Contracts in Racket

Robby Findler PLT & Northwestern

An exercise: map

Simple map

Simple map, with tests

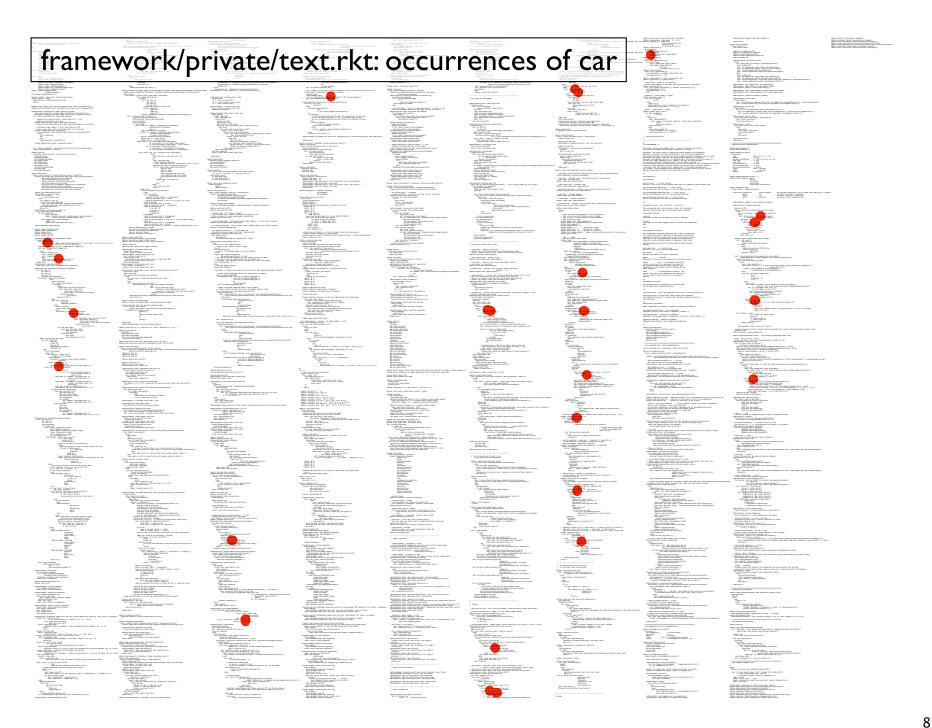
Scheme's map

```
(define (map1 f 1)
  (cond
    [(null? 1) '()]
    [else (cons (f (car 1))
                (map1 f (cdr 1)))]))
(define (map f 11 . 1s)
  (cond
    [(null? 11) null]
    [else
     (cons (apply f (map1 car (cons 11 ls)))
           (apply map f (map1 cdr (cons l1 ls))))))
(check-equal? (map add1 '()) '())
(check-equal? (map add1 '(1 2 3)) '(2 3 4))
(check-equal? (map + '(1 2 3) '(4 5 6)) '(5 7 9))
```

 $(map (\lambda (x) x) 1)$

car: expects argument of type <pair>; given 1

framework/private/text.rkt: occurrences of map A medical medical designation of the control of the Part and Company of the Company of t kefunijaktu (riano amindasi) kaker) kefunijaktu (suoti-dasi) Sala kefuni Jasi dibi angistana (apasal jasiki dang aki-angistana (adanagkala-lanki) salah (adi salahangkalan jakining kefuning pelungkalan pelungkala-kani))) The state of the s (Melhan Milana (Inglatura parkinasari Tindap-parkinasari) parkinasa (salitan Ari - Badari apaka (paskinasi aktion bassishi)) Melhasan a segatap fina parkinasa bata ha takar maparinasa aktion. (bat (Spathasas Aukhina) (palkinash hakar)) The part of the pa [as a controlled part of the process I mad man pri top heat ensise() I mad have best best left-fit) Ind heat () () (separated) (not respect point) (before extent (separate state state respectate) (separate state (separate state) (separate state (separate state) (separate state state state) (separate state state state) (separate state state state) (separate state state state) (separate state state) (separate state) (separate state state) (separate state) (se The state of the s (inform experiments) of the control Comment of the commen (Males Arbites conjustees had "") (Males Arbites speaking Space a Males Arbites (Males Arbites Arbites (Males Arbites Arbites Arbites Arbites (Males Arbites Arbites Arbites Arbites Arbites Arbites (Males Arbites Arb September 17: | September 17: The second section of the section of the second section of the section of the second section of the section Fig. 1985 The second s Secretary of the Control of the Cont Comment of the Commen Technical Control Con A procession of the control of the c The state of the s place described in the control of th Agricultural production of the control of the contr Enter the second The state of the s Ny INDEE DIE NOOM-NA DIE NA MANDRA DIE NA MA proper and the property of the A section of the control of the cont (Million Antho Mill Pillion Announces of Million An See a second sec See a Section of the Control of the | Description | The control of the co Total Street Str The second secon The control of the co (Management of the Control of the Co Special Security Sec., and Just's (1974) (in the 1974 Active) (White In principles and the security of the 1974 Active) (White In principles are security of the state of the s The state of the s Section 1. The sectio Salamangulatus (austrialia) [Salamangulatus (austrialia) [Salama Immedia of the same states of the same form of the same for the same form of the same form of the same form of the same for the same form of the same for the same form of the same form of the same form of the same for the same form of the same form of the same form of the same for the same form of the same form of the same form of the same for the same form of the same form of the same form of the same for the same form of the same form of the same form of the same for the same form of the same form of the same form of the same for the same form of the same for the same form of the same form of Specimes (and the service of the ser See 1 Compare money control () (1) The state of t promote by death of sample in community of the community the Planty or Committee of the Committee General Annual Conference of C The second secon (charactery private) (characte I will be a second or seco Speciment (COS) Section (1) and the sectio The control of the co The state of the s Application of the control of the co Comparison of the comparison of the command of the principal of all the principal of a strong to the strong to the selected strong (Mining Selected Selec position of the control of the contr The second secon The state of the s The state of the s The state of the s (of frameworks of framework (1 t t) Finds and Finds and Finds (and Fin Haber Light Annual (John Maria et al. Haber Light Annual (John Maria) (Haber Light Annual (Haber Light Ann A to it consists the part of the state of th The state of the s principles (season) address described (plantically) (delaw Augusted solder attack principles (plantically) And the state of t Communication of the control of the | Marine principal contention or representation of the content of The state of the s Section and commerce specialization for configuration and commerce specialization and EDespite of a more type completions (more RE) (soft (if your frame) (place "more") (maps are passed are pass



Scheme's map with meager error checking

```
(define (map f 11 . ls)
  (unless (procedure? f)
    (error 'map "expected a procedure"))
  (unless (procedure-arity-includes? f (+ (length ls) 1))
    (error 'map "bad arity"))
  (do-map f (cons 11 ls)))
(define (do-map f lss)
  (cond
    [(andmap null? lss) null]
    [(andmap pair? lss)
     (cons (apply f (map1 car lss))
            (do-map f (map1 cdr lss)))]
    [else (error 'map "bad lists")]))
(check-equal? (map add1 '()) '())
(check-equal? (map add1 '(1 2 3)) '(2 3 4))
(check-equal? (map + '(1 2 3) '(4 5 6)) '(5 7 9))
(check-exn \#rx"map" (\lambda () (map 1 2)))
(check-exn \#rx"map"(\lambda () (map(\lambda (x) x) 2)))
(check-exn \#rx"map" (\lambda () (map (\lambda (x y) x) (list 2))))
(check-exn \#rx"map" (\lambda () (map (\lambda (x) x) (cons 1 (cons 2 \#f)))))
```

Bad errors are everywhere

```
Example program good errors

(list->string (list 1)) 62%

(caddr (cons 1 #f)) 37%

(map (lambda (x y) x) | 12%

(list 1))
```

8 Implementations tried: Bigloo 3.4a, Chicken 4.3.0, Gambit 4.6.0, Guile 1.8, Ikarus 0.0.3, Larceny v0.97, Petite Chez 8.0, Scheme 48 1.8

Lessons:

- Writing error checking code by hand is error-prone
- Mixing the checks into the code makes them hard to extract for clients

[Meyer'92]

Contracts warmup

warmup/client1.rkt

```
#lang racket
(require "ep.rkt")
(e2p 2)
```

Introducing the notation, i:

- Each box is a module with a filename, requires, provides, and a body
- Provide/contract dictates the contracts on exported variables
- Server color scheme on left; client color scheme on right

warmup/client1.rkt

```
#lang racket
(require "ep.rkt")
(e2p 2)
```

Introducing the notation, ii:

- The function -> builds an arrow contract from domain and range contracts
- Predicates can be used directly as contracts

warmup/client1.rkt #lang racket (require "ep.rkt") (e2p 2)

What is the answer?

warmup/ep.rkt

warmup/client1.rkt

```
#lang racket
(require "ep.rkt")
(e2p 2)
```

2

```
warmup/client2.rkt
#lang racket
(require "ep.rkt")
(e2p 1)
```

What is the answer?

warmup/client2.rkt

```
#lang racket
(require "ep.rkt")
(e2p 1)
```

```
warmup/ep.rkt:4.2:
    (file warmup/client2.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<even?>, given: 1
```

warmup/ep.rkt

warmup/client2.rkt

```
#lang racket
(require "ep.rkt")
(e2p 1)
```

warmup/ep.rkt:4.2:
 (file warmup/client2.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<even?>, given: 1

source loc of the contract

warmup/ep.rkt #lang racket

warmup/client2.rkt

```
#lang racket
(require "ep.rkt")
(e2p 1)
```

```
warmup/ep.rkt:4.2:
    (file warmup/client2.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<even?>, given: 1
```

violator of the contract

warmup/ep.rkt #lang racket

warmup/client2.rkt

```
#lang racket
(require "ep.rkt")
(e2p 1)
```

```
warmup/ep.rkt:4.2:
    (file warmup/client2.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<even?>, given: 1
```

description of the violation

warmup/client3.rkt #lang racket (require "ep.rkt")

(e2p -2)

What is the answer?

warmup/ep.rkt

warmup/client3.rkt

```
#lang racket
(require "ep.rkt")
(e2p -2)
```

```
warmup/ep.rkt:4.2:
    (file warmup/ep.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<positive?>, given: -2
```

warmup/client4.rkt #lang racket (require "ep.rkt")

 $(e^2p - 1)$

What is the answer?

warmup/ep.rkt

warmup/client4.rkt

```
#lang racket
(require "ep.rkt")
(e2p -1)
```

```
warmup/ep.rkt:4.2:
   (file warmup/client4.rkt)
broke the contract (-> even?
positive?) on e2p; expected
<even?>, given: -1
```

Lessons:

- Contracts govern the interaction between components and thus must come with proper blame assignment
- Contracts are optimistic; they insist only that there is an okay use context (compare types that insist every context is okay)

[Parnas'72], [Meyer'92]

Higher-order functions

```
rout/rout.rkt
#lang racket
(provide/contract
  [run-on-user-thread
    (-> (->* ()
                    #:pre (eq? (current-thread)
                                        user-thread)
                    any)
          void?)])
                           (define user-thread
                            (thread (\lambda () (let loop ()
                                 ((async-channel-get ach))
             (define ach (make-async-channel))
                                              (async-channel-put ach thunk))
```

More notation:

 The ->* contract packages the arguments between parens and allows pre- and post-conditions, plus rest arguments & other doodads

rout/rout.rkt

```
rout/client1.rkt
```

```
#lang racket
(require "rout.rkt")
(run-on-user-thread
  (\lambda () (read-case-sensitive #f)))
```

What is the answer?

rout/rout.rkt #lang racket (require racket/async-channel) [run-on-user-thread (-> (->* () #:pre (eg? (current-thread) (define ach (make-async-channel)) (define user-thread (thread (λ () (let loop () ((async-channel-get ach)) (loop))))) (define (run-on-user-thread thunk) (async-channel-put ach thunk))

```
rout/client1.rkt

#lang racket
  (require "rout.rkt")
  (run-on-user-thread
     (λ () (read-case-sensitive #f)))
```

Silence: no error

rout/rout-broken.rkt

rout/client2.rkt

```
#lang racket
(require "rout-broken.rkt")
(run-on-user-thread
  (\lambda () (read-case-sensitive #f)))
```

What is the answer?

```
rout/rout-broken.rkt:3.2:
   (file rout/rout-broken.rkt)
broke the contract
   (-> (->* () #:pre ... any) void?)
on run-on-user-thread; #:pre violation
```

Blame rightly falls on rout-broken.rkt ... but it is the argument contract that fails!

How does that work?

Blame for functions

```
f : (A? -> B?) -> (C? -> D?)
```

Lets abstract for a moment. For each of A?, B?, C?, and D?, who should be blamed: f or f's caller?

Blame for functions

```
f: (A? -> B?) -> (C? -> D?)
f: (A? -> B?) × C? -> D?
```

These two as types are the same, right? Same for contracts. So, D? is the final result of f and thus f's responsibility

Blame for functions

```
f : (A? -> B?) -> (C? -> D?)
```

Lets mark **D?** with a + to indicate this position is **f**'s responsibility

When **f**'s argument is invoked, **f** is the one invoking it, so it must be that **f** is responsible for those inputs

Returning to the uncurried version C? is like an input, and thus f's caller's responsibility

```
f: (A? -> B?) -> (C? -> D?)
```

Lets mark C? with a – to indicate this position is f's caller's responsibility

```
f: (A? -> B?) -> (C? -> D?)
```

Finally, although **f**'s caller is not in direct control when the argument function returns, **f**'s caller is the one that chose which function to supply, so has the ultimate responsibility for that function's results

Recognize this pattern? Standard function-space contravariance! Count how many times you go left of the arrow. Odd \Rightarrow caller's fault; even \Rightarrow function's fault

Lesson:

• Higher-order values complicate the logical boundary between modules and thus blame assignment

[Findler, Felleisen'02], [Findler, Blume'06]

Performance

```
#lang racket
(struct node (num left right))
(define (bt? x)
   (or (node? x) (null? x)))
(define (find bst m)
   (match bst
     ['() #f]
     [(node n left right)
                                  (provide/contract
       (cond
                                  [node
  (-> real? bt? bt? bt?)]
                                   (-> bt? real? boolean?)]
         [(= n m) #t]
                                  [unmarshall
                                  (-> input-port? bt?)]
         [(> n m)]
          (find left m)]
         [ (< n m)
           (find right m)])))
(define (unmarshall port) '...)
(define (marshall bt port)
```

```
#lang racket
                     (define (unmarshall port) '...)
                     (define (marshall bt port)
(or (node? x) (null? x)))
            [(node n left right)
             (cond
             [(= n m) #t]
             (find left m)]
(provide/contract
  [bt?
   (-> any/c boolean?)]
  [node
   (-> real? bt? bt?)]
  [find
   (-> bt? real? boolean?)]
  [unmarshall
   (-> input-port? bt?)]
  [marshall
   (-> bt? output-port? void?)])
```

```
#lang racket
                     (define (unmarshall port) '...)
                     (define (marshall bt port)
(or (node? x) (null? x)))
            [(node n left right)
            (cond
             (find left m)]
(provide/contract
 [bt?
   (-> any/c boolean?)]
 Inode
   (-> real? bt? bt?)]
 [find
   (-> bt? real? boolean?)]
 [unmarshall
   (-> input-port? bt?)]
 [marshall
   (-> bt? output-port? void?)])
```

bt/bt-client.rkt

```
#lang racket
(require "bt.rkt")
(find (node
       null
        (node
        -5
         (node
         null
         null)
        null))
```

```
#lang racket
                     (define (unmarshall port) '...)
                     (define (marshall bt port)
(or (node? x) (null? x)))
            [(node n left right)
            (cond
             (find left m)]
(provide/contract
 [bt?
   (-> any/c boolean?)]
 Inode
   (-> real? bt? bt?)]
 [find
   (-> bt? real? boolean?)]
 [unmarshall
   (-> input-port? bt?)]
 [marshall
   (-> bt? output-port? void?)])
```

bt/bt-client.rkt

```
#lang racket
(require "bt.rkt")
(find (node
       null
        (node
        -5
         (node
         null
         null)
        null))
```

What is the answer?

```
#lang racket
                     (define (unmarshall port) '...)
                     (define (marshall bt port)
(or (node? x) (null? x)))
            [(node n left right)
            (cond
             (find left m)]
(provide/contract
 [bt?
   (-> any/c boolean?)]
 [node
   (-> real? bt? bt?)]
 [find
   (-> bt? real? boolean?)]
 [unmarshall
   (-> input-port? bt?)]
 [marshall
   (-> bt? output-port? void?)])
```


What is the answer?

```
#lang racket
                      (define (unmarshall port) '...)
(define (bt? x)
(or (node? x) (null? x)))
                      (define (marshall bt port)
             [(node n left right)
             (cond
              (find left m)]
(provide/contract
 [bt?
   (-> any/c boolean?)]
  [node
   (-> real? bt? bt?)]
  [find
   (-> bt? real? boolean?)]
  [unmarshall
   (-> input-port? bt?)]
  [marshall
   (-> bt? output-port? void?)])
```

bt/bt-client.rkt #lang racket

```
#lang racket
(require "bt.rkt")
(find 3
4)
```

#f

The contract violation has gone undected!

... but it isn't hard to implement a bst? predicate

```
#lang racket
(define (bst/bet? x lo hi)
  (match x
    ['() #t]
    [(node n left right)
     (and
      (<= lo n hi)
      (bst/bet? left lo n)
      (bst/bet? right n hi))]
    [else #f]))
(define (bst? x)
  (bst/bet? x - inf.0 + inf.0)
(provide/contract
 [bt? (-> any/c boolean?)]
 [node (-> real? bt? bt?)]
 [find (-> bst? real? boolean?)])
```

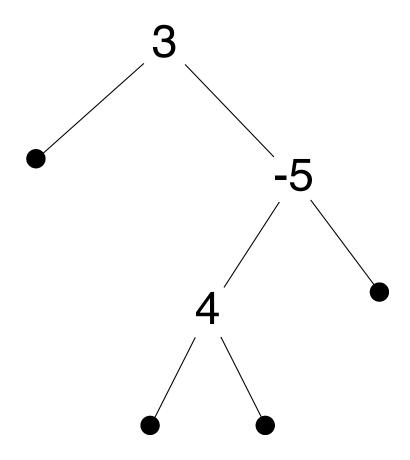
#lang racket (require "bst.rkt") (find 3

What is the answer?

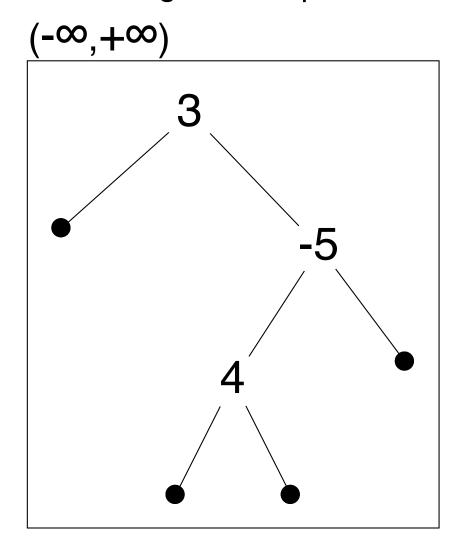
```
bt/bst.rkt:27.2:
    (file bt/bst-client.rkt)
broke the contract (-> bst?
real? boolean?) on find;
expected <bst?>, given:
#<node>
```

... but this contract changes **find**'s complexity from O(log(n)) to O(n)!

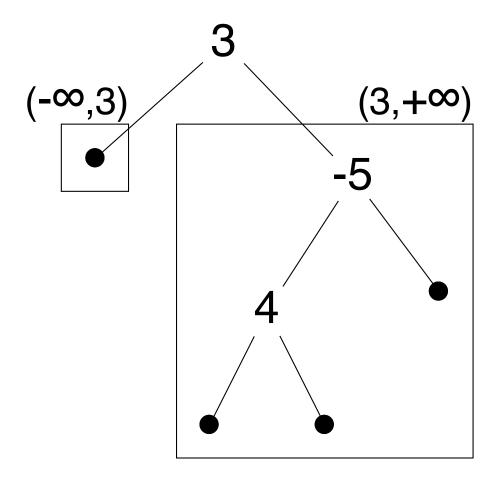
Idea: check lazily, as the program explores the tree



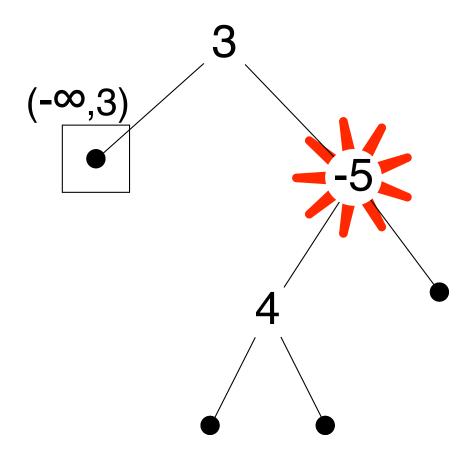
Box the portion remaining to check, put bounds on the box



Move the boxes as the data structure is explored

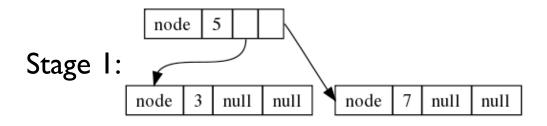


Discover violations while moving boxes

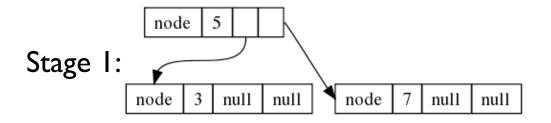


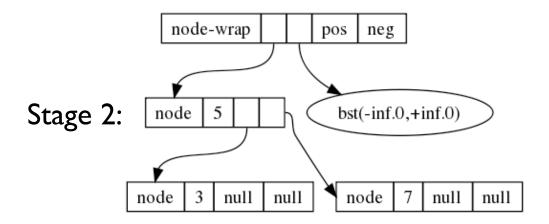
Implementation: use wrapper structs and intercept field selection operations to adjust the wrappers

I: An unadorned binary search tree

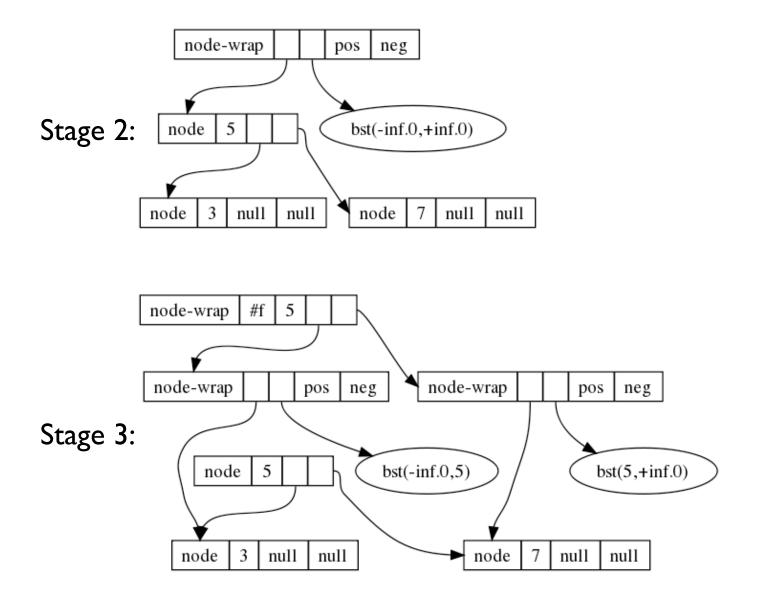


2: To add a contract, create a wrapper that records the bounds and the parties to be blamed

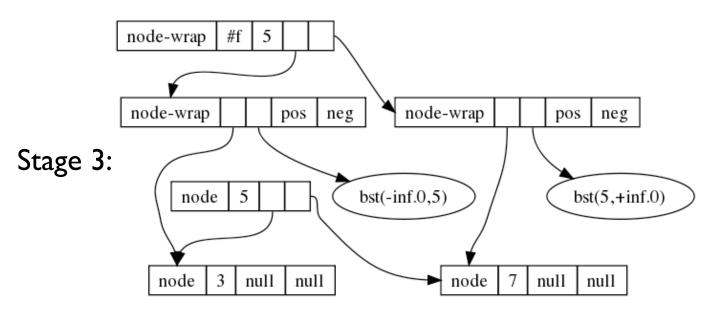


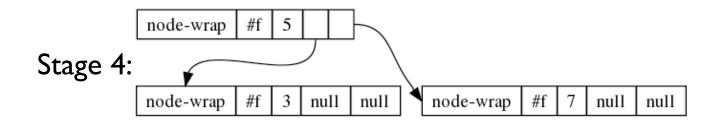


3: When selecting a field, mutate the wrapper into the original struct and create new wrappers inside



4: When fully explored, all that is left is morphed wrappers





bt/bst-lazy.rkt #lang racket (contract-struct node (num left right)) (define (bst/c low high) (or/c null? (node/dc [num (between/c low high)] [left (num) (bst/c low num)] [right (num) (bst/c num high)]))) (provide/contract [bt? (-> any/c boolean?)] [node (-> real? bt? bt?)] [find (-> (bst/c -inf.0 +inf.0) real? (or/c any/c #f))])

```
bt/bst-lazy.rkt:23.2:
  (file bt/bst-lazy-client.rkt)
broke the contract
  (->
   (or/c
    (node/dc
     (num (between/c -inf.0 +inf.0))
     (left ...)
     (right ...))
    null?)
   real?
   (or/c any/c #f))
on find; expected <(>=/c 3)>, given: -5
```

Lesson:

 Some contracts should be checked lazily to avoid changing the program's asymptotic compexity

[Chitil, McNeil, Runciman'03], [Hinze, Jeuring, Löh,'06], [Guo, Findler, Rogers'07]

Dependent contracts

Reminder: the evolution of contract arrow syntax

First just arguments and results:

```
(-> ctc-expr ...
ctc-expr)
```

Reminder: the evolution of contract arrow syntax

then pre- and post-conditions (plus other dodads we don't need):

```
(->* (ctc-expr ...)
#:pre bool-expr
ctc-expr
#:post bool-expr)
```

Reminder: the evolution of contract arrow syntax

finally, dependent contracts:

```
(->i ([arg-id (id ...) ctc-expr] ...)
    #:pre (id ...) bool-expr
    [res-id (id ...) ctc-expr]
    #:post (id ...) bool-expr)
```

```
indy/ctc.rkt
#lang racket
(require (planet cce/fasttest:3/random))
(provide deriv/c)
(define deriv/c
  (->i ([f () (-> real? real?)]
         [\delta () real?])
       [fp () (-> real? real?)]
       #:post (f \delta fp)
       (for/and ([i (in-range 0 100)])
          (define x (random-number))
          (define slope (/ (- (f (+ x 0.01)))
                                (f (-x 0.01))
                             (* 2 0.01)))
          (<= (abs (- slope (fp x))) \delta))))
```





indy/client.rkt

```
#lang racket
(require "deriv.rkt")
(define 2x
    (deriv sqr 0.01))
(2x 20)
```

What is the answer?

```
indy/deriv.rkt:4.2:
    (file indy/ctc.rkt)
broke the contract
    (->i
        ((f () ...) (δ () ...))
        (fp () ...)
        #:post
        (f δ fp)
        ...)
on deriv; expected <real?>, given:
-0.9015191986644407+13.564102564102564i
```





```
indy/ctc.rkt
#lang racket
(require (planet cce/fasttest:3/random))
(provide deriv/c)
(define deriv/c
  (->i ([f () (-> real? real?)]
         [\delta () real?])
       [fp () (-> real? real?)]
       #:post (f \delta fp)
       (for/and ([i (in-range 0 100)])
          (define x (random-number))
          (define slope (/ (- (f (+ x 0.01)))
                                (f (-x 0.01))
                             (* 2 0.01)))
          (<= (abs (- slope (fp x))) \delta))))
```





Lesson:

• Contracts are code too, and thus can crash

[Findler, Felleisen'02], [Ou, Tan, Mandelbaum, Walker'04], [Blume, McAllester'04], [Greenberg, Pierce, Weirich,'10]

Overall lessons

Thinking hard about blame clarifies contract checking

Each new aspect of a programming language demands its own form of contract checking

Thank you

Thanks also to Amal Ahmed, Matthias Blume, Christos Dimoulas, Matthias Felleisen, Matthew Flatt, Shu-yu Guo, Mario Latendresse, Jacob Matthews, Anne Rogers, Jeremy Siek, Stevie Strickland, Sam Tobin-Hochstadt, and Phil Wadler.