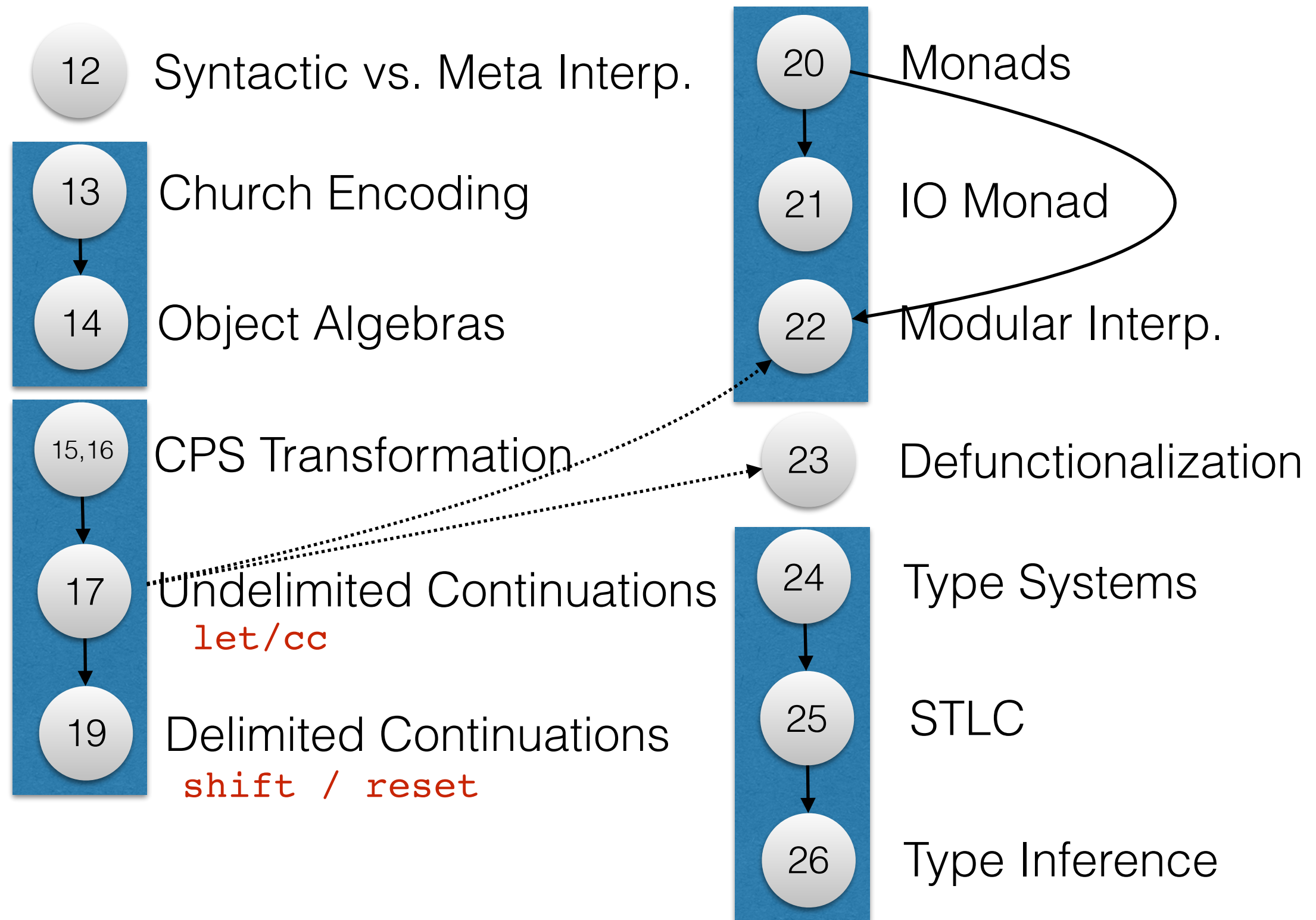


Lectures



12 Syntactic Vs. Meta Interpretation

- meta vs. syntactic interpretation of a feature:
uses vs. does not use
the corresponding host language feature
- e.g. first-class functions:
- meta interpreter returns host language function from
values to values as value (compositional)
- syntactic interpreter returns closures
- alternative meta: HOAS -> directly use (embed)
host language functions (from `expr.s` to `expr.s`)

13 Church Encoding

= "fold" over the respective represented object
(number, list, ...)

13

Church Encoding

= "fold" (catamorphism) over the respective represented object (number, list, ...)

know: fold over lists

```
def fold(l)(f,e) = match l with
  []      => e
  h::t    => f(h, fold(t)(f,e))
```

13 Church Encoding

```
def fold(l)(f,e) = match l with
  []      => e
  h::t    => f(h, fold(t)(f,e))

// Church encoding of [1,2,3]
def ce_123 : (A->B->B,B)->B =
  fold([1,2,3])
```

13 Church Encoding

```
def fold(l)(f,e) = match l with
  []      => e
  h::t    => f(h, fold(t)(f,e))
```

lists can be specialized to Peano nat.s:

```
// Nat = List of Unit
def fold(n)(f,e) = match n with
  []      => e
  ()::t   => f((), fold(t)(f,e))
```

13 Church Encoding

```
def fold(l)(f,e) = match l with  
  []      => e  
  h::t    => f(h, fold(t)(f,e))
```

lists can be specialized to Peano nat.s:

```
// Nat = List of Unit  
def fold(n)(f,e) = match n with  
  []      => e  
  ()::t   => f((), fold(t)(f,e))
```

always unit, no extra information at each position

13 Church Encoding

```
// Nat = List of Unit
def fold(n)(f,e) = match n with
  []      => e
  () :: t => f((), fold(t)(f,e))
```

compare:

```
def fold(n)(f,e) = match n with
  0      => e
  S(m)   => f(fold(m)(f,e))
```


13

Church Encoding

```
def fold(n)(f,e) = match n with  
  0      => e  
  S(m)   => f(fold(m)(f,e))
```

In other words:

fold over **n** = iterate **n** times the given function *f* starting with the given number *e*

$$f(f(\dots f(e)\dots))$$

13

Church Encoding

$$f(f(\dots f(e)\dots))$$

compare lists:

$$f((), f((), \dots f((), e)\dots))$$
$$f(1, f(2, \dots f(n, e)\dots))$$



13

Church Encoding

approach generalizes to other structures:
trees, booleans, ...



15,16 CPS Transformation

- every function gets an extra argument to pass the remaining computation = the **continuation**
- CPS transf. makes the evaluation explicit
- after CPS transf.: all function calls are tail calls

15,16 CPS Transformation

Compare:

```
def f(x) = 5 + g(h(x))
```

After CPS transf.:

```
def f(x) = h(x, a => g(a, b => 5 + b))
```

15,16 CPS Transformation

Compare:

```
def f(x) = 5 + g(h(x))
```

first thing to be
evaluated

After CPS transf.:

```
def f(x) = h(x, a => g(a, b => 5 + b))
```

first call

15,16 CPS Transformation

Compare:

`def f(x) = 5 + g(h(x))`

not a tail call

After CPS transf.:

not a tail call

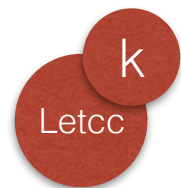
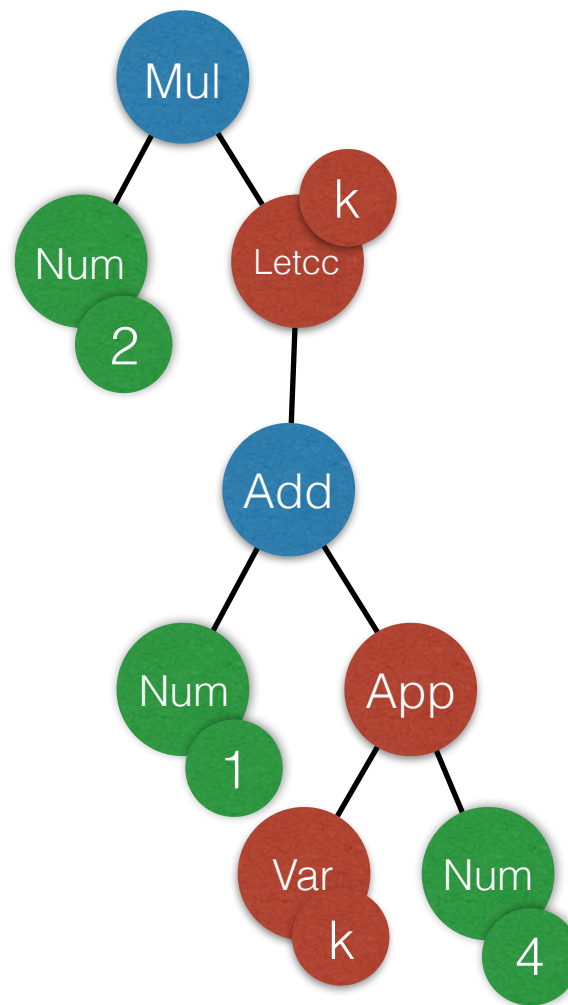
`def f(x) = h(x, a => g(a, b => 5 + b))`

tail call

tail call

17

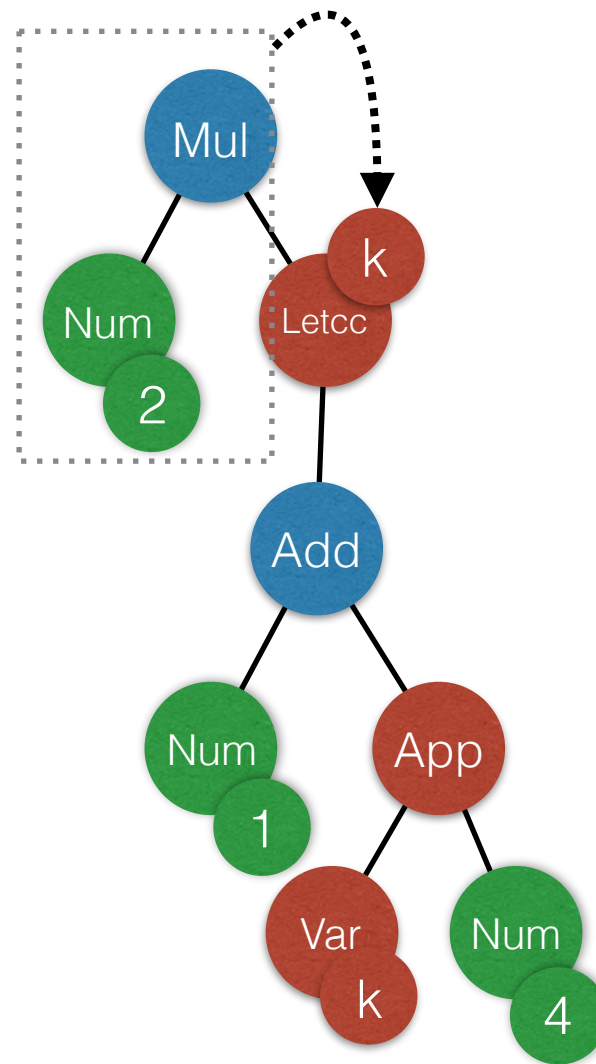
let/cc



new node type for binding first-class continuations

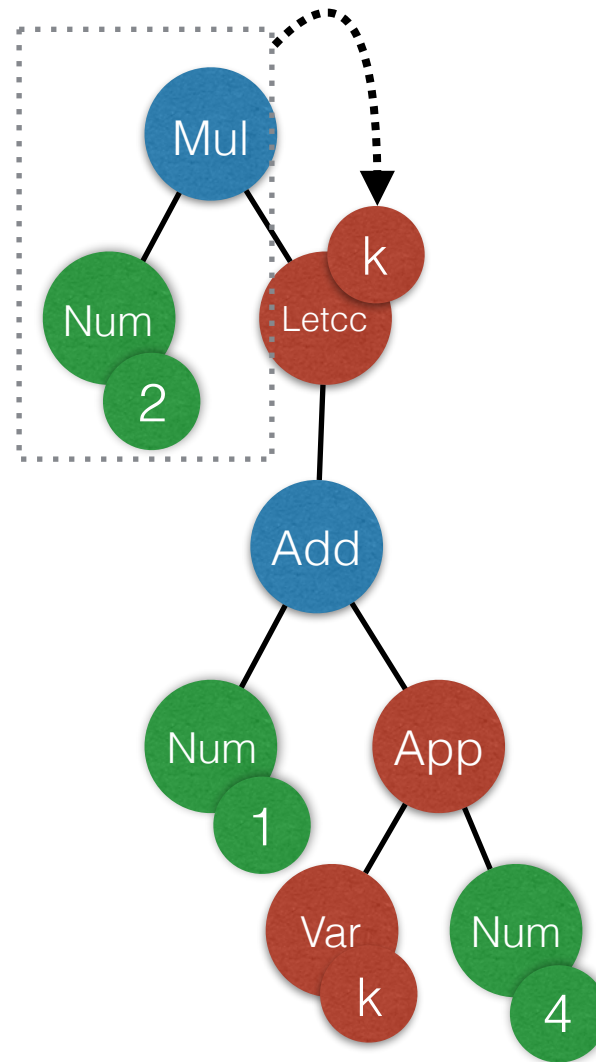
17

let/cc



17

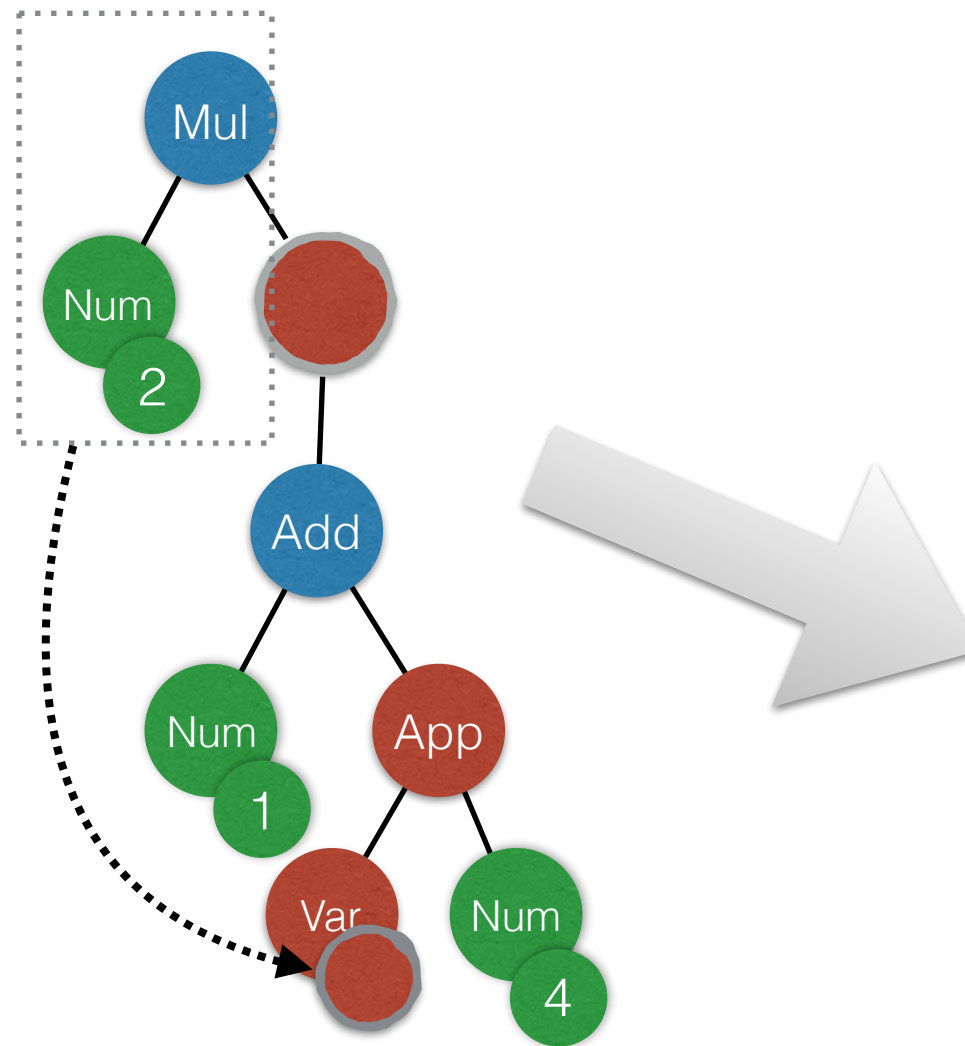
let/cc



= "the current continuation (**cc**)"

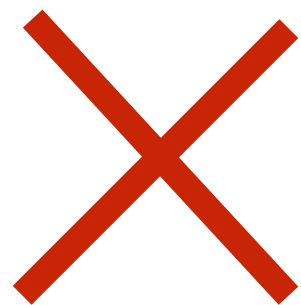
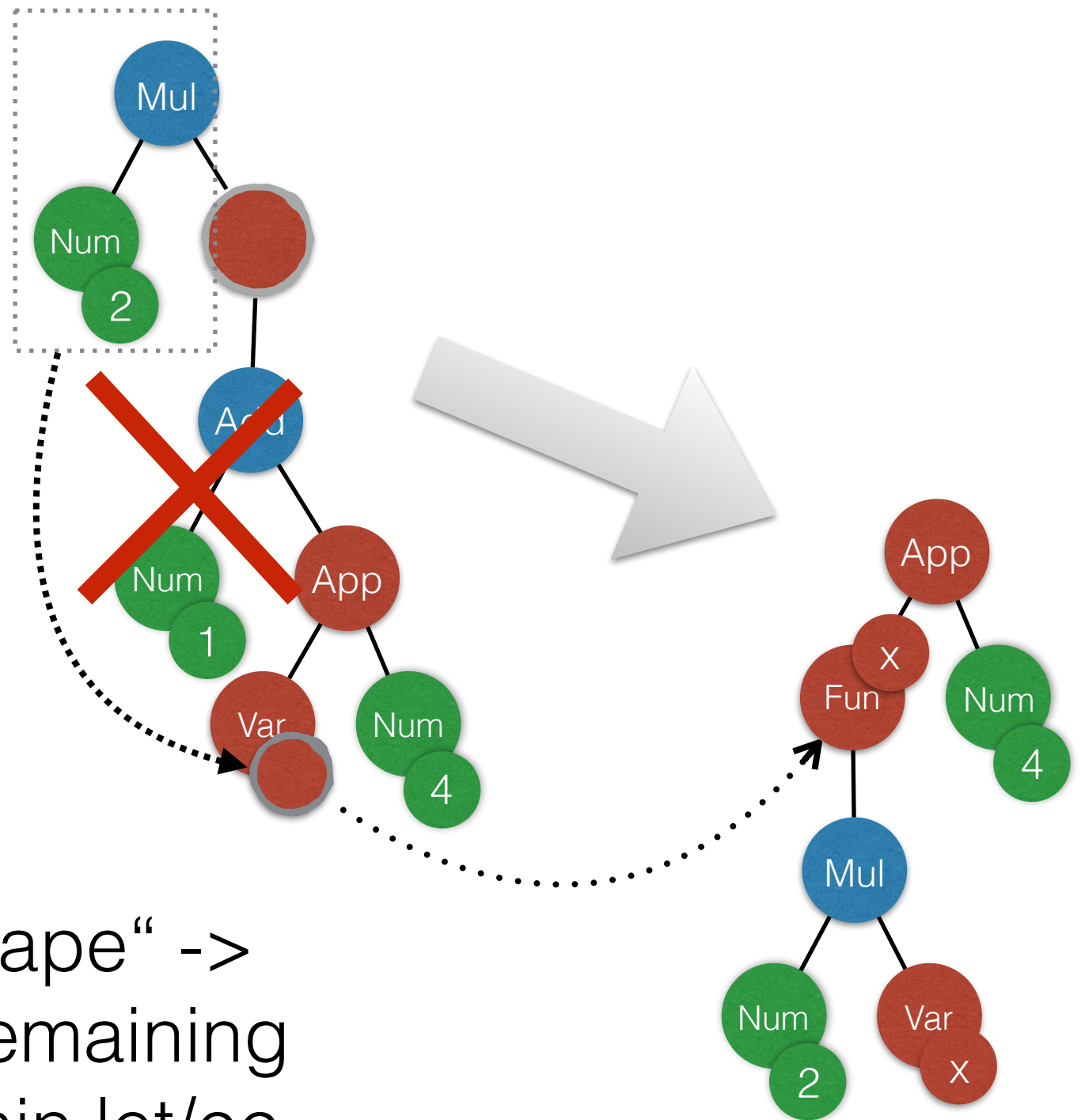
17

let/cc



17

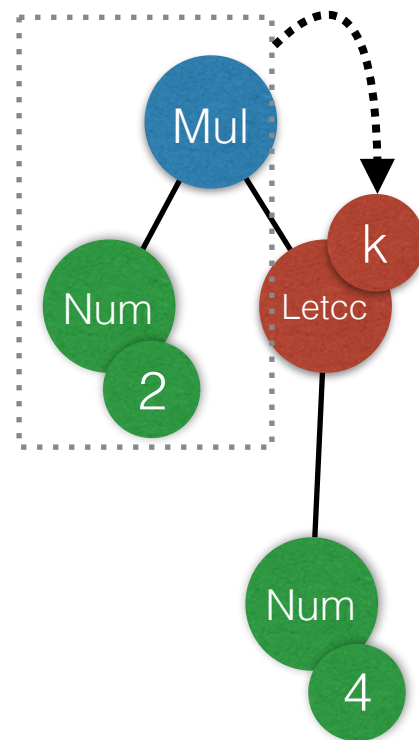
let/cc



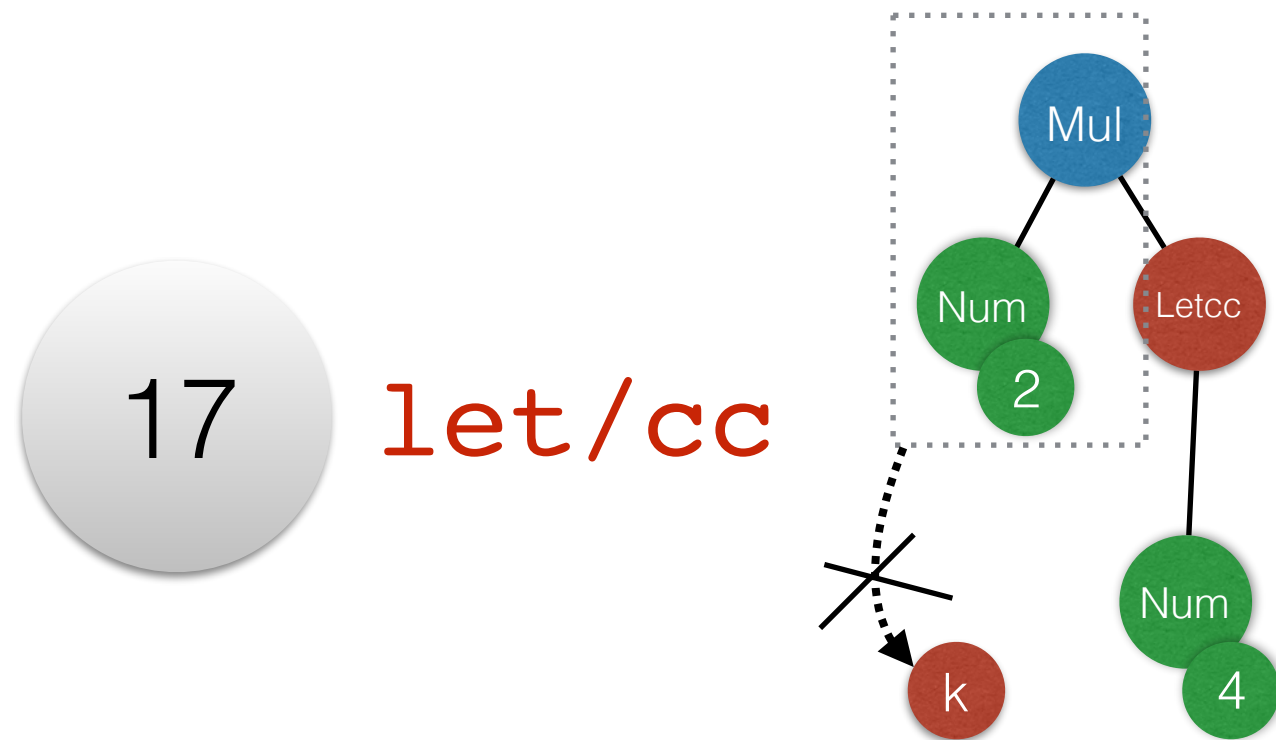
we want to "escape" ->
throw away the remaining
computation within let/cc

17

let/cc

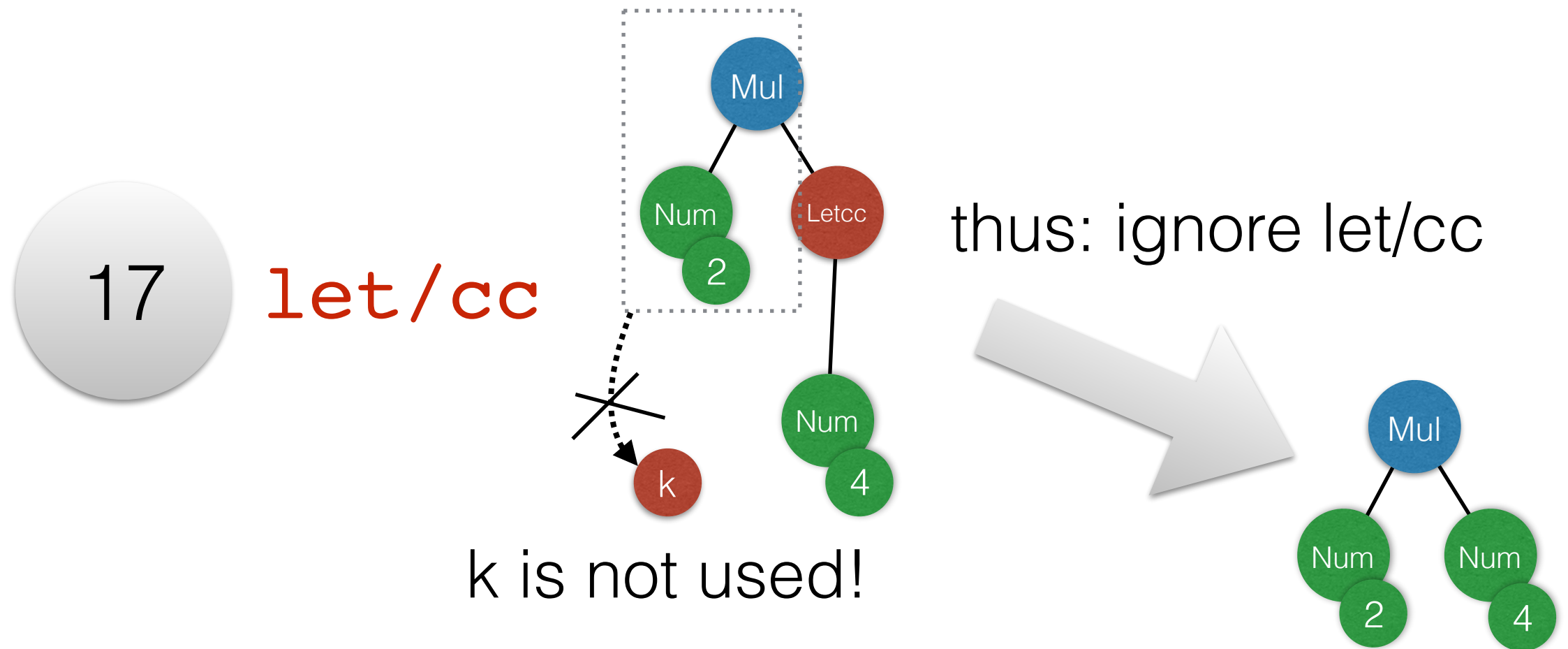


example with unused continuation

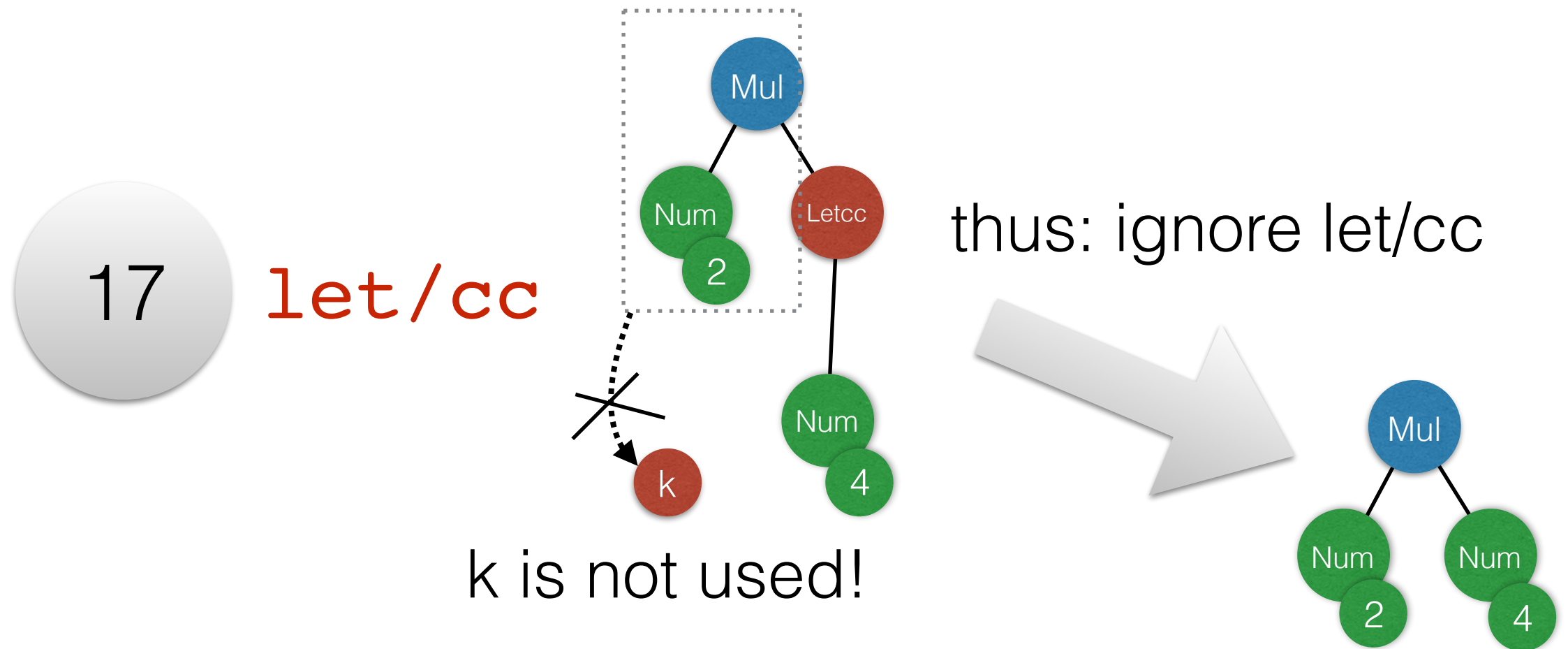


k is not used!

example with unused continuation

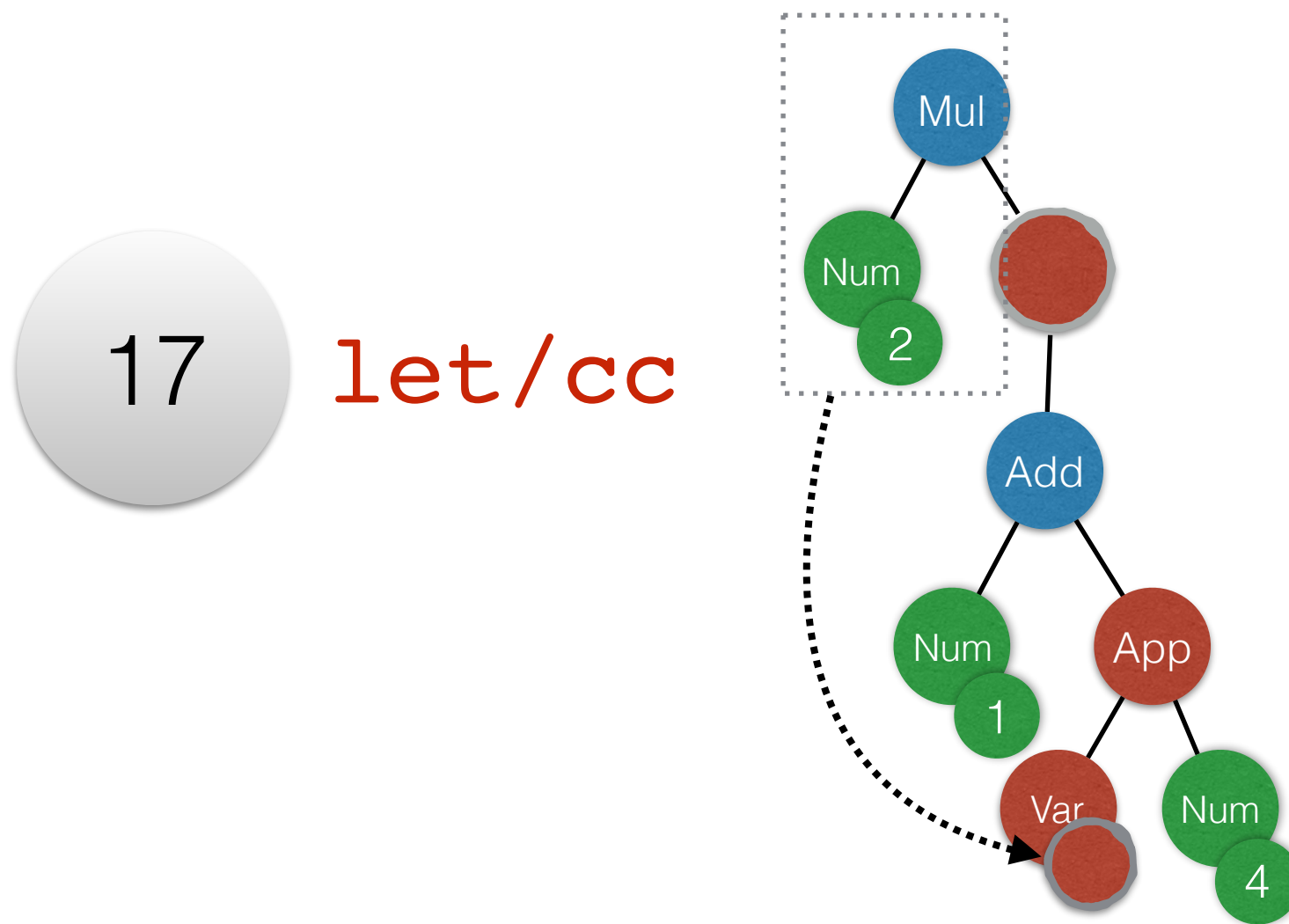


example with unused continuation



summary: with let/cc ...

- we can "escape" its body by calling the captured cont. (k)
- **but:** if no "escape" happens, computation ignores the let/cc

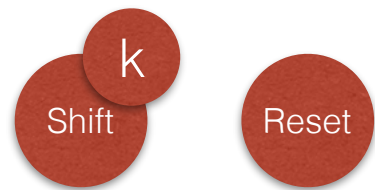
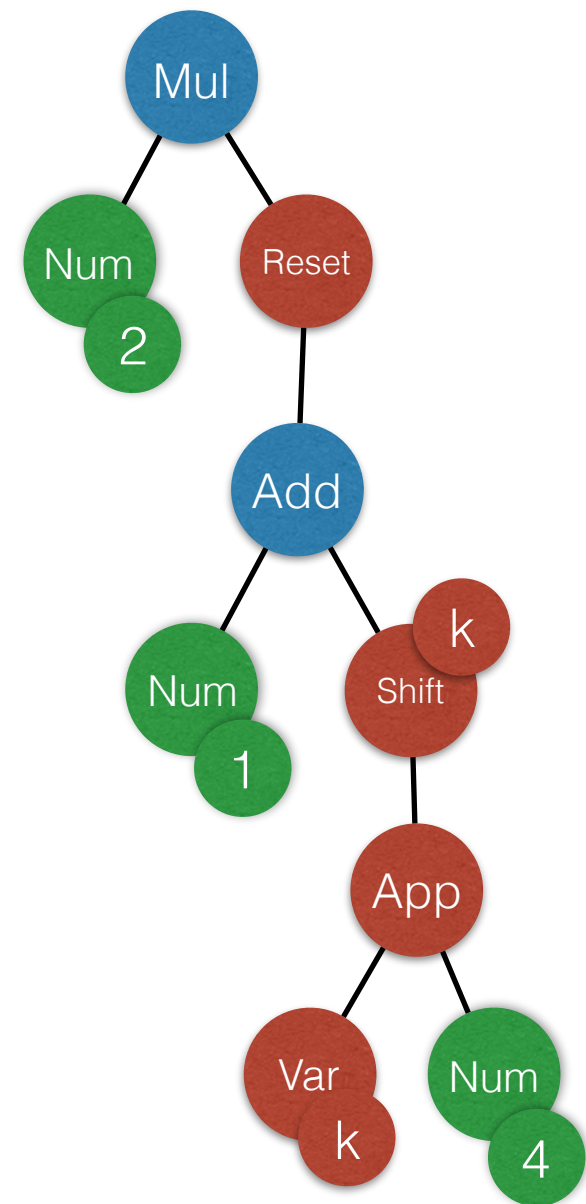


intuitively:

only even consider the "capturing" once k is actually applied

19

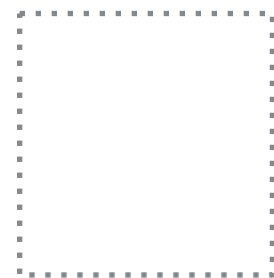
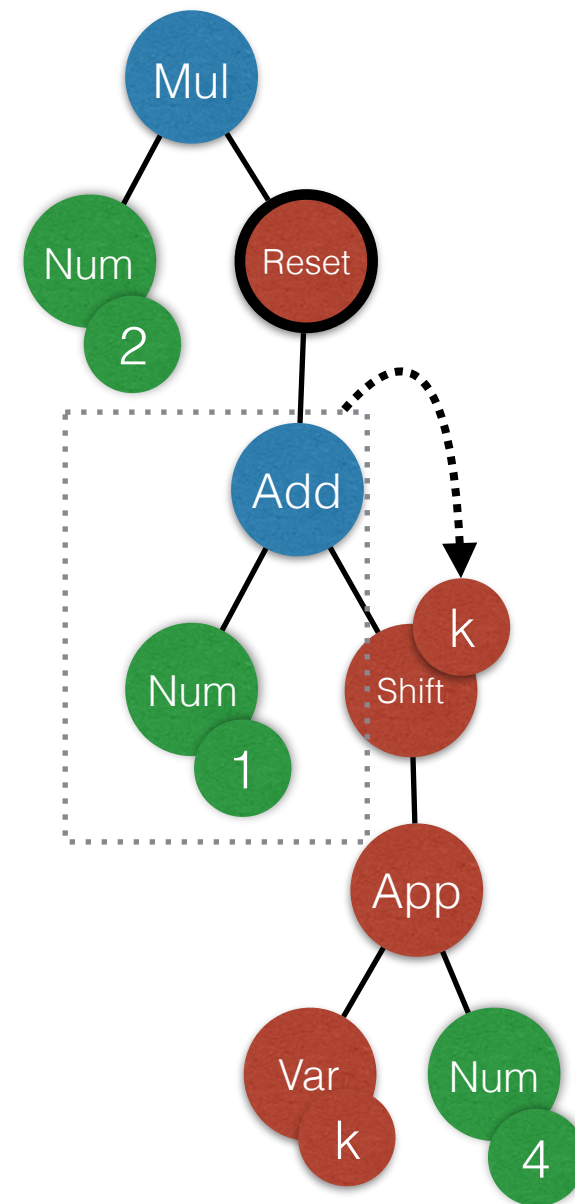
shift / reset



new node types for binding first-class
delimited continuations

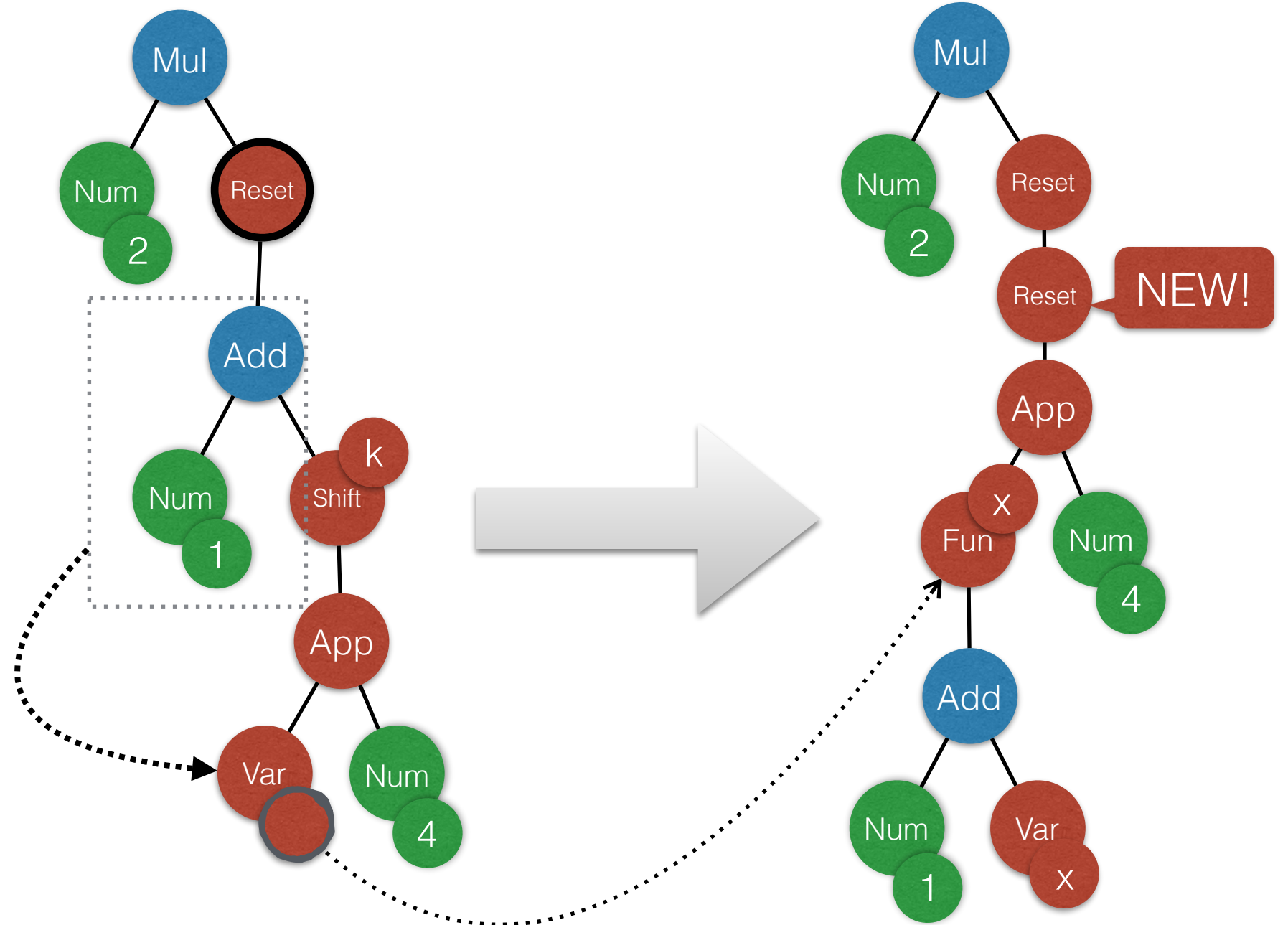
19

shift / reset



= "the delimited continuation up to the reset"

19





20


Monads

- a way to structure your code, especially in a (somewhat) modular way
- more specifically, allows to structure computational effects (e.g. the option monad for failure)
- example: **modular interpreter**

22

Modular Interpreter

- uses the technique of **monad transformers**
- with this, we **compose several monads** with which we **structure the effects** of the interpreters we saw so far:
 - *Reader* (for the environment)
 - *State* (for mutable state/reference cells)
 - *Continuation* (for first-class continuations)



23

Defunctionalization

- gets rid of all first-class functions
- to do so, all first-class functions become variants of a datatype (case classes)
- functionality is moved to an `apply` function for that datatype

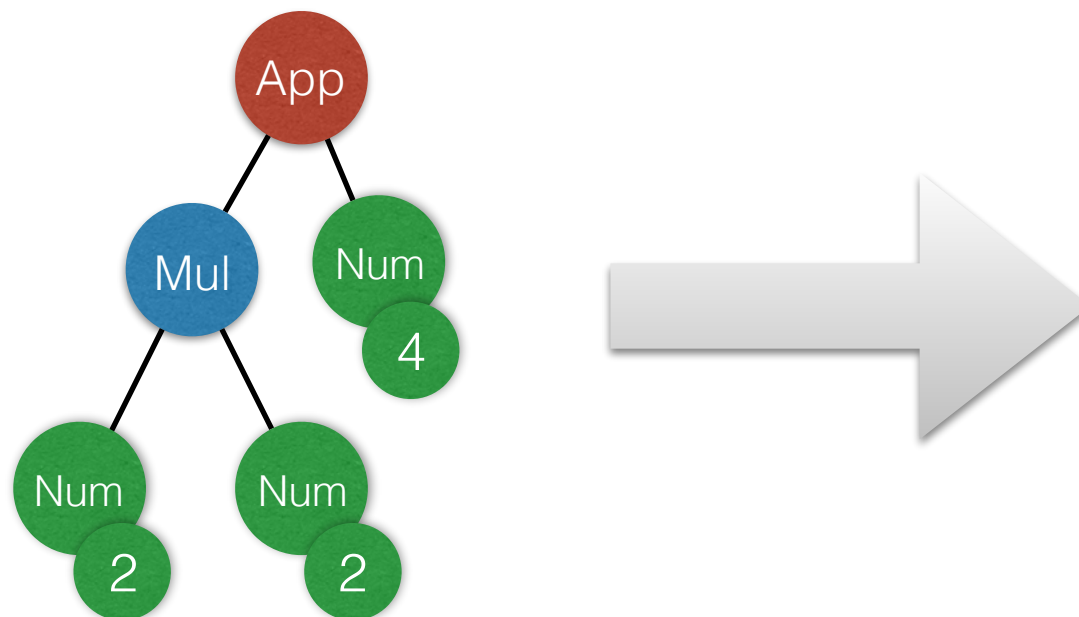
23

Defunctionalization

- requires **lambda lifting** as a pre-step
- especially useful after a CPS transformation:
- together, CPS transform and defunc. make the evaluation explicit and get rid of first-class functions
- the resulting program is closer to an abstract machine
-> useful to **compile** it to efficient low-level code
- important drawback:
defunc. is a **global** program transformation

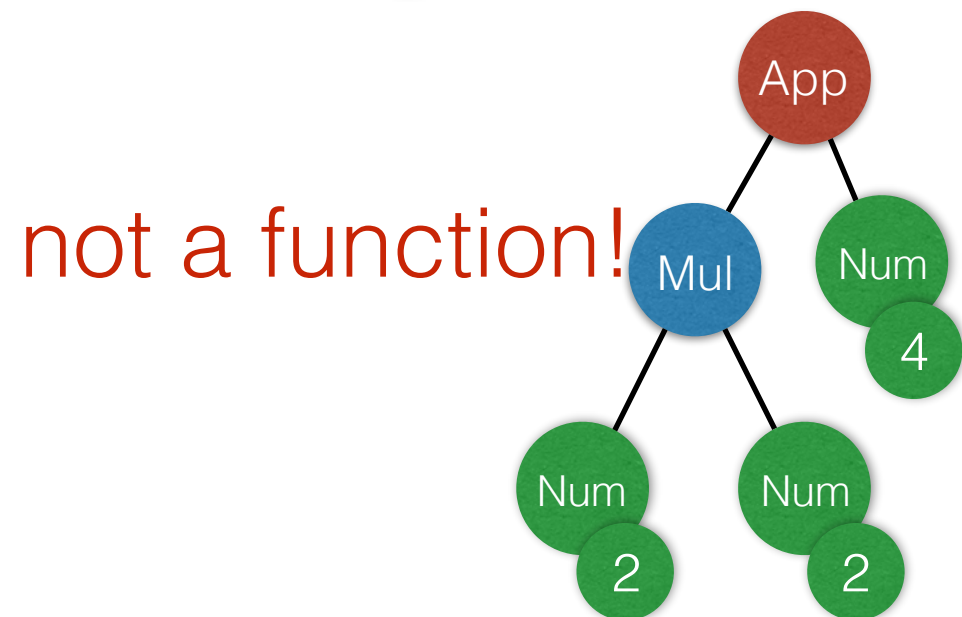
24

Type Systems



24

Type Systems

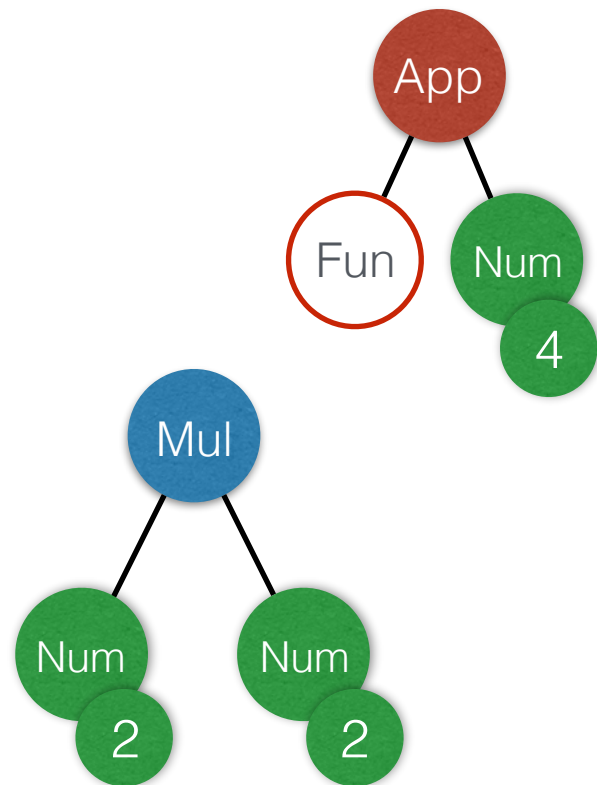


runtime type error!

24

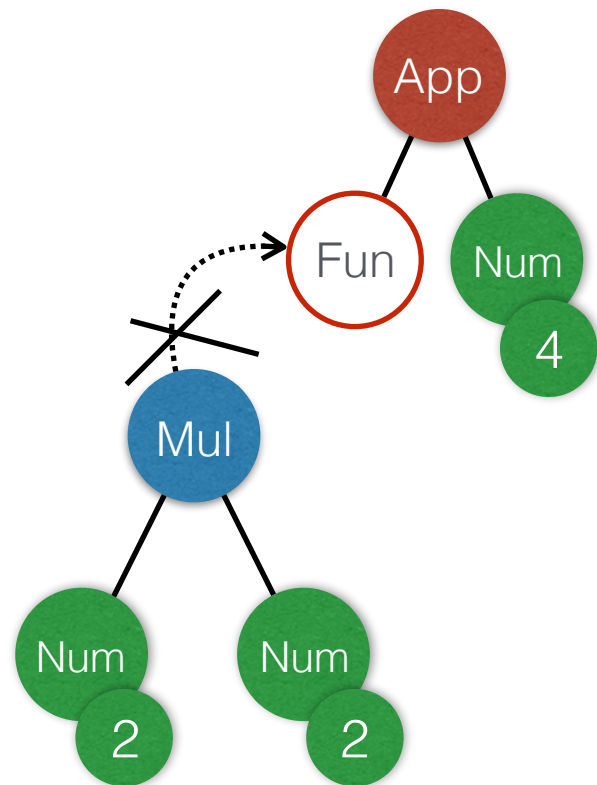
Type Systems

a solution: static type systems



24

Type Systems



a solution: static type systems

typechecking:
separate phase before
the program is run