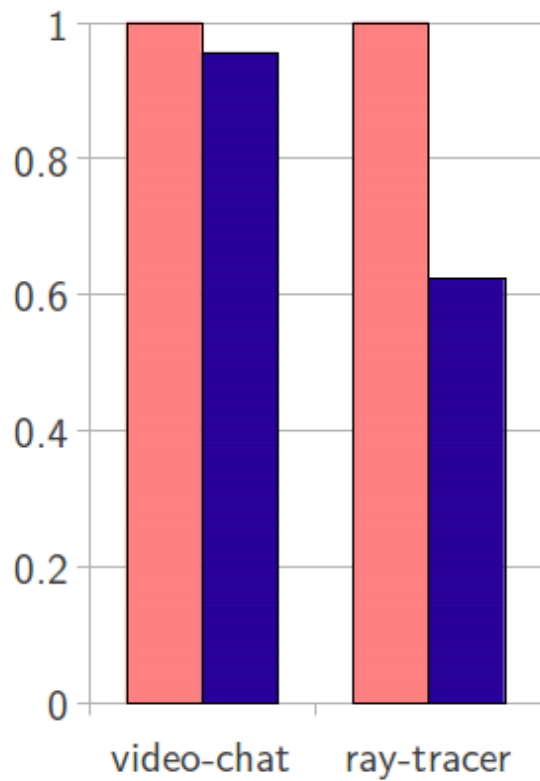
How well does it work?

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



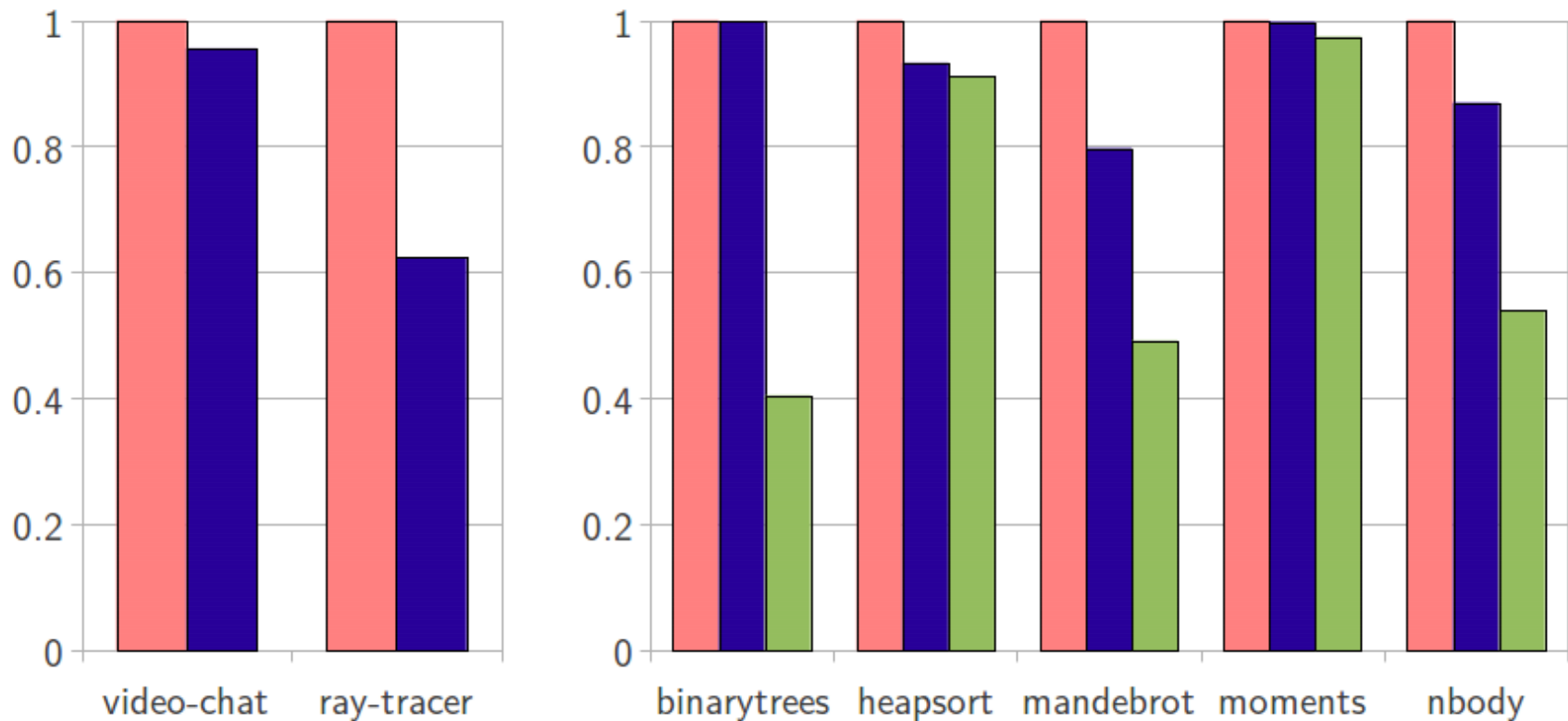
Lower is better

■ 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)

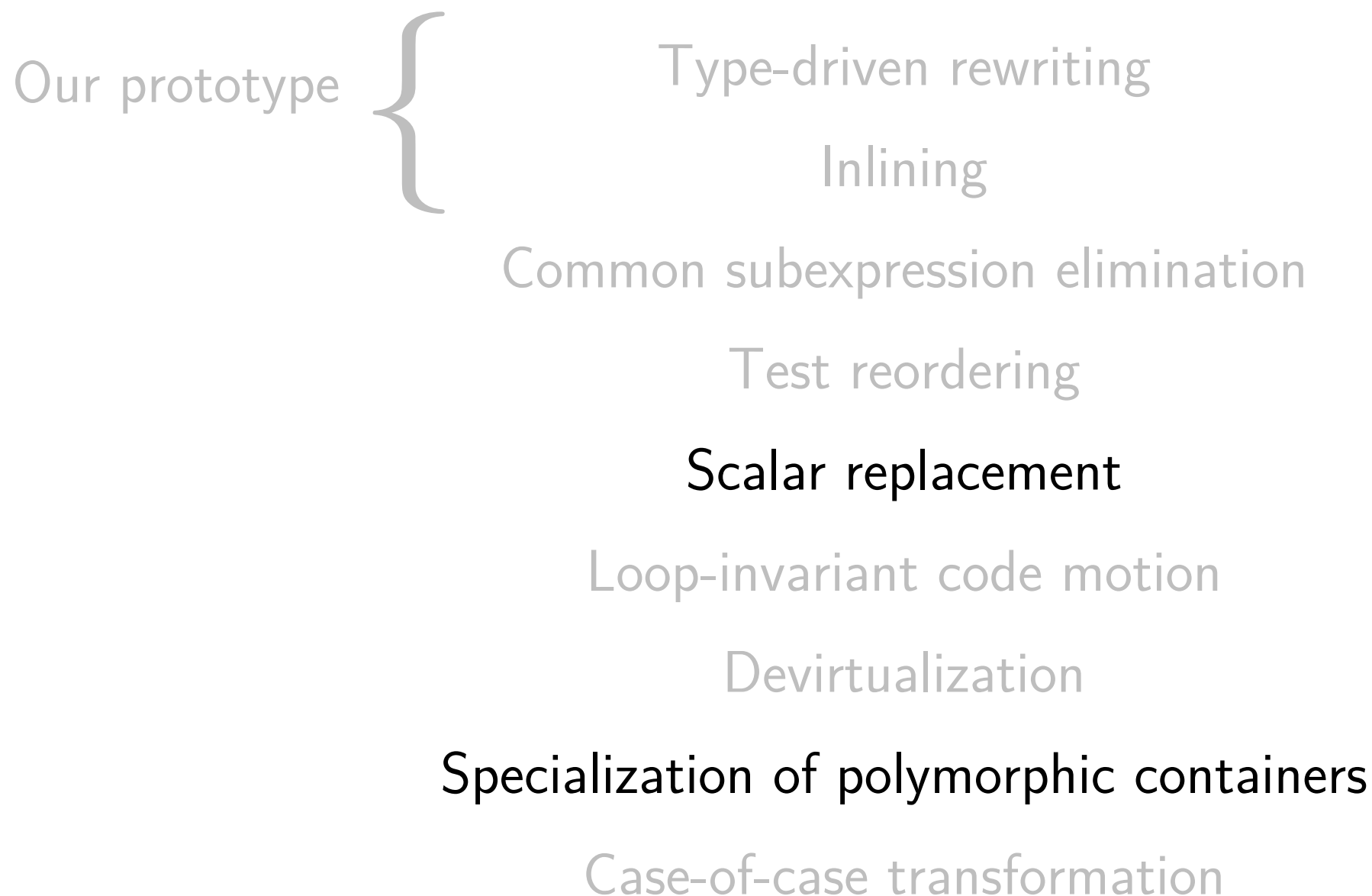


Lower is better

How to extend it?

Our prototype {

- Type-driven rewriting
- Inlining
- Common subexpression elimination
- Test reordering
- Scalar replacement
- Loop-invariant code motion
- Devirtualization
- Specialization of polymorphic containers
- Case-of-case transformation



Coaching Recipe

- Log successes and failures
- Define optimization analysis metrics
- Add recommendation generation logic

Scalar replacement

```
(define twiddle-c  
  (* c (exp (/ (* pi 0.0+1.0i  
                (->fl k))  
                n/2))))
```

```
(+ b twiddle-c)
```

```
(define real-c (real-part c))  
(define imag-c (imag-part c))  
...  
(define real-twiddle-c ...)  
(define imag-twiddle-c ...)
```

```
(define real-b (real-part b))  
(define imag-b (imag-part b))  
(make-rectangular  
  (+ real-b real-twiddle-c)  
  (+ imag-b imag-twiddle-c))
```

Scalar replacement

```
(define twiddle-c  
  (* c (exp (/ (* pi 0.0+1.0i  
                (->fl k))  
                n/2))))
```

```
(printf "~a\n" twiddle-c)  
(+ b twiddle-c)
```

```
(define real-c (real-part c))  
(define imag-c (imag-part c))  
...  
(define real-twiddle-c ...)  
(define imag-twiddle-c ...)
```

```
(define real-b (real-part b))  
(define imag-b (imag-part b))  
(make-rectangular  
  (+ real-b real-twiddle-c)  
  (+ imag-b imag-twiddle-c))
```

Scalar replacement

```
(define twiddle-c  
  (* c (exp (/ (* pi 0.0+1.0i  
                (->fl k))  
                n/2)))))
```

```
(printf "~a\n" twiddle-c)  
(+ b twiddle-c)
```

```
(define real-c (real-part c))  
(define imag-c (imag-part c))  
...  
(define real-twiddle-c ...)  
(define imag-twiddle-c ...)
```

```
(define real-b (real-part b))  
(define imag-b (imag-part b))  
(make-rectangular  
  (+ real-b real-twiddle-c)  
  (+ imag-b imag-twiddle-c))
```

$$\Delta = (\# \text{ boxed uses}) / (\# \text{ unboxed uses})$$

Irritants = {boxed uses}

Specialization of polymorphic containers

```
(define-type 3D-path  
  (Vectorof (List Float Float Float)))
```

```
(: p : 3D-path)  
(define p (vector '(1.2 3.4 5.6)  
                  '(7.8 9.1 0.1)  
                  '(1.1 2.1 3.1)))
```



```
(define p (vector 1.2 3.4 5.6  
                  7.8 9.1 0.1  
                  1.1 2.1 3.1))
```

Specialization of polymorphic containers

```
(define-type 4D-path  
  (Vectorof (List Float Float Float Float Float)))
```

```
(: p : 4D-path)  
(define p (vector '(1.2 3.4 5.6 12.3)  
                   '(7.8 9.1 0.1 45.6)  
                   '(1.1 2.1 3.1 78.9)))
```



~~(define p (vector 1.2 3.4 5.6
 7.8 9.1 0.1
 1.1 2.1 3.1))~~

Specialization of polymorphic containers

```
(define-type 4D-path  
  (Vectorof (List Float Float Float Float Float)))
```

```
(: p : 4D-path)  
(define p (vector '(1.2 3.4 5.6 12.3)  
                  '(7.8 9.1 0.1 45.6)  
                  '(1.1 2.1 3.1 78.9)))
```



~~(define p (vector 1.2 3.4 5.6
 7.8 9.1 0.1
 1.1 2.1 3.1))~~

$$\Delta = (\text{element size}) - (\text{max optimized element size})$$

Conclusion



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

Demo