Contracts in Racket

Robby Findler
PLT & Northwestern
An exercise: map
Simple map

(define (map f l)
  (cond
   [(null? l) '()]  
   [else (cons (f (car l))
             (map f (cdr l)))])))
Simple map, with tests

```
(define (map f l)
  (cond
   [(null? l) '()]  
   [else (cons (f (car l))
                 (map f (cdr l)))]))

(check-equal? (map add1 '()) '())
(check-equal? (map add1 '(1 2 3)) '(2 3 4))
```
Scheme’s map

(define (map1 f l)
  (cond
    [(null? l) '()]
    [else (cons (f (car l))
                (map1 f (cdr l)))]))

(define (map f l1 . ls)
  (cond
    [(null? l1) null]
    [else
     (cons (apply f (map1 car (cons l1 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))
\[(\text{map} \ (\lambda \ (x) \ x) \ 1)\]

car: expects argument of type \(<\text{pair}>\); given \(I\)
framework/private/text.rkt: occurrences of car
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" (λ () (map 1 2)))
(check-exn #rx"map" (λ () (map (λ (x) x) 2)))
(check-exn #rx"map" (λ () (map (λ (x y) x) (list 2))))
(check-exn #rx"map" (λ () (map (λ (x) x) (cons 1 (cons 2 #f)))))
Bad errors are everywhere

<table>
<thead>
<tr>
<th>Example program</th>
<th>Impls. with good errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>(list-&gt;string (list 1))</td>
<td>62%</td>
</tr>
<tr>
<td>(caddr (cons 1 #f))</td>
<td>37%</td>
</tr>
<tr>
<td>(map (lambda (x y) x) (list 1))</td>
<td>12%</td>
</tr>
</tbody>
</table>

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
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
Introducing the notation, ii:

- The function \(-\rightarrow\) builds an arrow contract from domain and range contracts
- Predicates can be used directly as contracts
What is the answer?
```
#lang racket
(define (e2p x) x)
(provide/contract [e2p (-> even? positive?)])
```

```
#lang racket
(require "ep.rkt")
(e2p 2)
```
What is the answer?
#lang racket
(define (e2p x) x)
(provide/contract [e2p (-> even? positive?)])

#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
#lang racket
(define (e2p x) x)
(provide/contract [e2p (-> even? positive?)])

#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

#lang racket
(define (e2p x) x)
(provide/contract [e2p (-> even? positive?)])

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

warmup/client2.rkt

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

violer of the contract
```
#lang racket
(define (e2p x) x)
(provide/contract
  [e2p (-> even? positive?)])
```

```
#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
Define a function `e2p` which takes a number `x` and returns `x` itself.

Provide the contract for `e2p`:

- `(e2p -> even? positive?)`

What is the answer?

In `client3.rkt`:

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

What is the answer?
#lang racket
(define (e2p x) x)
(provide/contract
  [e2p (-> even? positive?)])

#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
What is the answer?
#lang racket
(define (e2p x) x)
(provide/contract [e2p (-> even? positive?)])

#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?)])

More notation:
• The ->* contract packages the arguments between parens and allows pre- and post-conditions, plus rest arguments & other doodads
#lang racket

(require racket/async-channel)

(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/rout.rkt

```racket
#lang racket
(require racket/async-channel)
(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

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

What is the answer?

40
rout/rout.rkt

```racket
#lang racket
(require racket/async-channel)
(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

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

Silence: no error
rout/rout-broken.rkt

#lang racket
(provide/contract
  [run-on-user-thread
   (-> (->* ()
       #:pre (eq? (current-thread) user-thread)
       any)
   void?)])

(define (run-on-user-thread thunk) (thunk))
(define user-thread (thread (λ () 1)))

rout/client2.rkt

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

What is the answer?

42
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

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

Let's abstract for a moment. For each of A?, B?, C?, and D?, who should be blamed: \( f \) or \( f \)'s caller?
Blame for functions

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]
\[ f : (A? \rightarrow B?) \times C? \rightarrow D? \]

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

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

Let's mark \( D? \) with a + to indicate this position is \( f \)'s responsibility
Blame for functions

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

When \( f \)'s argument is invoked, \( f \) is the one invoking it, so it must be that \( f \) is responsible for those inputs.
Blame for functions

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

\[ f : (A? \rightarrow B?) \times C? \rightarrow D? \]

Returning to the uncurried version \( C? \) is like an input, and thus \( f \)'s caller's responsibility
Blame for functions

\[ + \quad - \quad + \]

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

Let's mark \( C? \) with a \(-\) to indicate this position is \( f \)'s caller's responsibility.
Blame for functions

\[ f : (A? \to B?) \to (C? \to D?) \]

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.
Blame for functions

\[ f : (A? \rightarrow B?) \rightarrow (C? \rightarrow D?) \]

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) 
      (cond
        [(= n m) #t]
        [(> n m)
          (find left m)]
        [(< n m)
          (find right m)]])])
(define (unmarshall port) '...)
(define (marshall bt port) '...)

provide/contract
[bt? [-> any/c boolean?]]
[node [-> real? bt? bt? bt?]]
[find [-> bt? real? boolean?]]
[unmarshall [-> input-port? bt?]]
[marshall [-> bt? output-port? void?]]
#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)
      (cond
        [(= n m) #t]
        [(> n m) (find left m)]
        [(< n m) (find right m)]])))

(define [unmarshall port]
  '(...)

(define [marshall bt port]
  '(...)

(provide/contract
  [bt? 
   (-> any/c boolean?)])

[find
  (-> bt? real? boolean?)])

[unmarshall
  (-> input-port? bt?)])

[marshall
  (-> bt? output-port? void?)])
#lang racket

(provide/contract
  [bt? 
    (-> any/c/c boolean?)]

[node
  (-> real? bt? 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
  3
  null
  (node
    -5
    (node
      4
      null
      null)
    null))
  4)
bt/bt.rkt

#lang racket

(provide/contract
  [bt? 
    (-> any/c boolean?)])

[node
  (-> real? bt? 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 3
      null
      (node -5
            null
            null
            null))
      4)

What is the answer?
(provide/contract
  [bt? (-> any/c c boolean?)]
  [node (-> real? bt? bt? bt?)])
[find (-> bt? real? boolean?)]
[unmarshall (-> input-port? bt?)]
[marshall (-> bt? output-port? void?)])
bt/bt.rkt

#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) (cond [(= n m) #t] [(> n m) (find left m)] [(< n m) (find right m)] )])

bt/bt-client.rkt

#lang racket

(require "bt.rkt")
(find
  3
  -5
  4
  #f)
The contract violation has gone undected!

... but it isn’t hard to implement a bst? predicate
bt/bst.rkt
#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? bt? bt?)]
  [find (-> bst? real? boolean?)])

bt/bst-client.rkt
#lang racket
(require "bst.rkt")
(find 4 -5 3 4)

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

\((-\infty, +\infty)\)
Move the boxes as the data structure is explored
Discover violations while moving boxes

4

-5

3

(-∞,3)

4

76
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

Stage 1:

Stage 2:
3: When selecting a field, mutate the wrapper into the original struct and create new wrappers inside

Stage 2:

Stage 3:
4: When fully explored, all that is left is morphed wrappers

Stage 3:

Stage 4:
#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? 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 complexity

[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:

\((-\rightarrow \text{ctc-expr} \ldots \text{ctc-expr})\)
Reminder: the evolution of contract arrow syntax

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

\[
(- > * \ (ctc\text{-}expr \ldots) \\
\text{#:pre bool\text{-}expr} \\
tc\text{-}expr \\
\text{#:post bool\text{-}expr})
\]
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?)]
      [δ () real?])
        [fp () (-> real? real?)])
#:post (f δ 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))) δ))))
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
```racket
#lang racket
(require (planet cce/fasttest:3/random))
(provide deriv/c)
(define deriv/c
  (->i ([f () (-> real? real?)])
    [δ () real?])
  [fp () (-> real? real?)])
#:post (f δ 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)) δ)))
```
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.