CONTRACTS
SEMANTICS & PRAGMATICS

Matthias Felleisen. PLT
NU PRL

What a Great Half-time Show
Congratulations, Pierre-Louis

What a Great Half-time Show
Congratulations, Pierre-Louis

And it has been way too long,
at the opening workshop of PPS,
with a talk on contracts for ho languages

Contracts: Logical Assertions in Component Interfaces

- statically checkable, e.g., types
- behavioral assertions, typically checked at run-time
- temporal assertions, e.g., race conditions, sequencing
- quality of service promises, e.g., # of requests handled

Contracts: Beyond Types

impose *obligations*

make *promises*

```
@ensure x >= 0.0
@require sqrt >= 0.0
// compute the square root of x
double sqrt(double x) {
  ...
}
```

a little bit

Contracts: Beyond Types

impose *obligations*

make *promises*

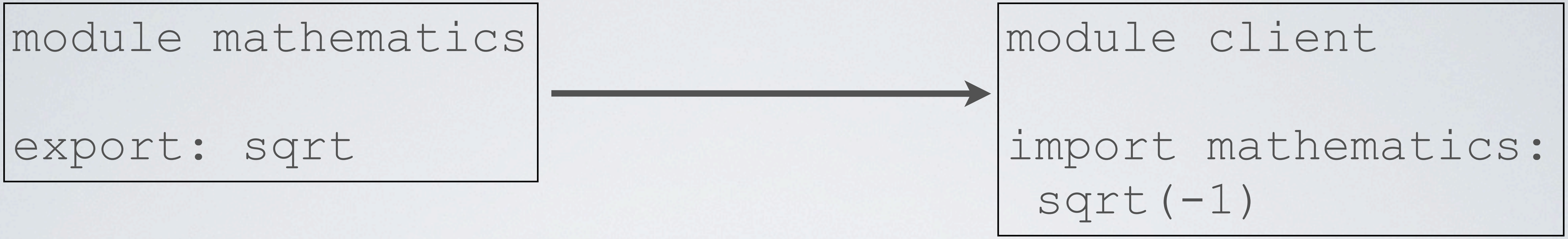
```
@ensure x >= 0.0
@require sqrt >= 0.0
// compute the square root of x
double sqrt(double x) {
   ...
}
```

a little bit

```
@ensure x >= 0.0
@require abs(x - sqrt * sqrt)< EPSILON
// compute the square root of x
double sqrt(double x) {
   ...
}
```

a lot more

Contracts: Pointing Fingers



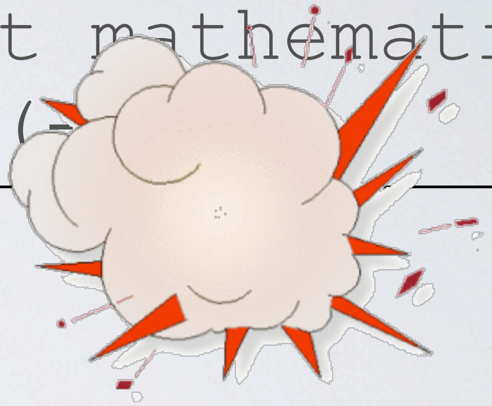
```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(-1)
```

Contracts: Pointing Fingers

```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(-
```



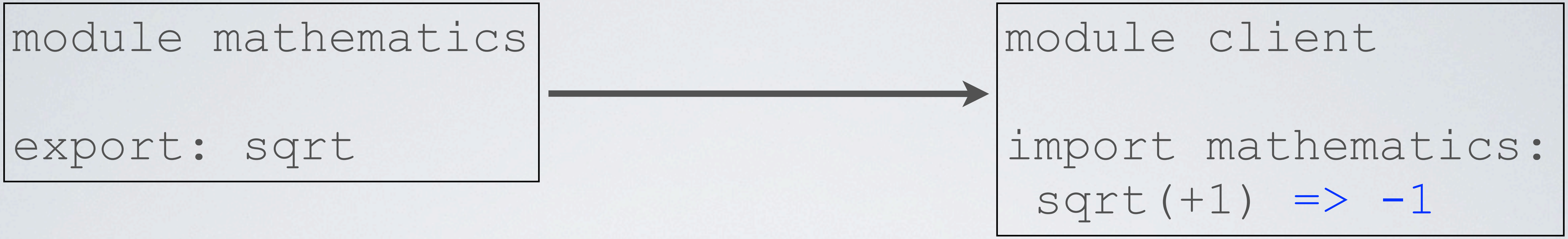
Contracts: Pointing Fingers

```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(-
```

client failed to *oblige*, blame *client*

Contracts: Pointing Fingers



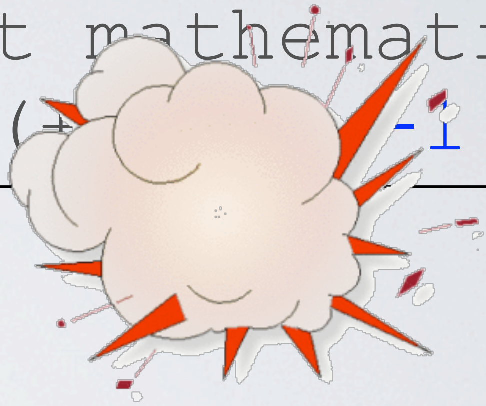
```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(+1) => -1
```

Contracts: Pointing Fingers

```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(+ [illegible] -1
```



Contracts: Pointing Fingers

```
module mathematics

export: sqrt
```

```
module client

import mathematics:
 sqrt(+        -1
```

mathematics broke *promise,*
blame *mathematics*

Contracts: Beyond Types

```
class ModuleBrowser {

  @require window.topFrame().isTopLevel()
  void mouseHandler(Event e, Posn p) {
    ...
  }
}
```

state of the world

Contracts: Beyond Types

```
class Turn {

  private void setPlaced() {...}
  private boolean isPlacedSet() {...}
  ...

  @require isMergerPosition(p,myBoard())
  @ensure setPlacementCalled()
  void placeHotel(Hotel h, Position p) {
    ...
  }

  @require isPlacedSet()
  Decision retrieveDecision() {
    ...
  }
}
```

simple temporal

Contracts: Beyond Types

```
class Turn {

  private void setPlaced() {...}
  private boolean isPlacedSet() {...}
  ...

  @require isMergerPosition(p,myBoard())
  @ensure setPlacementCalled()
  void placeHotel(Hotel h, Position p) {
    ...
  }

  @require isPlacedSet()
  Decision retrieveDecision() {
    ...
  }
}
```

simple temporal

Contracts: Beyond Types

```
(provide
 (contract-out
  ;; Nat Nat [Nat] [Nat] [[Listof X]] [[Listof Y]] ->* Image [Nat Nat ->* X Y]
  ;; (grid n m x-delta y-delta x-labels y-labels) creates:
  ;; -- a graphical grid of size (* n x-delta) x (* m y-delta)
  ;;    divvied up into n x m tiles of size x-delta x y-delta
  ;;    labeled with x-labels x y-labels
  ;; -- a function event-handler that deals with mouse clicks for this grid
  ;; (event-handler x y) = (an-x-lablel, a-y-label)
  ;; iff (x,y) is a mouse click in the tile labeled (an-x-lablel, a-y-label)
  (grid (->i ((n natural-number/c) (m natural-number/c))
             ((delta-x natural-number/c)
              (delta-y natural-number/c)
              (x-labels list?)
              (y-labels list?)
              #:grid? (grid? #t))
             #:pre/name (n x-labels) "correct # of x labels"
             (or (unsupplied-arg? x-labels) (= (length x-labels) n))
             #:pre/name (m y-labels) "correct # of y labels"
             (or (unsupplied-arg? y-labels) (= (length y-labels) m))
             (values
              (grid-scene image?)
              (event-handler
               (delta-x delta-y n m x-labels y-labels)
               (->i ((x natural-number/c) (y natural-number/c))
                    #:pre/name (x) "x in proper range"
                    (or (unsupplied-arg? delta-x) (< x (* n delta-x)))
                    #:pre/name (y) "y in proper range"
                    (or (unsupplied-arg? delta-y) (< y (* m delta-y)))
                    (values (xl (if (unsupplied-arg? x-labels) any/c (apply or/c x-labels)))
                            (yl (if (unsupplied-arg? y-labels) any/c (apply or/c y-labels)))))))))))
```

from code base

Contracts: Beyond Types

```
(provide
 (contract-out
  ;; Nat Nat [Nat] [Nat] [[Listof X]] [[Listof Y]] ->* Image [Nat Nat ->* X Y]
  ;; (grid n m x-delta y-delta x-labels y-labels) creates:
  ;; -- a graphical grid of size (* n x-delta) x (* m y-delta)
  ;;    divvied up into n x m tiles of size x-delta x y-delta
  ;;    labeled with x-labels x y-labels
  ;; -- a function event-handler that deals with mouse clicks for this grid
  ;; (event-handler x y) = (an-x-lablel, a-y-label)
  ;; iff (x,y) is a mouse click in the tile labeled (an-x-lablel, a-y-label)
  (grid (->i ((n natural-number/c) (m natural-number/c))
             ((delta-x natural-number/c)
              (delta-y natural-number/c)
              (x-labels list?)
              (y-labels list?)
              #:grid? (grid? #t))
             #:pre/name (n x-labels) "correct # of x labels"
             (or (unsupplied-arg? x-labels) (= (length x-labels) n))
             #:pre/name (m y-labels) "correct # of y labels"
             (or (unsupplied-arg? y-labels) (= (length y-labels) m))
             (values
              (grid-scene image?)
              (event-handler
               (delta-x delta-y n m x-labels y-labels)
               (->i ((x natural-number/c) (y natural-number/c))
                    #:pre/name (x) "x in proper range"
                    (or (unsupplied-arg? delta-x) (< x (* n delta-x)))
                    #:pre/name (y) "y in proper range"
                    (or (unsupplied-arg? delta-y) (< y (* m delta-y)))
                    (values (xl (if (unsupplied-arg? x-labels) any/c (apply or/c x-labels)))
                            (yl (if (unsupplied-arg? y-labels) any/c (apply or/c y-labels))))))))))
```

from code base

Contracts for Object-Oriented or Functional Languages

Eiffel is object-oriented.

Objects are legitimate values.

Why are there no contracts on objects?

Or contracts on higher-order functions?

Contracts for Object-Oriented or Functional Languages

Higher-order programming in Eiffel

```
interface IFunction {
  double apply(double x);
}
```

```
class Differentiator {
  @ensure ddx.isSlopeOfTangent()
  IFunction ddx(IFunction f) {
    ...
  }
}
```

Contracts for Object-Oriented or Functional Languages

Higher-order programming in Eiffel

```
interface IFunction {
  double apply(double x);
}
```

```
class Differentiator {
  ~~@ensure ddx.isSlopeOfTangent()~~
  IFunction ddx(IFunction f) {
    ...
  }
}
```

Contracts for Object-Oriented or Functional Languages

Eiffel **flattens** contracts

```
interface Function {
  double apply(double x);
}
```

```
interface IDifferentiated extends Function{
  @ensure isSlopeOfTangent(x)
  double apply(double x);
  boolean isSlopeOfTangent(double x)
}
```

```
class Differentiator {
  IDifferentiated d/dx(Function f) {
    ...
  }
}
```

Contracts for Object-Oriented or Functional Languages

Eiffel's approach relies on

-- nominal class types

-- duplication of types for contracts
e.g. int->int must exist once
per contract attached to D/R

-- closed world of software projects

Contracts for Object-Oriented or Functional Languages

Higher-order contracts in Racket, 1999--2002

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x)))))))])

(define (tangent? f x) ...)
```

Findler and Felleisen: ICFP 2002

Contracts for Object-Oriented or Functional Languages

Racket's approach imagines

-- an open, growing project

-- with *post-hoc* imposition of contracts on values

-- that is, a *structural* approach to imposing assertions

Contracts for Object-Oriented or Functional Languages

So what’s the big deal?

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x))))))])

(define (tangent? f x) ...)
```

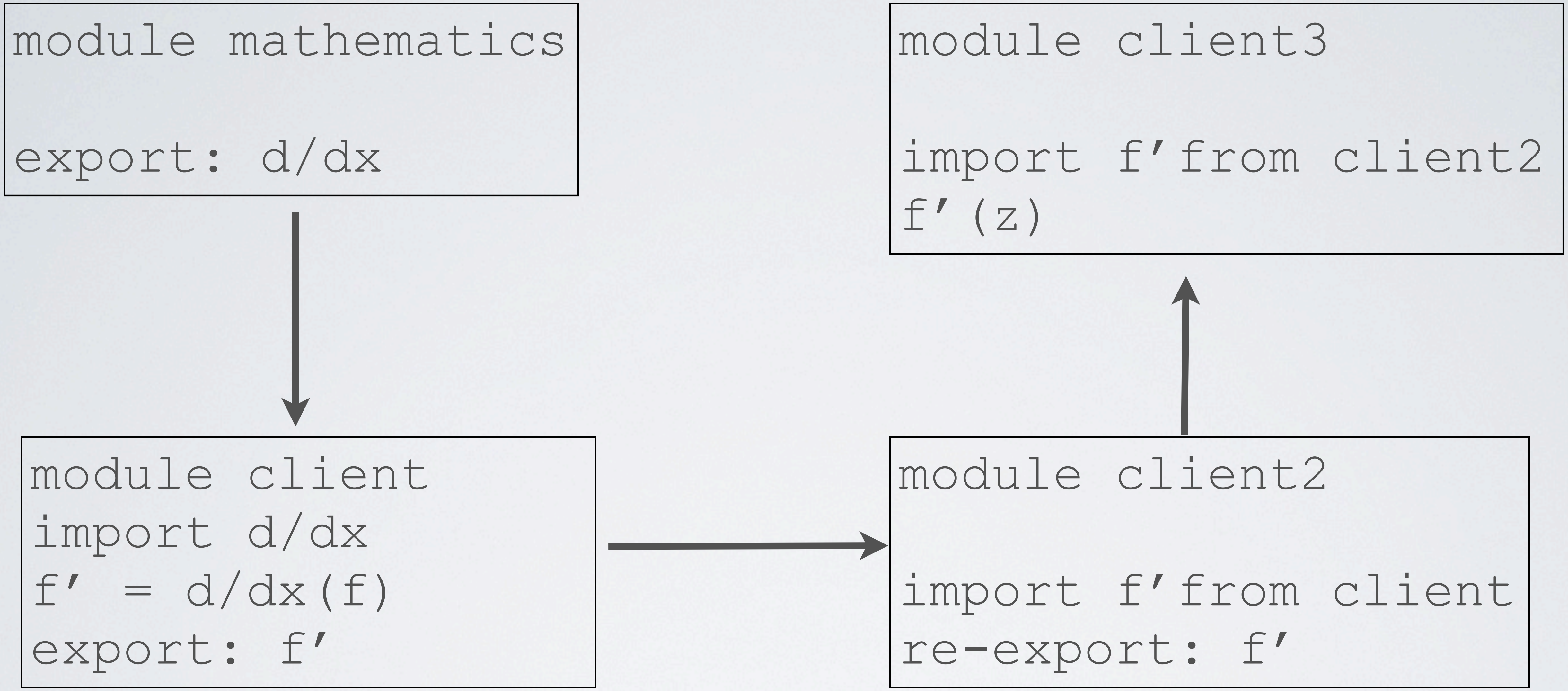
Contracts for Object-Oriented or Functional Languages

So what's the big deal?

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x))))))])

(define (tangent? f x) ...)
```

You cannot check this property when the function returns.

Contracts for Object-Oriented or Functional Languages



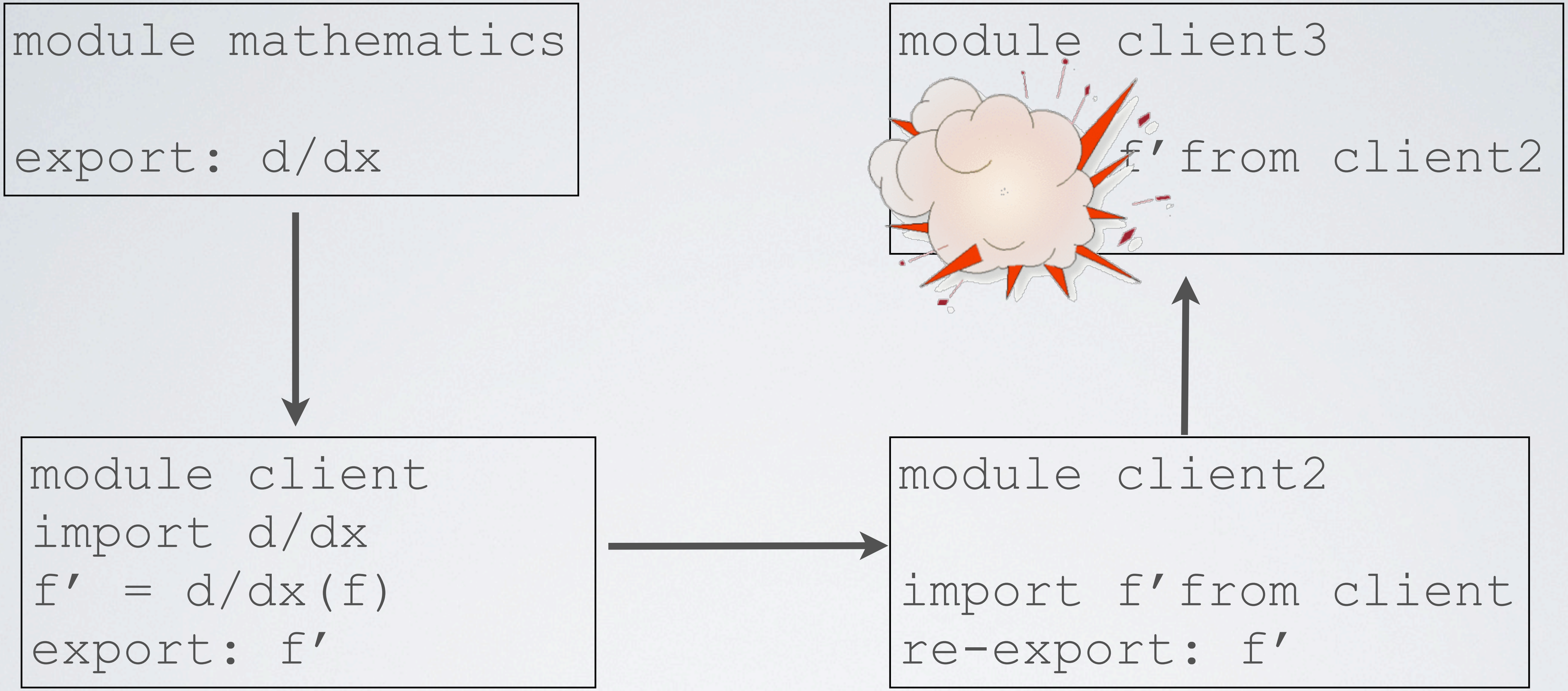
```
module mathematics

export: d/dx
```

```
module client
import d/dx
f' = d/dx(f)
export: f'
```

```
module client2

import f'from client
re-export: f'
```

```
module client3

import f'from client2
f'(z)
```

Contracts for Object-Oriented or Functional Languages



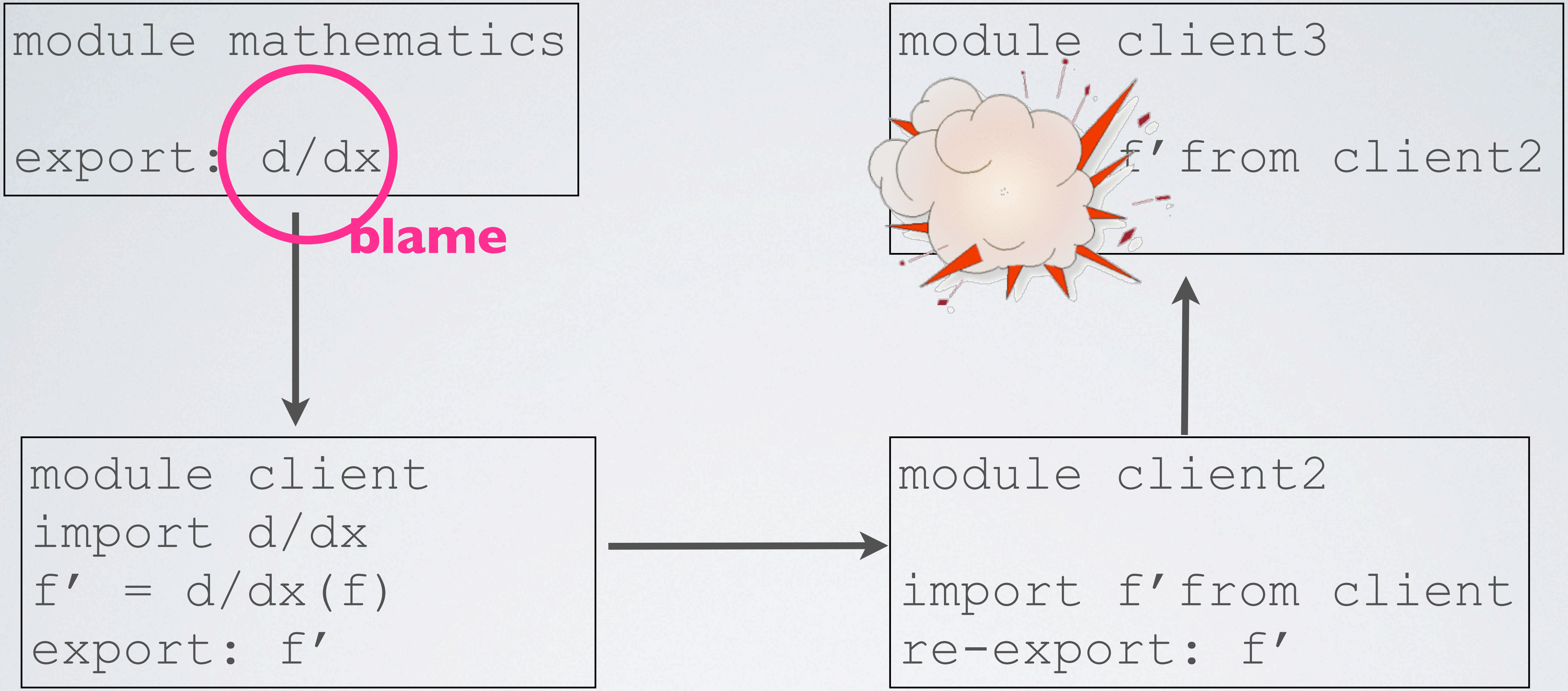
```
module mathematics

export: d/dx
```

```
module client
import d/dx
f' = d/dx(f)
export: f'
```

```
module client2

import f'from client
re-export: f'
```

```
module client3

f'from client2
```

Contracts for Object-Oriented or Functional Languages



```
module mathematics

export: d/dx
```

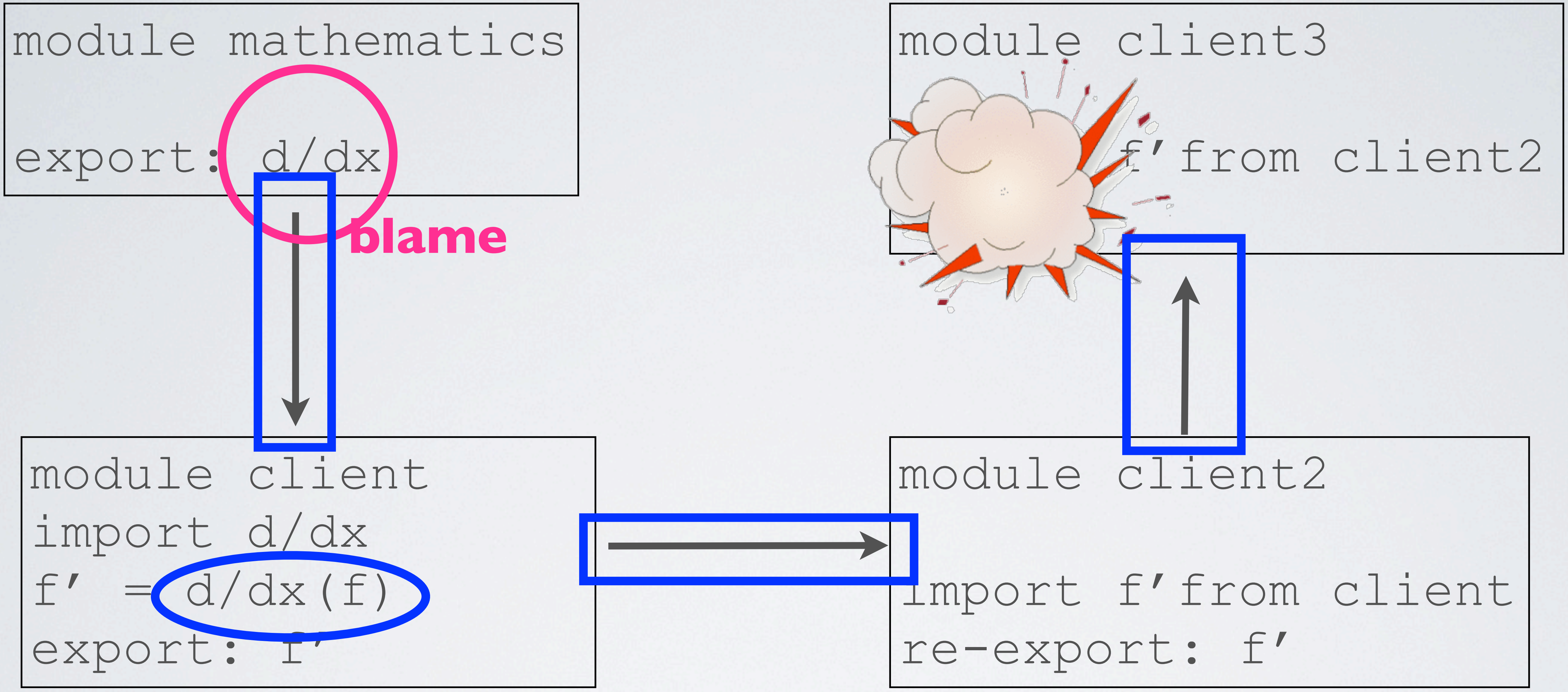
blame

```
module client3

f’ from client2
```

```
module client
import d/dx
f’ = d/dx(f)
export: f’
```

```
module client2

import f’from client
re-export: f’
```

Contracts for Object-Oriented or Functional Languages



```
module mathematics

export: d/dx
```

```
module client
import d/dx
f' = d/dx(f)
export: f'
```

```
module client3

f' from client2
```

```
module client2

import f' from client
re-export: f'
```

Wrap object with contract checker that carry along source information.

Contracts for Objects and Functions are Complicated

Semantics?

-- what do ho contracts mean?

-- what does blame mean?

Pragmatics!

-- is blame useful/correct?

-- is there “enough” blame?

SEMANTICS

Terms:

```
e = x | λx:t.e | e e | intf(c,e)| ...
t = b | t -> t
```

Terms:

interface

```
e = x | λx:t.e | e e | intf(c,e) | ...
t = b | t -> t
```

Terms:

interface

```
e = x | λx:t.e | e e | intf(c,e) | ...
t = b | t -> t
```

Contracts:

```
c = flat(λx:b.e) | c -> c | c -> λx:t.c
```

Terms:

interface

```
e = x | λx:t.e | e e | intf(c,e)| ...
t = b | t -> t
```

Contracts:

```
c = flat(λx:b.e) | c -> c | c -> λx:t.c
```

predicates

Terms:

interface

```
e = x | λx:t.e | e e | intf(c,e)| ...
t = b | t -> t
```

Contracts:

```
c = flat(λx:b.e) | c -> c | c -> λx:t.c
```

predicates

ho contracts

Terms:

interface

```
e = x | λx:t.e | e e | intf(c,e)| ...
t = b | t -> t
```

Contracts:

```
c = flat(λx:b.e) | c -> c | c -> λx:t.c
```

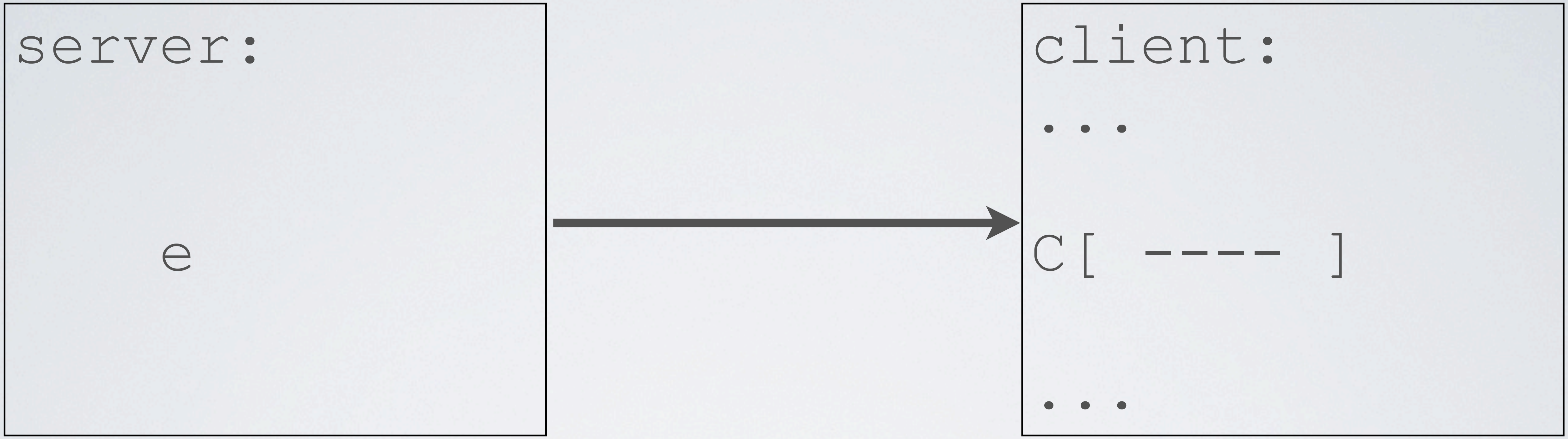
predicates

ho contracts

dependent

`C[intf(c,e)]`

C[intf(c,e)]



server:

e

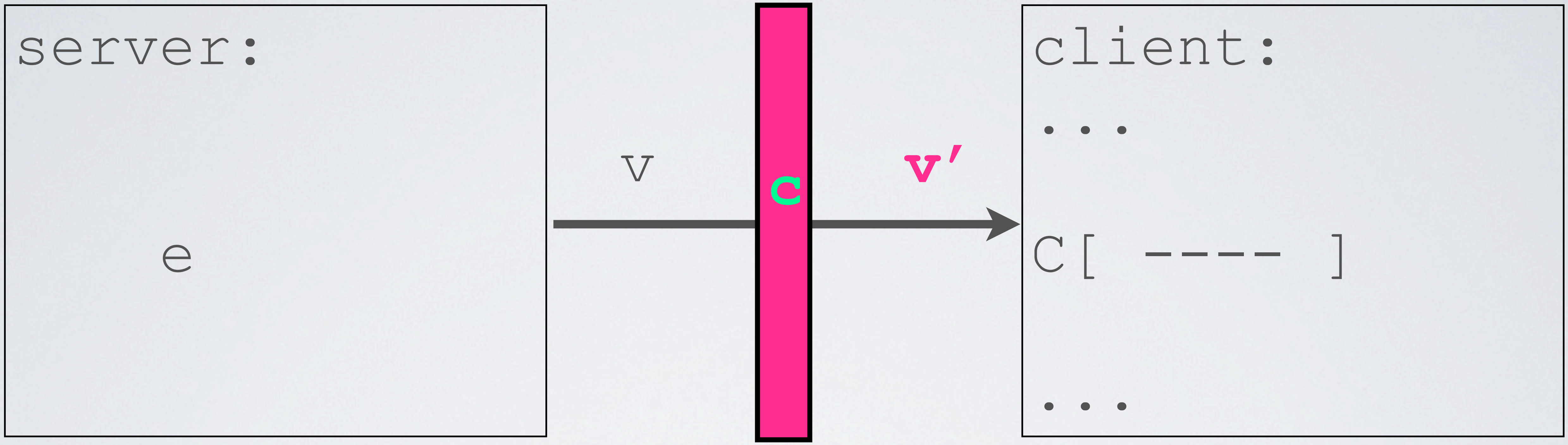
client:

...

C[----]

...

C[intf(c,e)]


server:

e

v

c

v′

client:

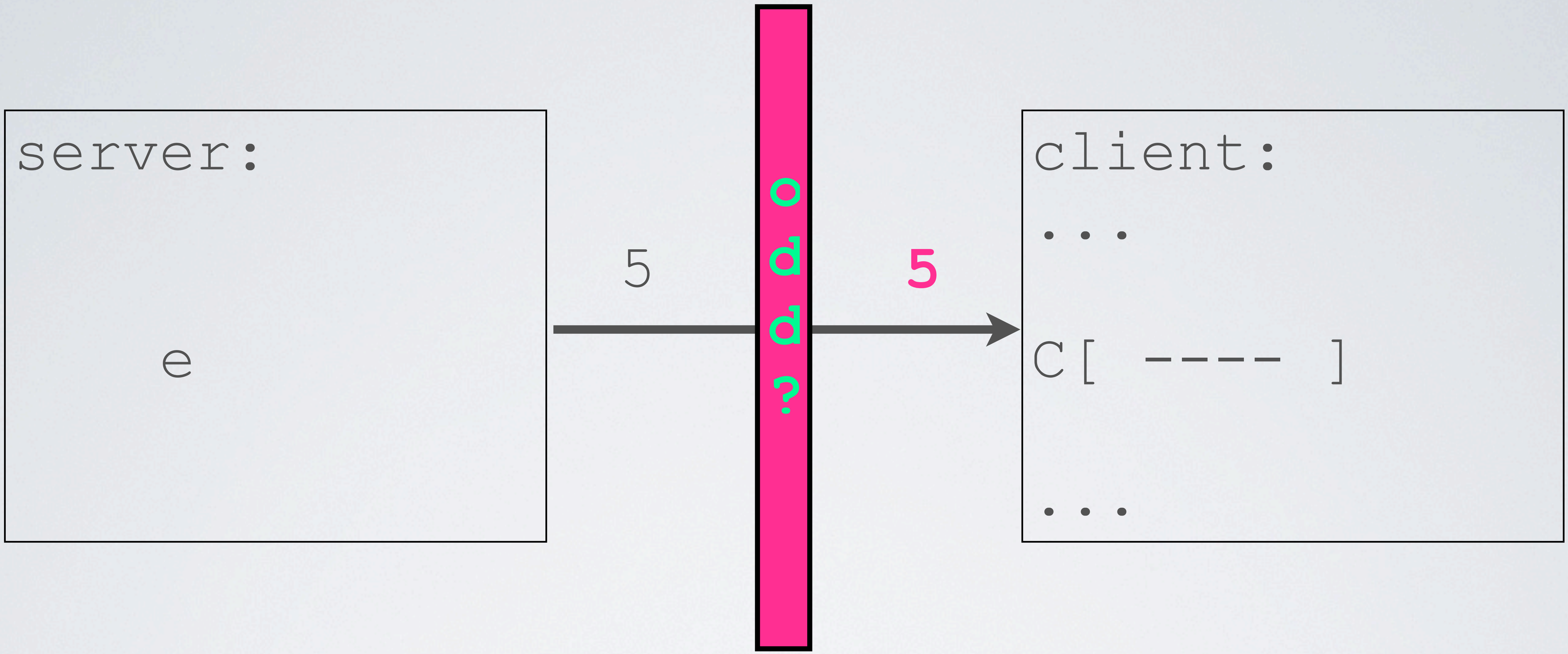
...

C[----]

...

Values flow through contracts, and contracts do their work.

`C[intf(odd?,5)]`



`server:`

`e`

`5`

`odd?`

`5`

`client:`

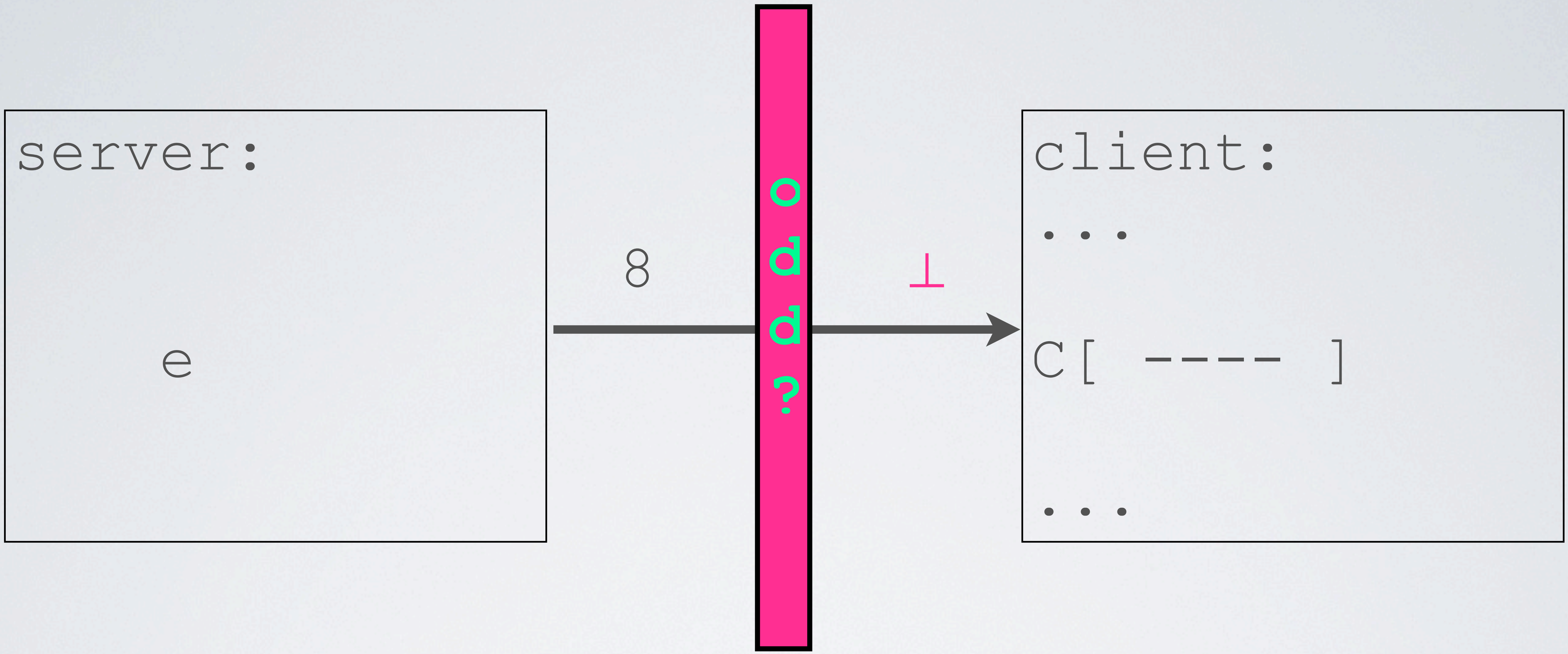
`...`

`C[ ---- ]`

`...`

For flat domains, check the value and let it thru if okay.

`C[intf(odd?,8)]`



`server:`

`e`

`8`

`odd?`

$\perp$

`client:`

`...`

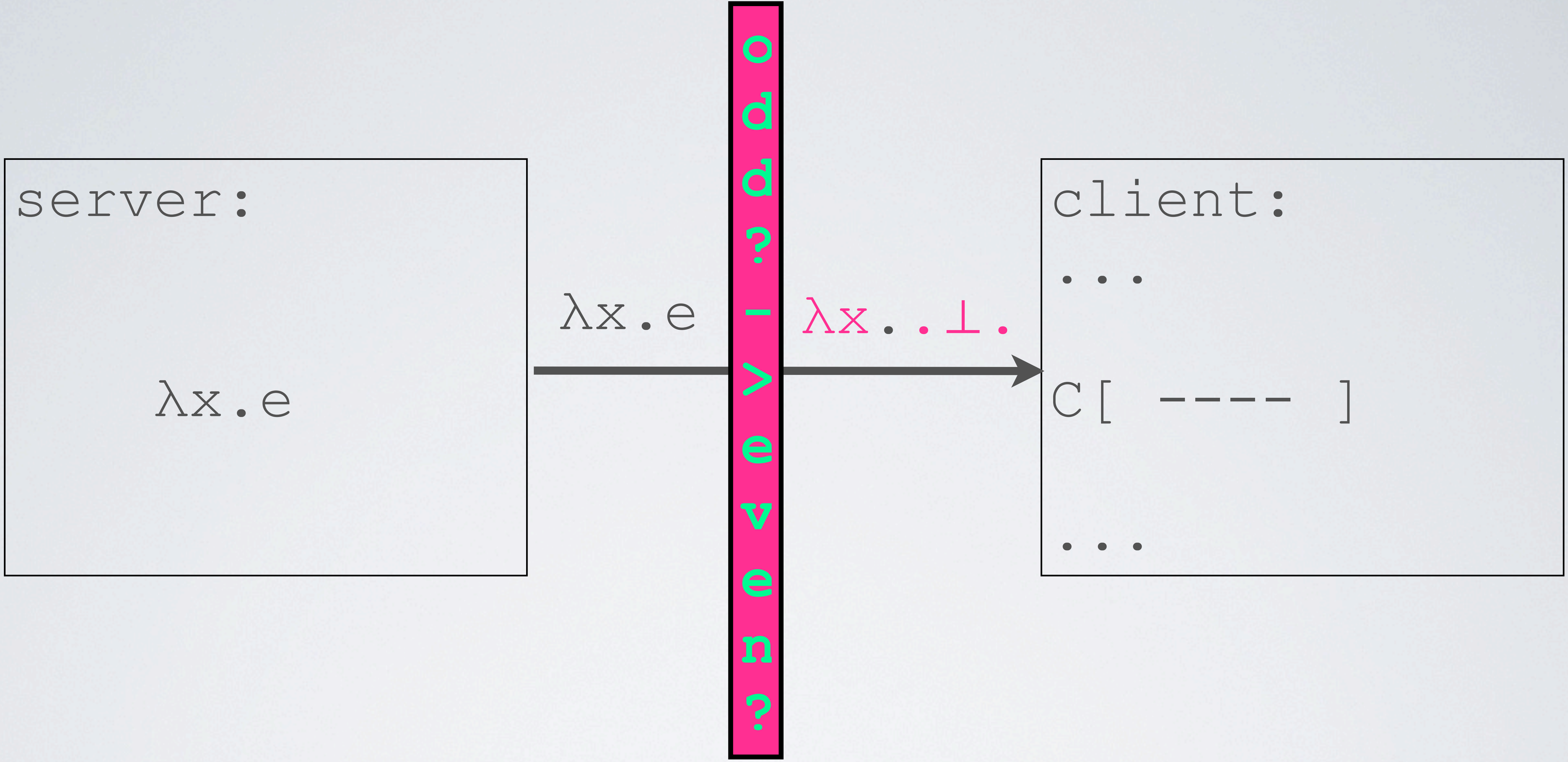
`C[ ---- ]`

`...`

For flat domains, produce an error if check doesn't hold.

Milner: TCS 1976; Plotkin: TCS 1978, Scott, ICALP 1981

`C[intf(odd? -> even?, λx:int.e)]`



```
server:

    λx.e
```

λx.e

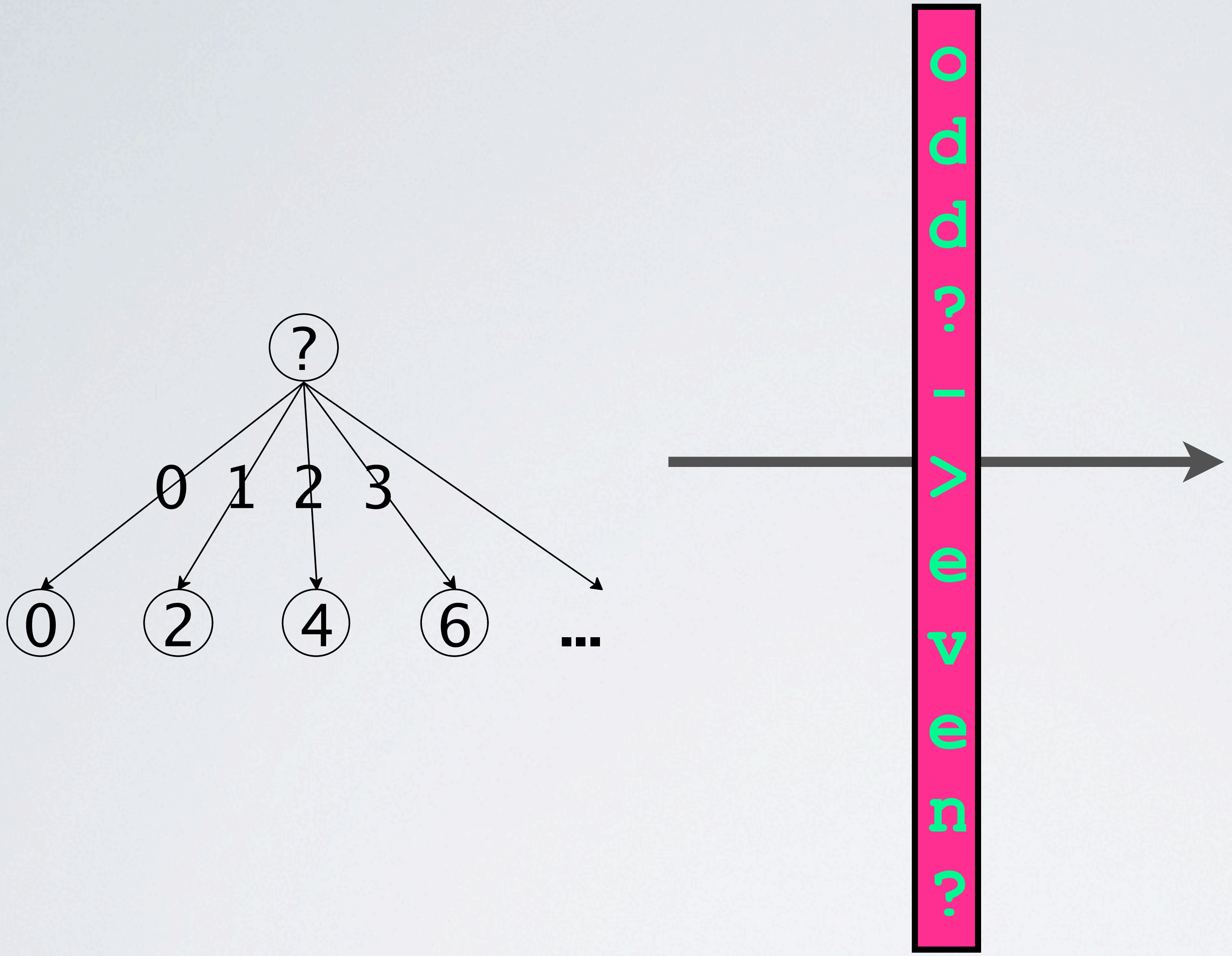
odd?->even?

λx..⊥.

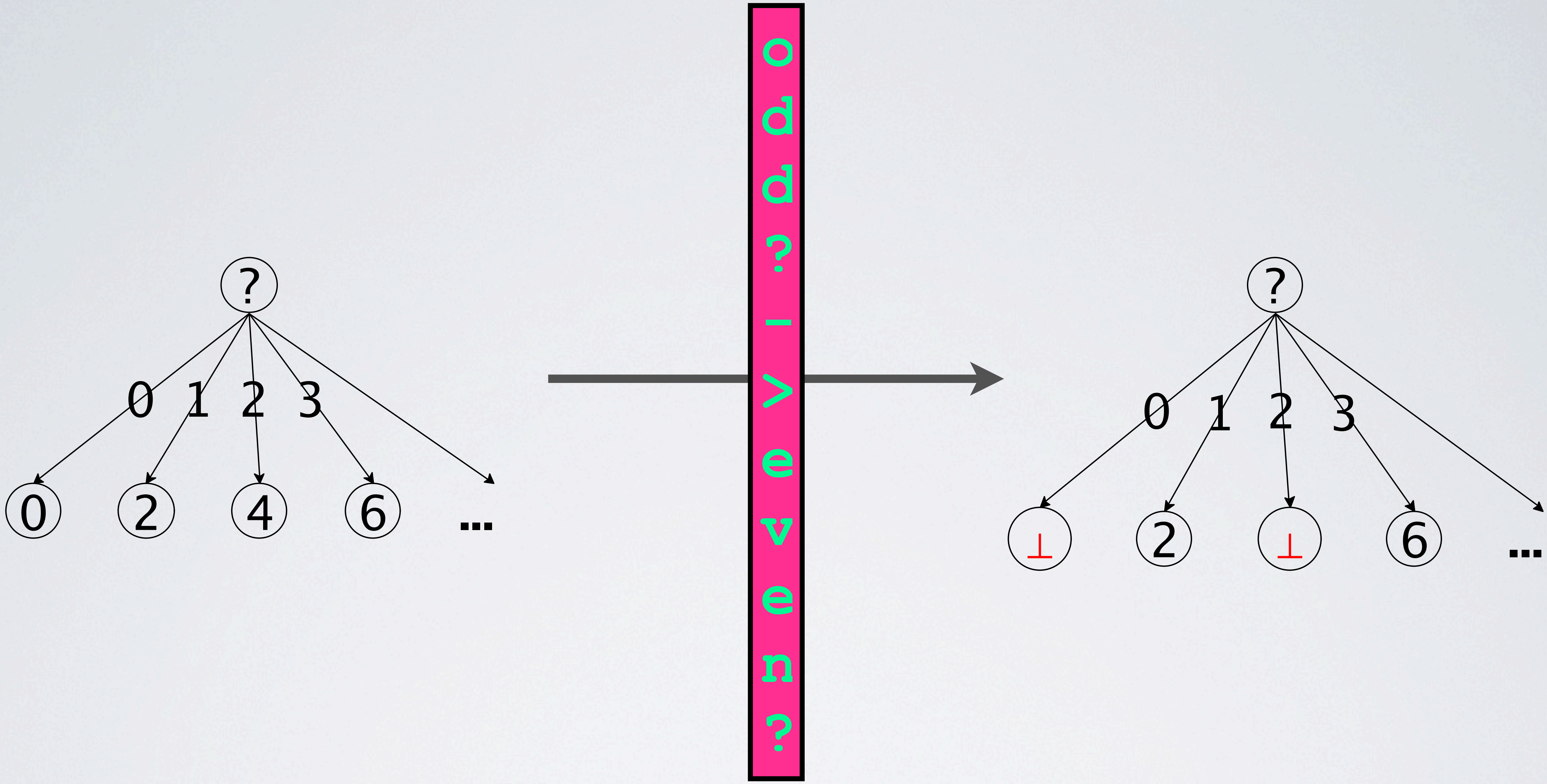
```
client:
...

C[ ---- ]

...
```

For function domains, ???

`C[intf(odd? -> even?, λx:int.2*x)]`



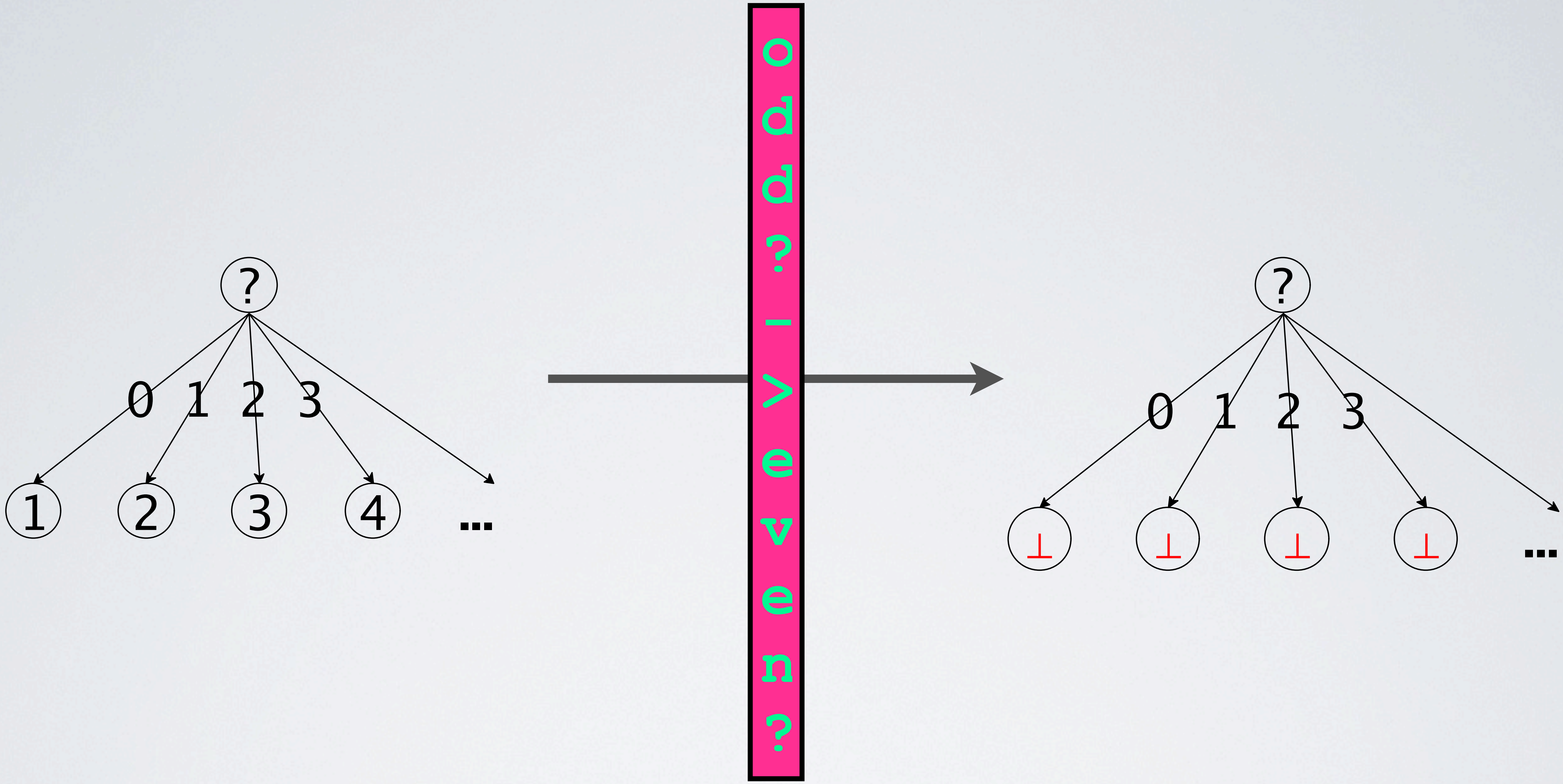
C[intf(odd? -> even?, λx:int.2*x)]



For function domains, prune “bad” behavior.

Milner: TCS 1976; Plotkin: TCS 1978, Scott, ICALP 1981

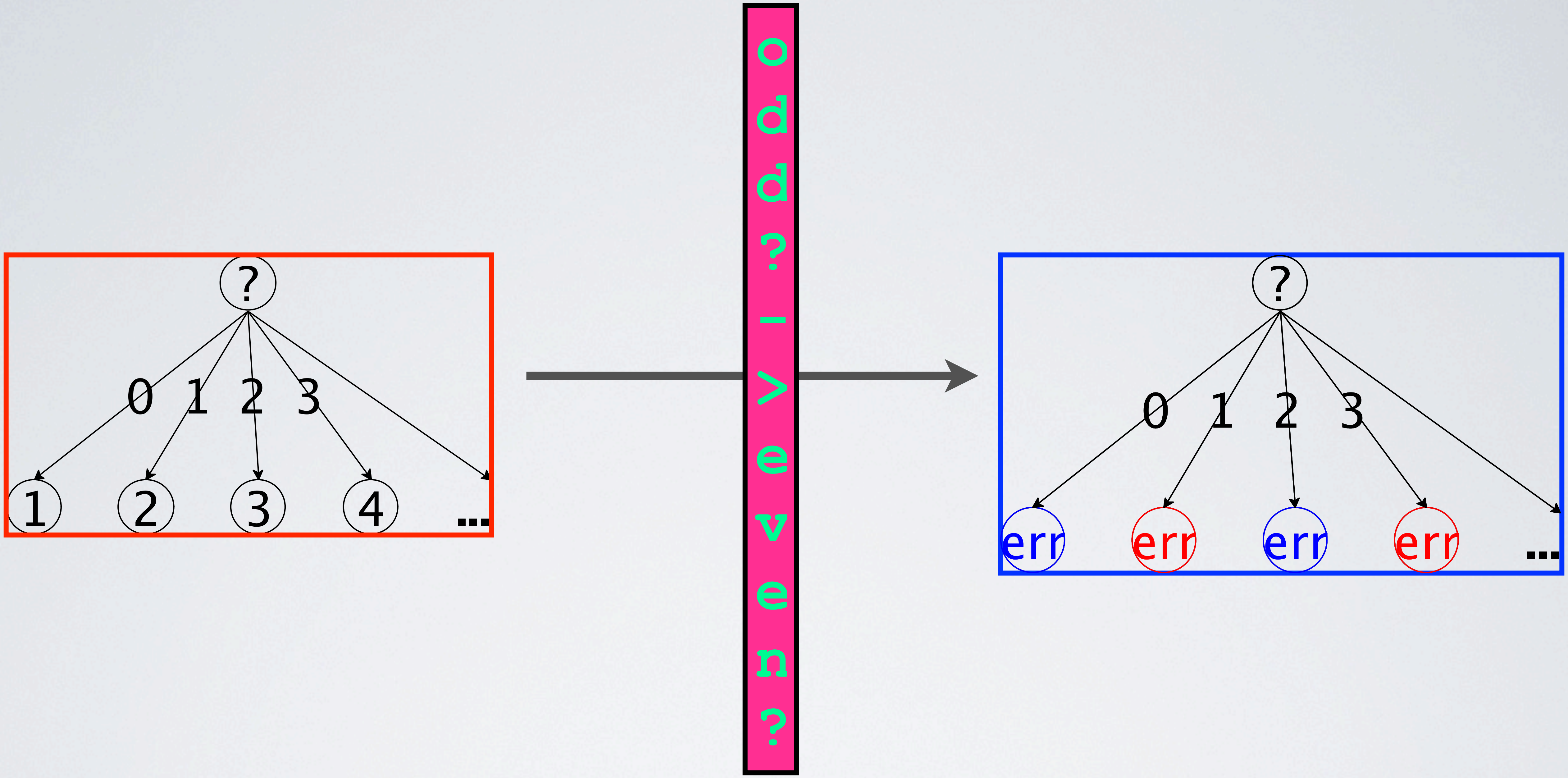
`C[intf(odd? -> even?, λx:int.x+1)]`



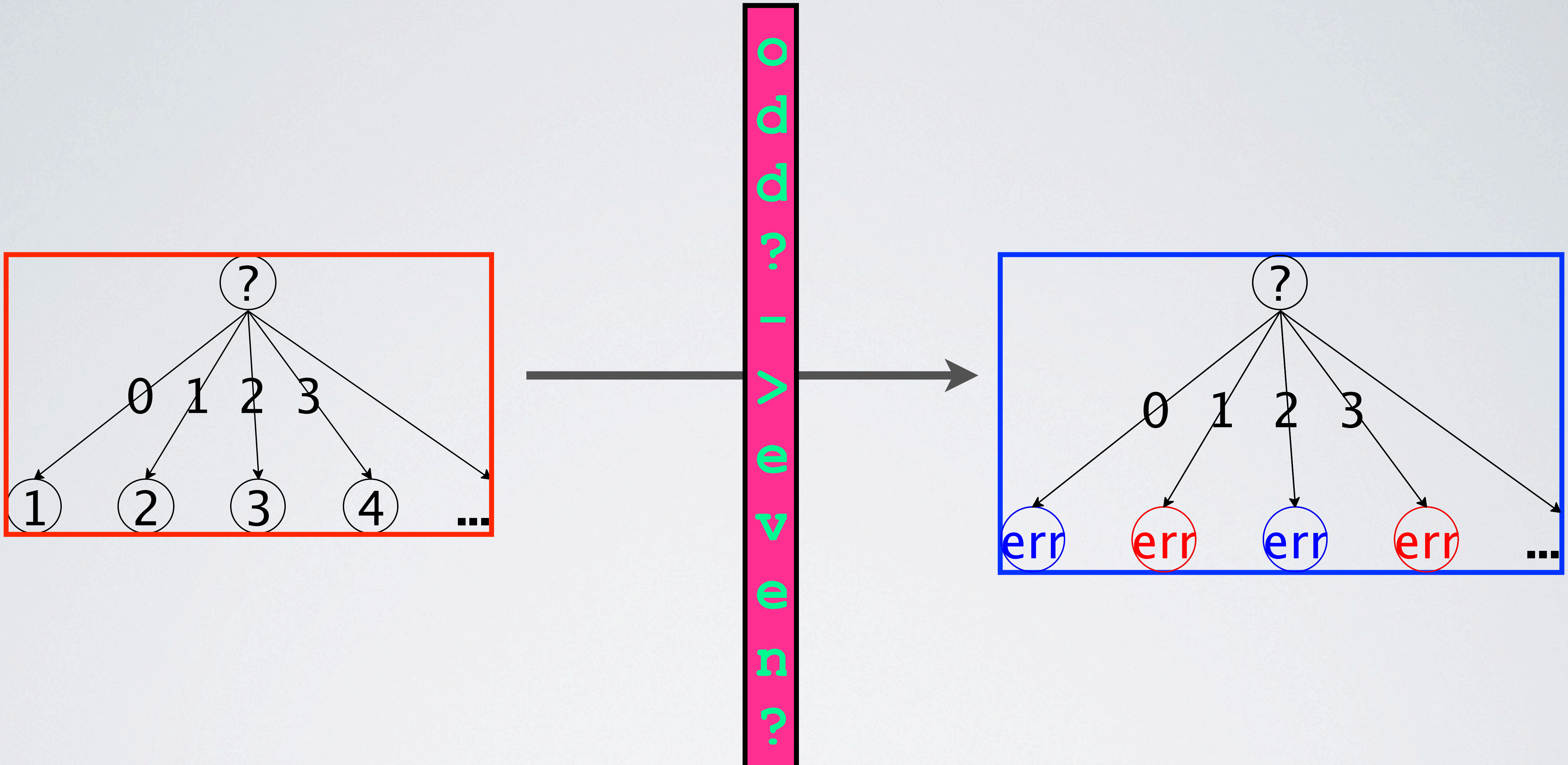
OUCH!

Milner: TCS 1976; Plotkin: TCS 1978, Scott, ICALP 1981

`C[intf(odd? -> even?, λx:int.x+1)]`



`C[intf(odd? -> even?, λx:int.x+1)]`



Thanks for observably sequential functions and *two* errors

Cartwright, **Curien,** Felleisen: IC 1995

Contracts Compose *Pairs of Error Projections* in **ObsSeqfun**

-- the idea works at all contract levels

-- contracts come with a natural order:

$$c1 \sqsubseteq c2 \textit{ iff } c1 = c1 \circ c2$$

$$(c1\text{-dom},c1\text{-rng}) \sqsubseteq (c2\text{-dom},c2\text{-rng})$$
$$\textit{iff } c2\text{-dom} \sqsubseteq c1\text{-dom},$$
$$\textit{and } c2\text{-rng} \sqsubseteq c1\text{-rng}$$

-- contracts are Dana Scott's data types:

$$\frac{\texttt{c1} \sqsubseteq \texttt{c2}}{\texttt{c2 -> c} \sqsubseteq \texttt{c1 -> c}} \qquad \frac{\texttt{c1} \sqsubseteq \texttt{c2}}{\texttt{c -> c1} \sqsubseteq \texttt{c -> c2}}$$

Findler, Blume, Felleisen: UC **Tech. Rp**t. 2004

Life is good and practical

-- greatly simplified implementation

Findler & Blume: FLOPS 2004

-- produced significant performance benefits

Findler 2004--present

-- guided many contract implementation efforts

- functor-like modules — Strickland & Felleisen, IFL 2008
- first-class classes — Strickland et al., TOPLAS 2013
- mutable objects — Dimoulas et al., ESOP 2013
- continuations — Takikawa et al., ESOP 2013

Still the model is imperfect

-- works for flat and higher-order contracts

Findler, Blume, Felleisen: UC Tech. Rpt. 2004

Still the model is imperfect

-- works for flat and higher-order contracts

-- but it all completely fails for dependent contracts

```
((unit? -> unit?)
->
 λf:real->real.
   (flat(\x. ...(f 0)...) -> unit?))
```

-- and this kind of code is realistic!

Findler, Blume, Felleisen: UC Tech. Rpt. 2004

Probing HO Values is Real

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x))))))])

(define (tangent? f x) ...)
```

Probing HO Values is Real

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x)))))))])

(define (tangent? f x) ...)
```

expensive

Probing HO Values is Real

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime
         (->d ((x real?))
              ;; result of fprime:
              (y (and/c real? (tangent? f x)))))])

(define (tangent? f x) ...)
```

expensive

```
(provide/contract
 [d/dx
  (->d ((f (-> real? real?)))
       ;; result of d/dx:
       (fprime (-> real? real?))
       #post (test-tangent-randomly fprime f))])

(define (tangent? f x) ...)
```

Semantics fails to explain pragmatics, especially the proper assignment of blame.

Semantics fails to explain pragmatics, especially the proper assignment of blame.

It hid a lingering bug for over a decade.

PRAGMATICS

> From a strictly operational point of view, semantics are unnecessary. To specify a language, we need only give its syntax and pragmatics.
>
> -- James Morris, MIT 1968

PRAGMATICS

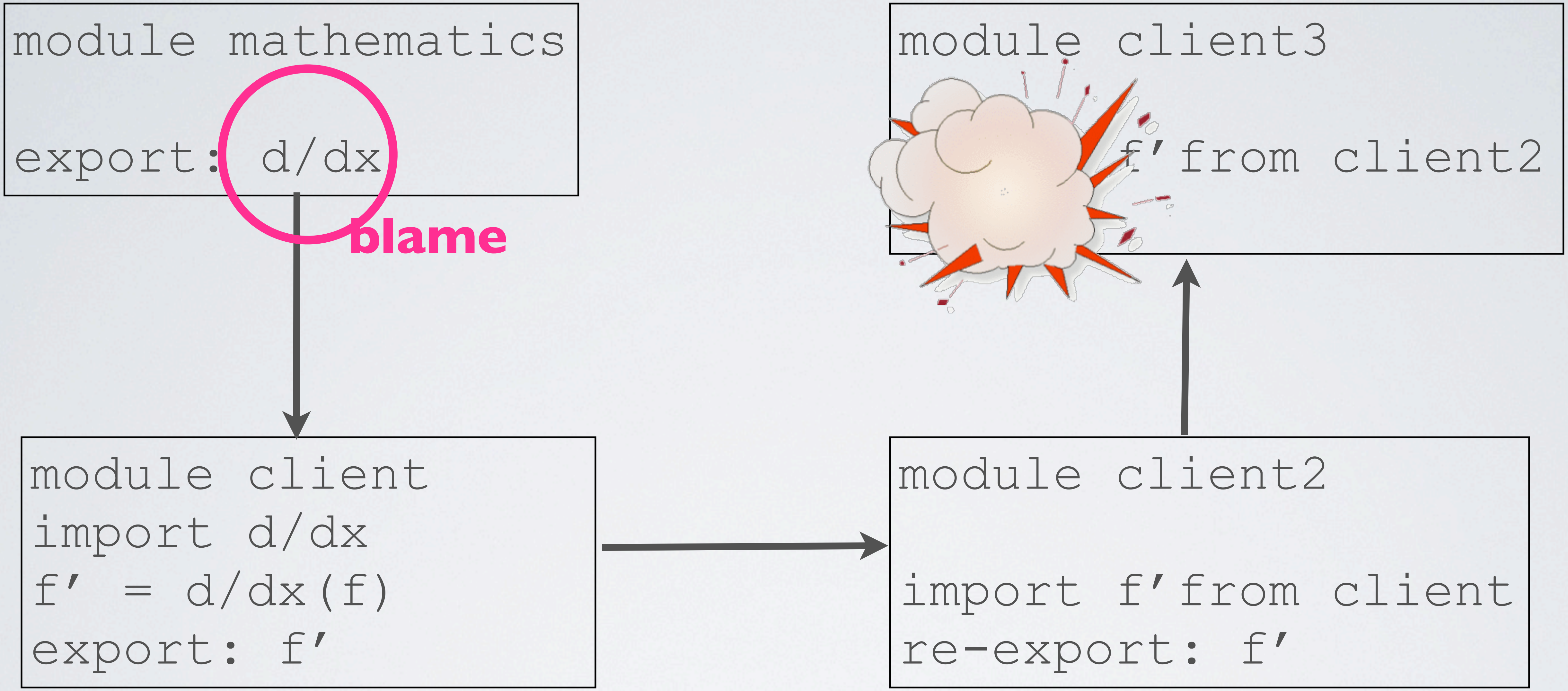
~

OPERATIONAL WORKINGS

~

OPERATIONAL SEMANTICS

Pragmatics Means “who’s fault is it”



```
module mathematics

export: d/dx
```

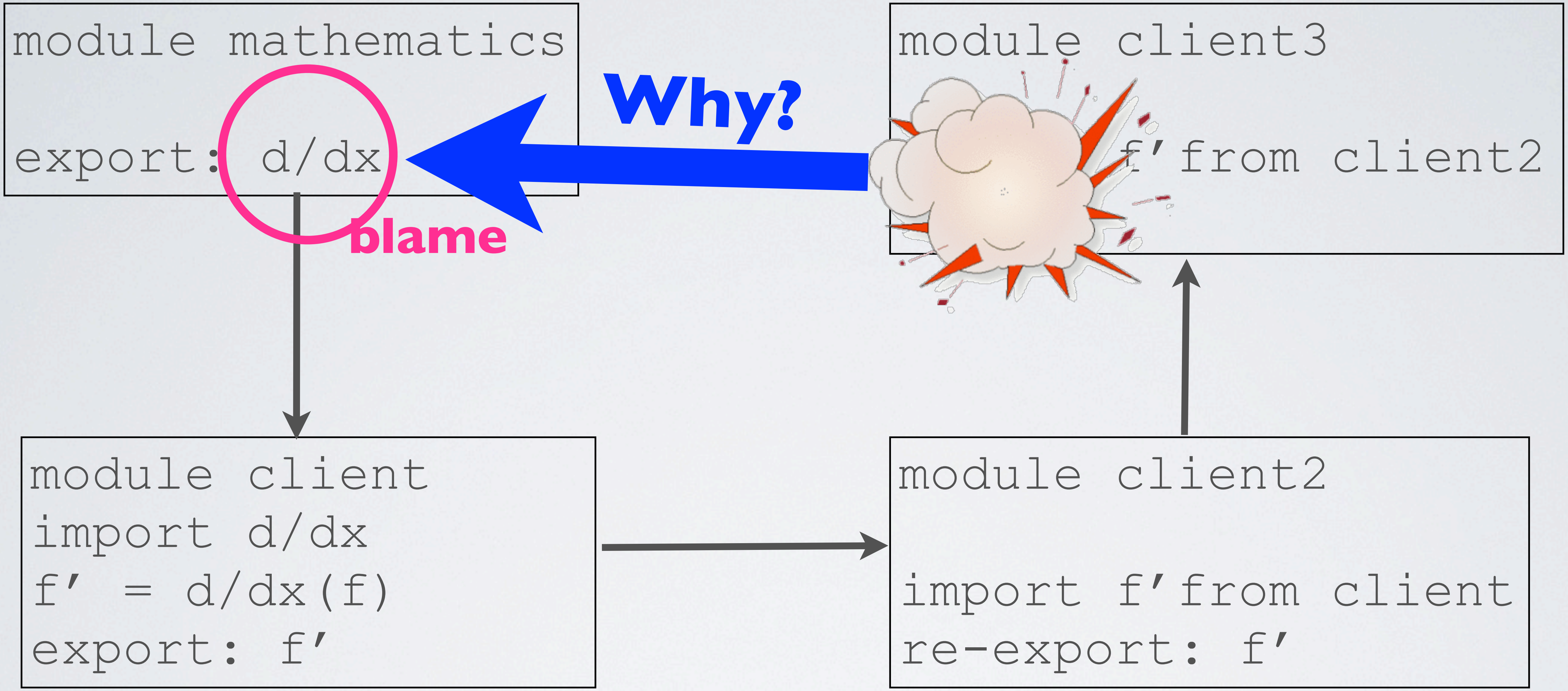
blame

```
module client
import d/dx
f’ = d/dx(f)
export: f’
```

```
module client2

import f’from client
re-export: f’
```

```
module client3

f’ from client2
```

Pragmatics Means “who’s fault is it”



```
module mathematics

export: d/dx
```

Why?

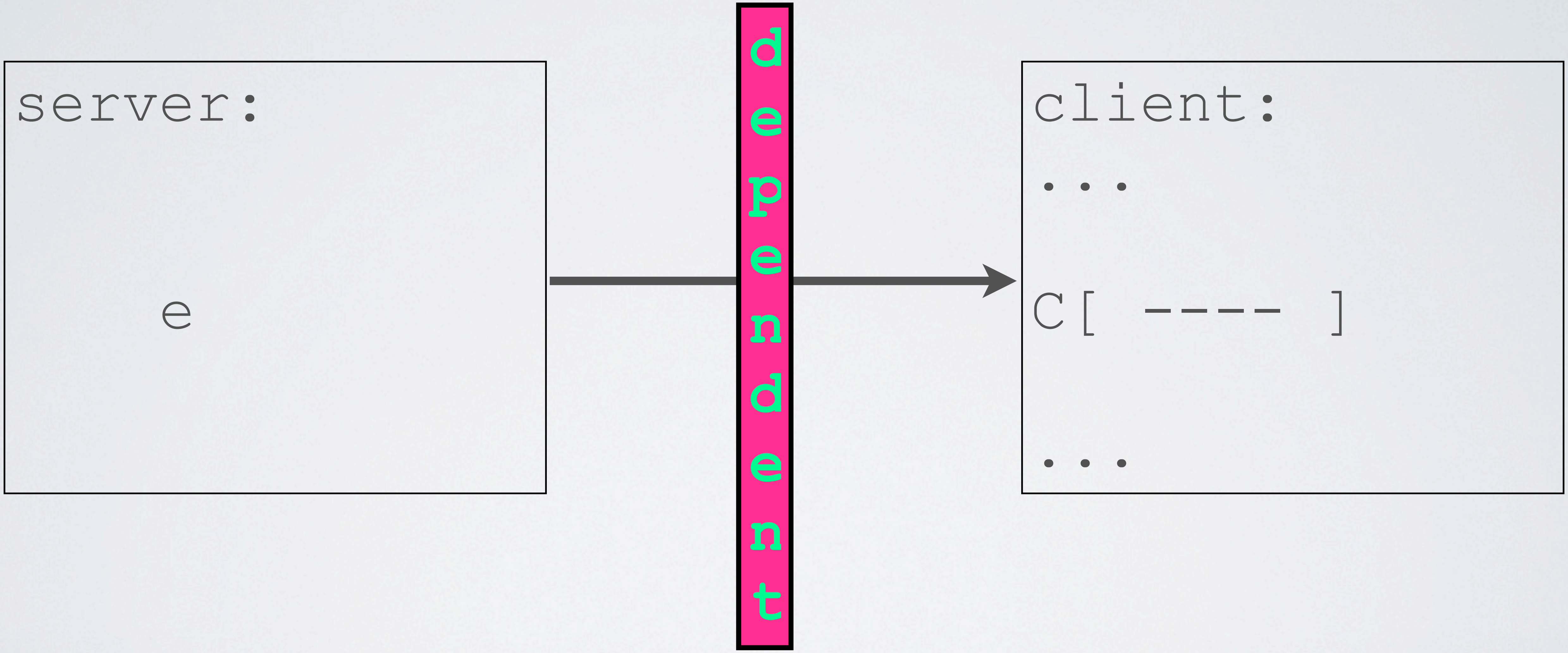
blame

```
module client3

f’ from client2
```

```
module client
import d/dx
f’ = d/dx(f)
export: f’
```

```
module client2

import f’from client
re-export: f’
```

```
C[
  intf((odd? -> even?) -> λf.flat(f(0) > 0),
        λf:int->int.f(1)) ]
```



server:

e

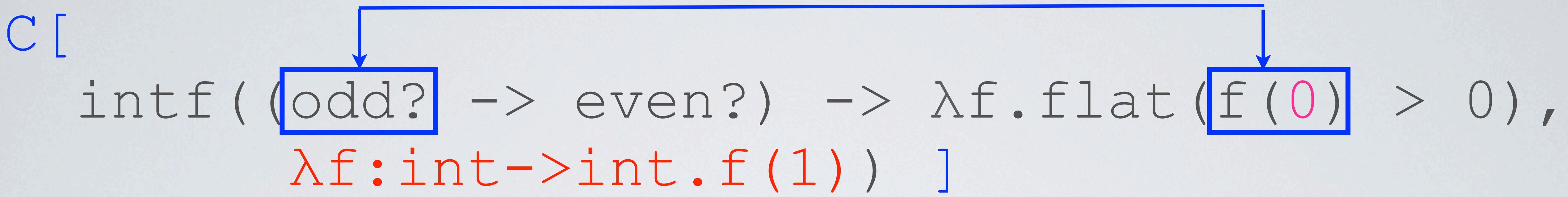
dependent

client:

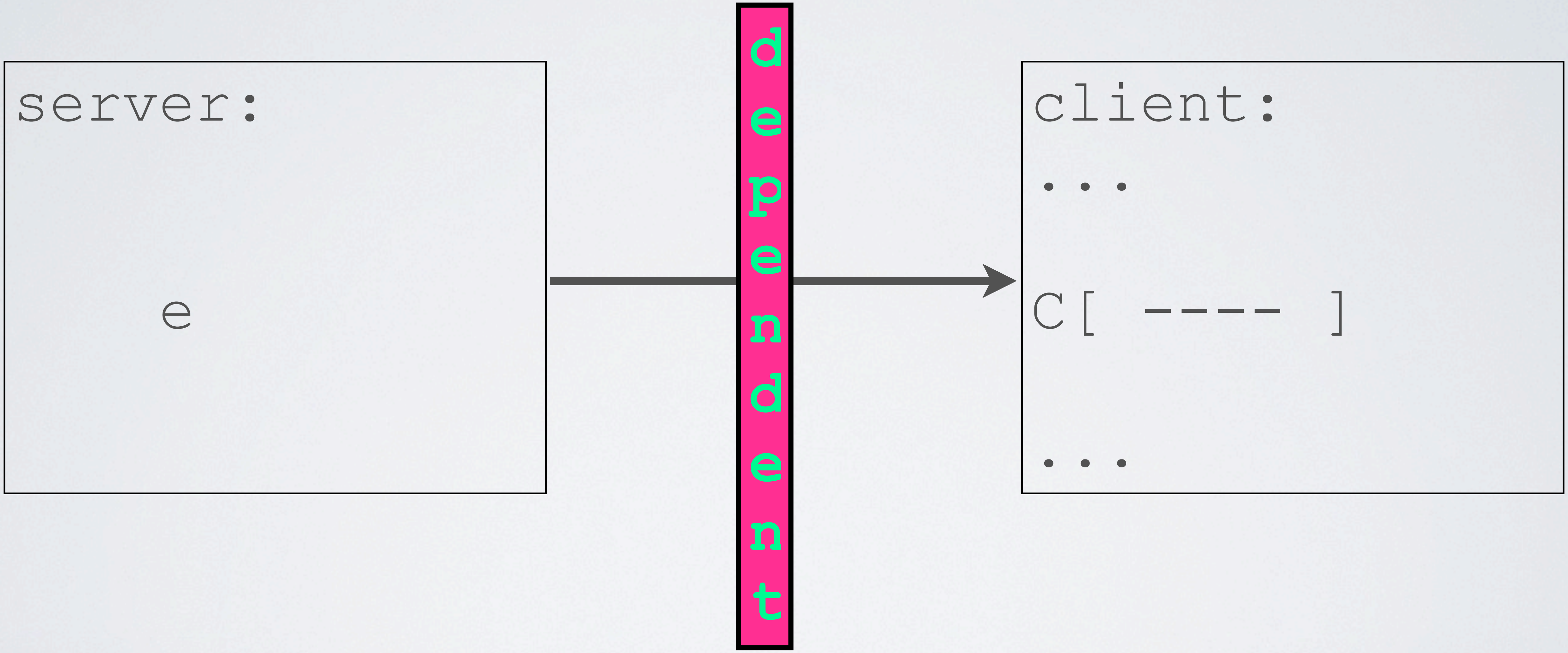
...

C[----]

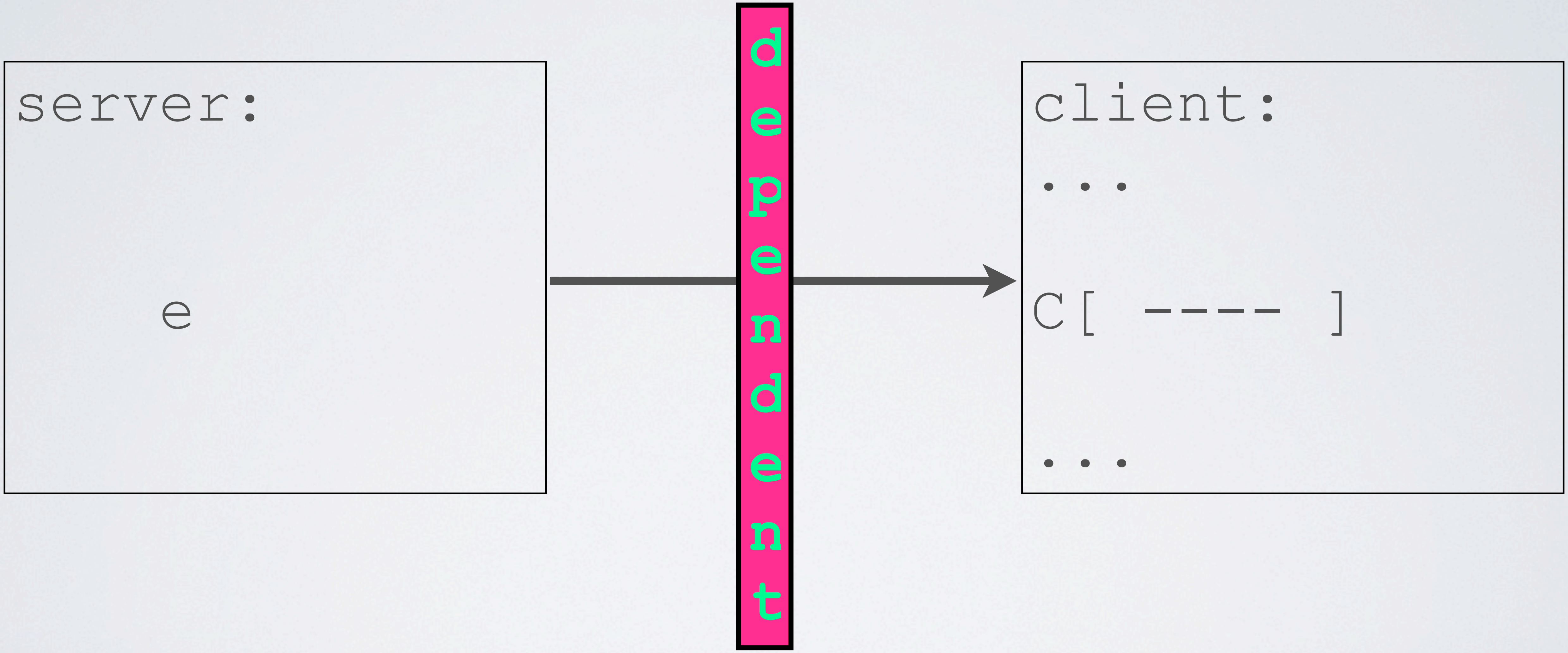
...



```
C[
  intf((odd? -> even?) -> λf.flat(f(0) > 0),
       λf:int->int.f(1)) ]
```



```
C[
  intf((odd? -> even?) -> λf.flat(f(0) > 0),
       λf:int->int.f(1)) ]
```



server:

e

dependent

client:

...

C[----]

...

Who to blame?

Reductions: No Blame Assignment

```
intf(flat(λx:b.e), v)=>
                          if e(v){v} err

intf(c->(λx.c’), f)   =>
                 λx.intf(c’,f(intf(c,x))
```

Findler & Felleisen: ICFP 2002

LAX BLAME: Contracts Correct by Definition

```
intf[P,C](flat(λx:b.e), v) =>
                             if e(v){v} err[P]

intf [P,C](c->(λx.c’), f)  =>
      λx.intf[P,C](c’,f(intf[C,P](c,x))
```

Findler & Felleisen: ICFP 2002

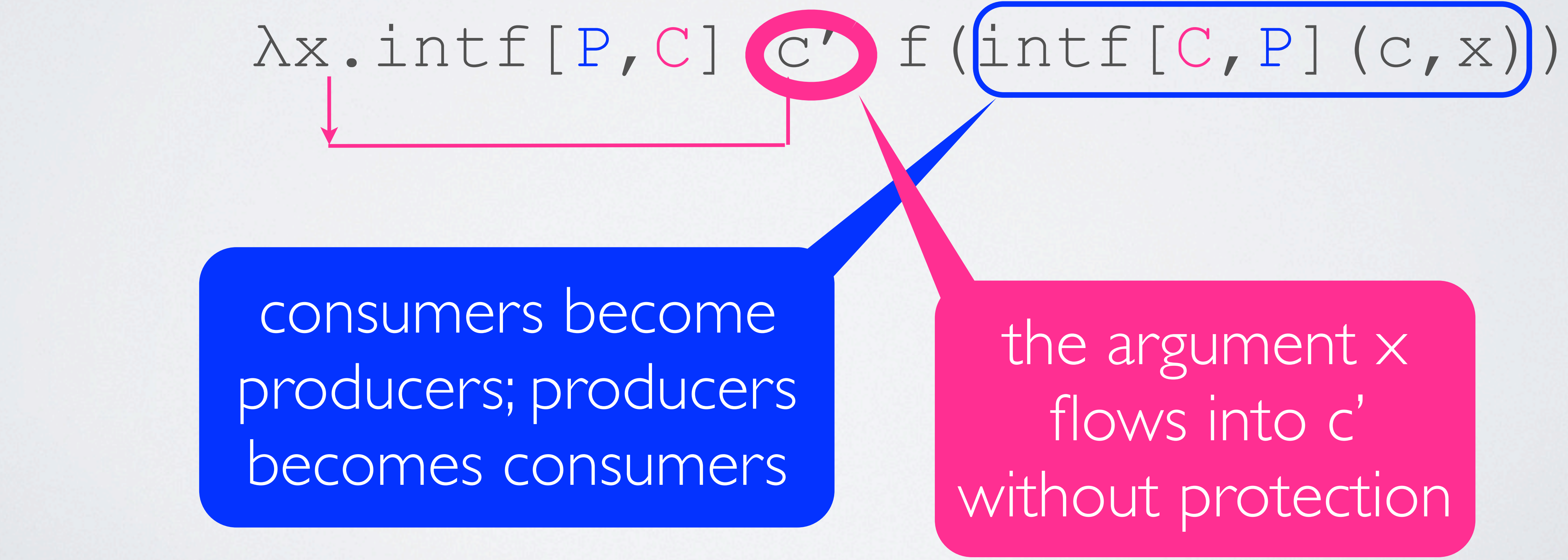
LAX BLAME: Contracts Correct by Definition

```
intf[P,C](flat(λx:b.e), v) =>
                           if e(v){v} err[P]

intf [P,C](c->(λx.c’), f)   =>
      λx.intf[P,C](c’,f(intf[C,P](c,x)))
```

consumers become producers; producers becomes consumers

Findler & Felleisen: ICFP 2002

LAX BLAME: Contracts Correct by Definition



```
intf[P,C](flat(λx:b.e), v) =>
                               if e(v){v} err[P]

intf [P,C](c->(λx.c’), f)  =>
      λx.intf[P,C] c’ f(intf[C,P](c,x))
```

consumers become producers; producers becomes consumers

the argument x flows into c’ without protection

Findler & Felleisen: ICFP 2002

LAX BLAME: Contracts Correct by Definition

```
intf[P,C](flat(λx:b.e), v) =>
                             if e(v){v} err[P]


intf [P,C](c->(λx.c’), f)   =>
      λx.intf[P,C] c’ f(intf[C,P](c,x))
```

consumers become producers; producers becomes consumers

the argument x flows into c’ without protection

Findler & Felleisen: ICFP 2002

PICKY BLAME: Contracts May Violate Contracts

```
intf[P,C](flat(λx:b.e), v)  =>
                      if e(v){v} err[P]

intf [P,C](c->(λx.c′), f)   =>
    λx.intf[P,C](c′{x:=intf[P,C](c,x)},
                      f(intf[C,P](c,x))
```

Blume & MacAllister JFP 2006

Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.

Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.

But *picky* may blame the wrong component.

Theorem (Greenberg, Pierce, & Weirich, POPL 2010)

The *picky* pragmatics detects all the contract violations that the *lax* pragmatics discovers and then some.

But *picky* may blame the **wrong** component.

Definition (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

M may be blamed for a contract violation *iff* it controls the flow of a value through a contract and the value violates a part of the contract that belongs to M's obligations.

Theorem (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

The *lax* pragmatics never blames the wrong module; *picky* may blame the wrong module.

But *lax* may fail to discover a contract violation.

Definition (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

M may be blamed for a contract violation *iff* it controls the flow of a value through a contract and the value violates a part of the contract that belongs to M's obligations.

Theorem (Dimoulas, Findler, Flanagan, Felleisen, POPL 2011)

The *lax* pragmatics never blames the wrong module; *picky* may blame the wrong module.

But *lax* may **fail** to discover a contract violation.

Definition (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another *without flowing through a contract boundary.*

Definition (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another *without flowing through a contract boundary.*

Theorem (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

The *picky* pragmatics fails to discover contract violations, but *lax* may allow values to slip through w/o checking.

Definition (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

A contract system fails to discover a contract violation if a bad value can migrate from one module to another *without flowing through a contract boundary.*

Theorem (Dimoulas, Tobin-Hochstadt, Felleisen, ESOP 2012)

The *picky* pragmatics fails to discover contract violations, but *lax* may allow values to slip through w/o checking.

Neither *lax* nor *picky* are "good" pragmatics.

Neither *lax* nor *picky* are "good" pragmatics.

INDY BLAME: Contracts *May Blame Contracts*

```
intf[P,C,K](flat(λx:b.e), v) =>
                          if e(v){v} err[P]

intf[P,C,K](c->(λx.c'), f)  =>
  λx.intf[P,C,K](c'{x:=intf[P,K,K](c,x)},
                          f(intf[P,C,K](c,x))
```

Dimoulas et al.: POPL 2011

INDY BLAME: Contracts *May Blame Contracts*

```
intf[P,C,K](flat(λx:b.e), v) =>
                              if e(v){v} err[P]

intf[P,C,K](c->(λx.c’), f)  =>
  λx.intf[P,C,K](c’{x:=intf[P,K,K](c,x)},
                             f(intf[P,C,K](c,x))
```

contracts are *independent* components, and they become consumers

Dimoulas et al.: POPL 2011

Theorem (Dimoulas et al.: POPL 2011, ESOP 2012)

The *indy* pragmatics detects all contract violations and assigns blame to a module that controls the flow of the bad value.

Theorem (Dimoulas et al.: POPL 2011, ESOP 2012)

The *indy* pragmatics detects all contract violations and assigns blame to a module that controls the flow of the bad value.

And there is no "but" other than the proof is horribly complex.

Life is perfect

-- *correct blame* uncovered bugs in our implementation

-- *complete monitoring* became our design guide

-- *correct blame* with *complete monitoring*
together have become the foundation of our
Typed Racket (gradual typing) research program

CONCLUSIONS

Two Semantic Frameworks for HO Contracts

- SPCF's semantics of "concrete data structures"
- it is elegant theory
- it's *imperfect* but *easy-to-use*

- *perfect* reduction pragmatics
- with an ugly subject reduction proof
- but practical implications

THE END

So thanks again for *CDS*,
SPCF, and two errors.