CSE 130 [Winter 2014] Programming Languages

Environments & Closures

Ravi Chugh
UCSDCSE
Computer Science and Engineering

Jan 30

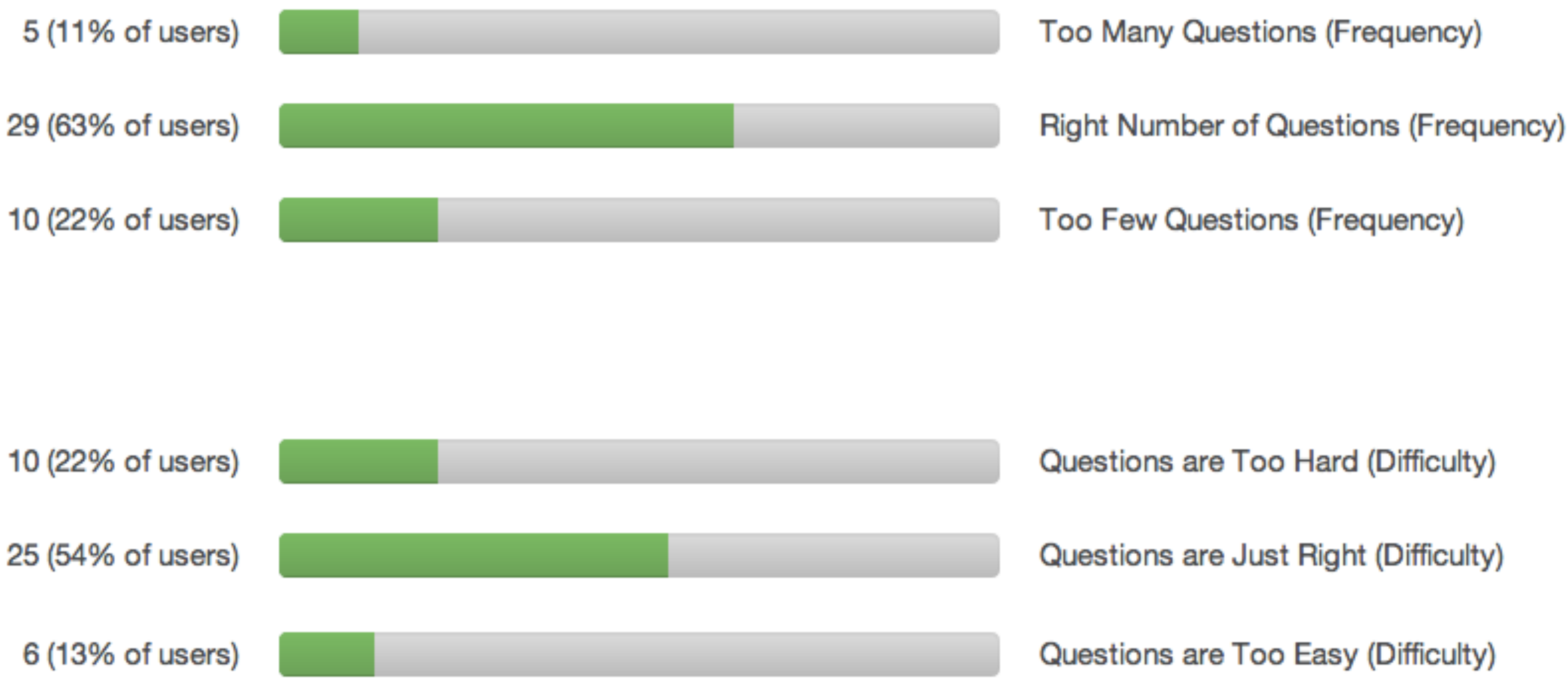
Announcements

- HW#1 Grades
  - Point breakdown by problem (by email)
  - All course grades (on Gradesource by secret number)

- Email TAs if you haven’t received these

Results: Poll on Clickers

~45 votes



Results: KQS Survey

~13 responses

KEEP Doing

- Pace of lectures
- Fair number of clicker questions
  - Enjoying clickers / discussion / talking with peers

QUIT Doing

- Group Seating
  - Sorry ☺ (But we can shuffle groups for 2nd half…)
- Explaining each wrong answer so slowly
- Waiting for last few votes

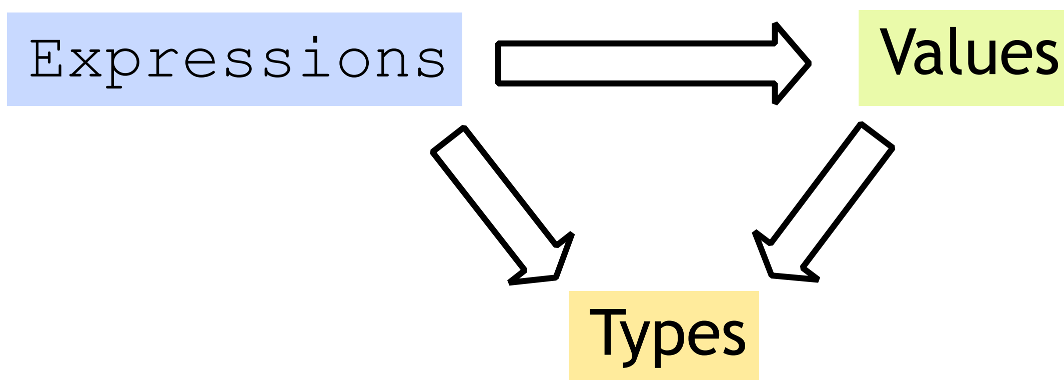
Results: KQS Survey

~13 responses

START Doing

- Timer countdown for clicker questions
  - Let's try **20 seconds** for vote after group discussion
- Posting slides before class
  - (I've been posting them ~1 to 3 hours before class)
- More full examples at interpreter
- Keep discussion section material same
- Highlighting answers to clicker questions
  - I'm still reluctant, because some are open-ended, buggy, and most can be evaluated in REPL

Higher-Order Functions
are awesome…
… but how do they work?



Let’s start with the humble variable…

Q: What is the value of res ?

```
let x   = 0       ;;
let y   = x + 1   ;;
let z   = (x, y)  ;;
let x   = 100     ;;
let res = z       ;;
```

(a) (0, 1)
(b) (100, 101)
(c) (0, 100)
(d) (1, 100)

Variables and Bindings

```
let x = e;;
```

"**Bind** value of **expr** *e* to **variable** x"

```
# let x = 2 + 2;;
val x : int = 4
# let y = x * x * x;;
val y : int = 64
# let z = [x; y; x+y];;
val z : int list = [4;64;68]
```

- Later on, **most recently** "bound" value **used** to evaluate x
- Sounds like C/Java ? **NO!**

Environments (“Phone Book”)

How ML deals with variables:

- Variables = “names”
- Values = “phone number”

`x1`	v1
`x2`	v2
`x3`	v3
`x4`	v4
…	…

Environments and Evaluation

ML begins in a "top-level" environment

- Phone book has entries for some names (e.g. `+`, `-`, `print_string`)

ML program = sequence of let-bindings

Repeat for each binding, in order:

1. Evaluate expr **`e`** in current env to get value ***v*** : ***t***
2. Extend env to bind **`x`** to ***v*** : ***t***

Environments and Evaluation

```
# let x = 2 + 2;;
val x : int = 4

# let y = x * x * x;;
val y : int = 64

# let z = [x; y; x+y];;
val z : int list = [4;64;68]

# let x = x + x ;;
val x : int = 8
```

…	…

…	…
x	4 : int

…	…
x	4 : int
y	64 : int

…	…
x	4 : int
y	64 : int
z	[4;64;68] : int list

…	…
x	4 : int
y	64 : int
z	[4;64;68] : int list
x	8 : int

New binding!
“Shadows” previous binding

Environments and Evaluation

"Phone Book"

- Variables = "names"
- Values = "phone number"

1. Evaluate:

- Find and use **most recent** value of variable

2. Extend:

- Add **new binding** at end of "phone book"
- **Old binding**, if any, persists but is "shadowed"

Environments and Evaluation

How is it different from C/Java's "store" ?

```
# let x = 2 + 2 ;;
val x : int = 4

# let f = fun y -> x + y ;;
val f : int -> int = <fun>

# let x = x + x ;;
val x : int = 8

# f 0 ;;
- : int = 4
```

…	…

…	…
x	4 : int

…	…
x	4 : int
f	<fun> : int -> int

…	…
x	4 : int
f	<fun> : int -> int
x	8 : int

New binding:

- No change or mutation
- Old binding frozen in `f`

Cannot Change the World

Can extend env with fresh binding

- Does not affect previous uses of variable
- That is, cannot "assign" to variables

Env at fun decl frozen inside <fun> value

- Frozen env used to evaluate application `f e`

Q: Why is this a good thing?

Bindings used to eval `(f …)`

Bindings not used to eval `(f …)`

…	…
`x`	4 : int
`f`	<fun> : int -> int
`x`	8 : int

Cannot Change the World

Can extend env with fresh binding

- Does not affect previous uses of variable
- That is, cannot “assign” to variables

Env at fun decl frozen inside <fun> value

- Frozen env used to evaluate application `f e`

Q: Why is this a good thing?

A: Function behavior frozen at declaration!

Immutability: The Colbert Principle

"A function behaves the same way on Wednesday as it behaved on Monday, *no matter what happened on Tuesday!*"

Cannot Change the World

Q: Why is this a good thing ?

A: Function behavior frozen at declaration

- Nothing evaluated later affects function
- **Same inputs** always produce **same outputs**
  - Localizes debugging
  - Localizes reasoning about the program
  - No “sharing” means no evil aliasing

No Sharing

No addresses/pointers $\Rightarrow$ No sharing/aliasing

- Each variable is bound to a different value

So tuples, lists, etc. are extremely inefficient?

No! Compiler's job is to optimize code

- Efficiently implement the "no-sharing" semantics
- There is sharing and pointers, but hidden from programmer

Programmer's job is to write…

- … correct, clean, readable, extendable systems
- Made easier by simplified semantics

Q: What is the value of res ?

```
let f x = 1;;
let f x = if x<2 then 1 else (x * f(x-1));;
let res = f 5;;
```

(a) 120

(b) 60

(c) 20

(d) 5

(e) 1

Function Bindings

Functions are values, can bind using `let`

```
let fname = fun x -> e
```

Problem: Can't define recursive functions!

- `fname` is bound after computing value on right-hand side
- No "old" binding for occurences of `fname` inside *e*

```
let rec fname x = e
```

Occurences of `fname` inside *e* bound to "this" definition

```
let rec fac x =
   if x<=1 then 1 else x * fac (x-1)
```

Local Bindings

So far: bindings remain "global" until a re-binding

Local, "temporary" variables are useful inside functions

- Avoid repeating computations
- Make functions more readable

```
let x = e1 in e2
```

Q: What is the value of res ?

```
let y = let x = 10 in
          x + x ;;

let res = (x, y);;
```

(a) Syntax Error

(b) `(10,20)`

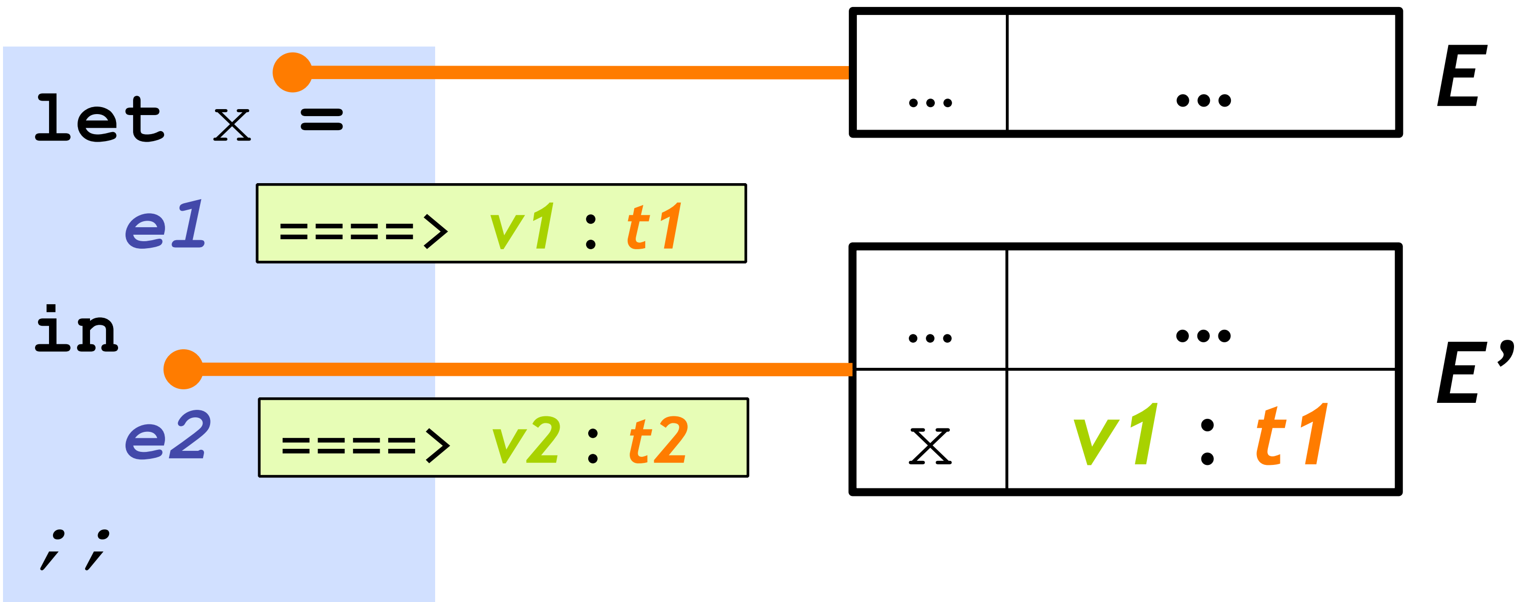
(c) `(10,10)`

(d) Type Error

Local Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***



Local Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***

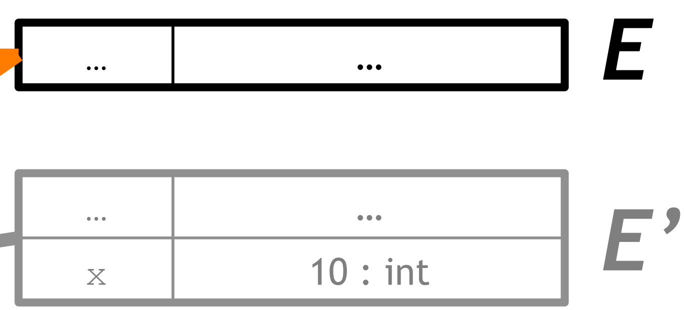
```
let x =
  10
in
  x * x
;;
```

Local Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***

```
let x =
  10
in
  x * x
;;
```



Local Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E’*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***

```
let y =
  let x =
    10
  in
    x * x
;;
```

E

…	…

E’

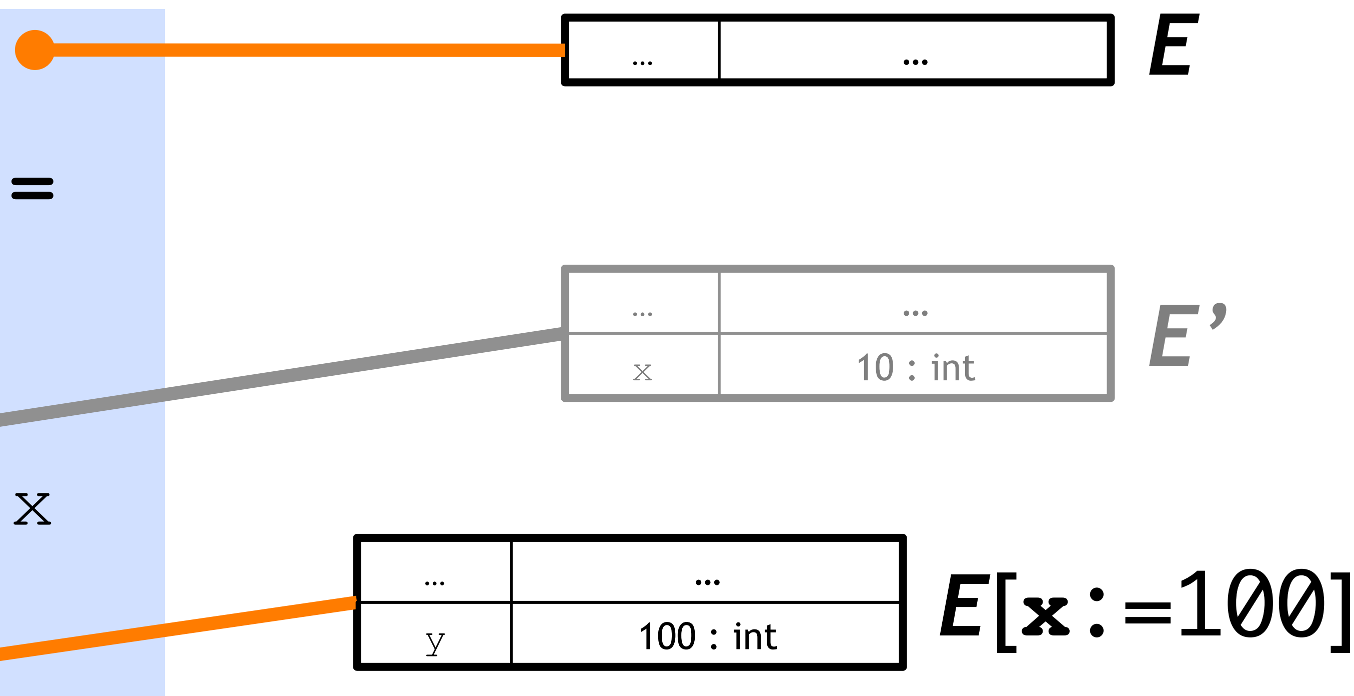
…	…
x	10 : int

Local Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***

```
let y =
  let x =
    10
  in
    x * x
;;
```

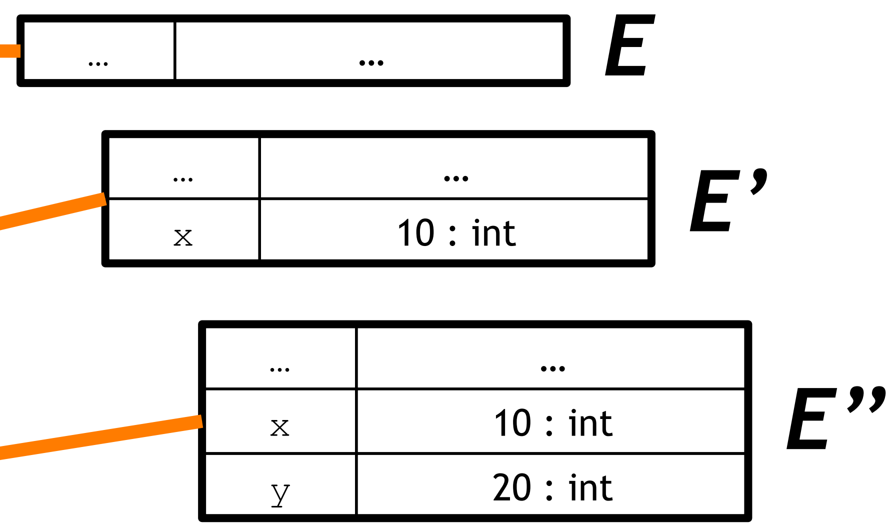


Nested Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***

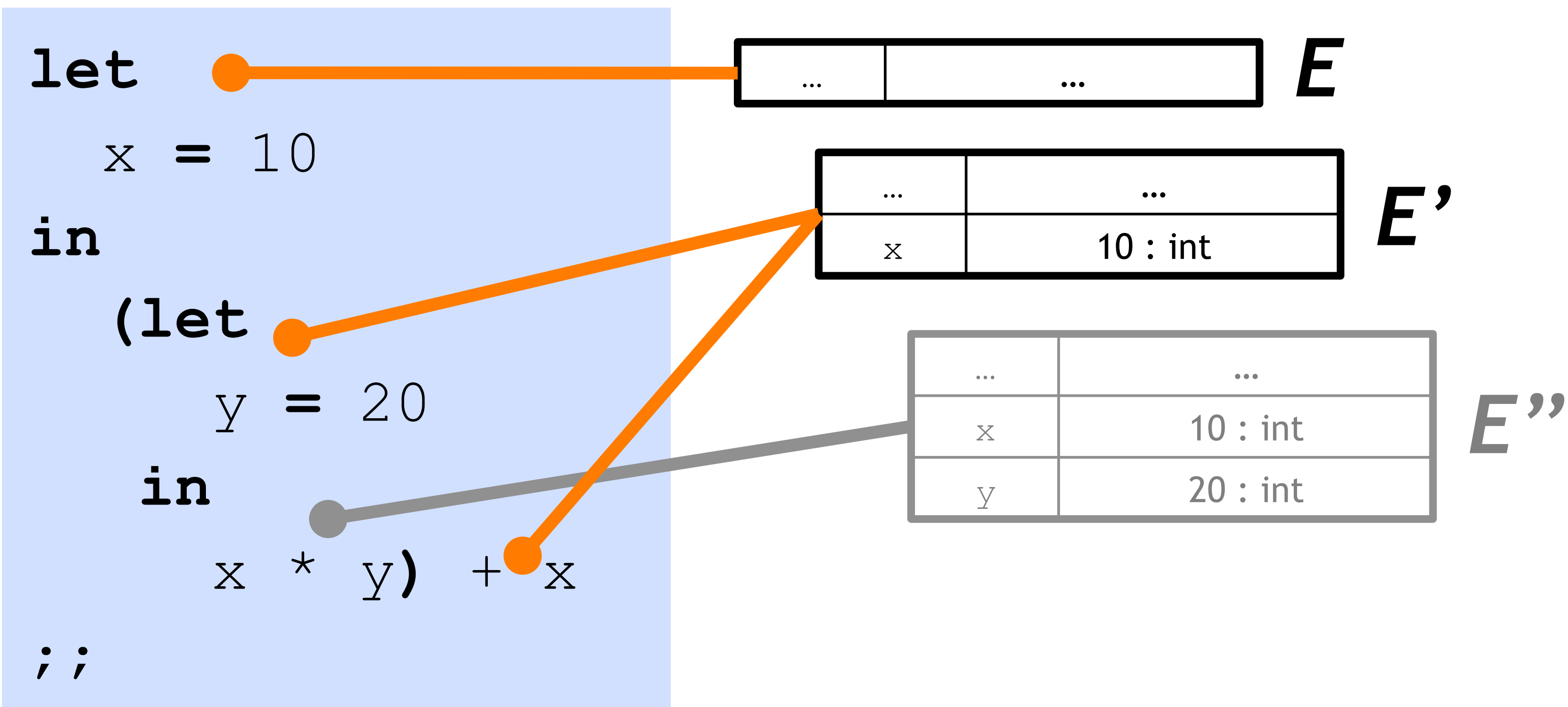
```
let
  x = 10
in
  (let
     y = 20
   in
     x * y) + x
;;
```



Nested Bindings

Evaluating **`let-in`** in env ***E***:

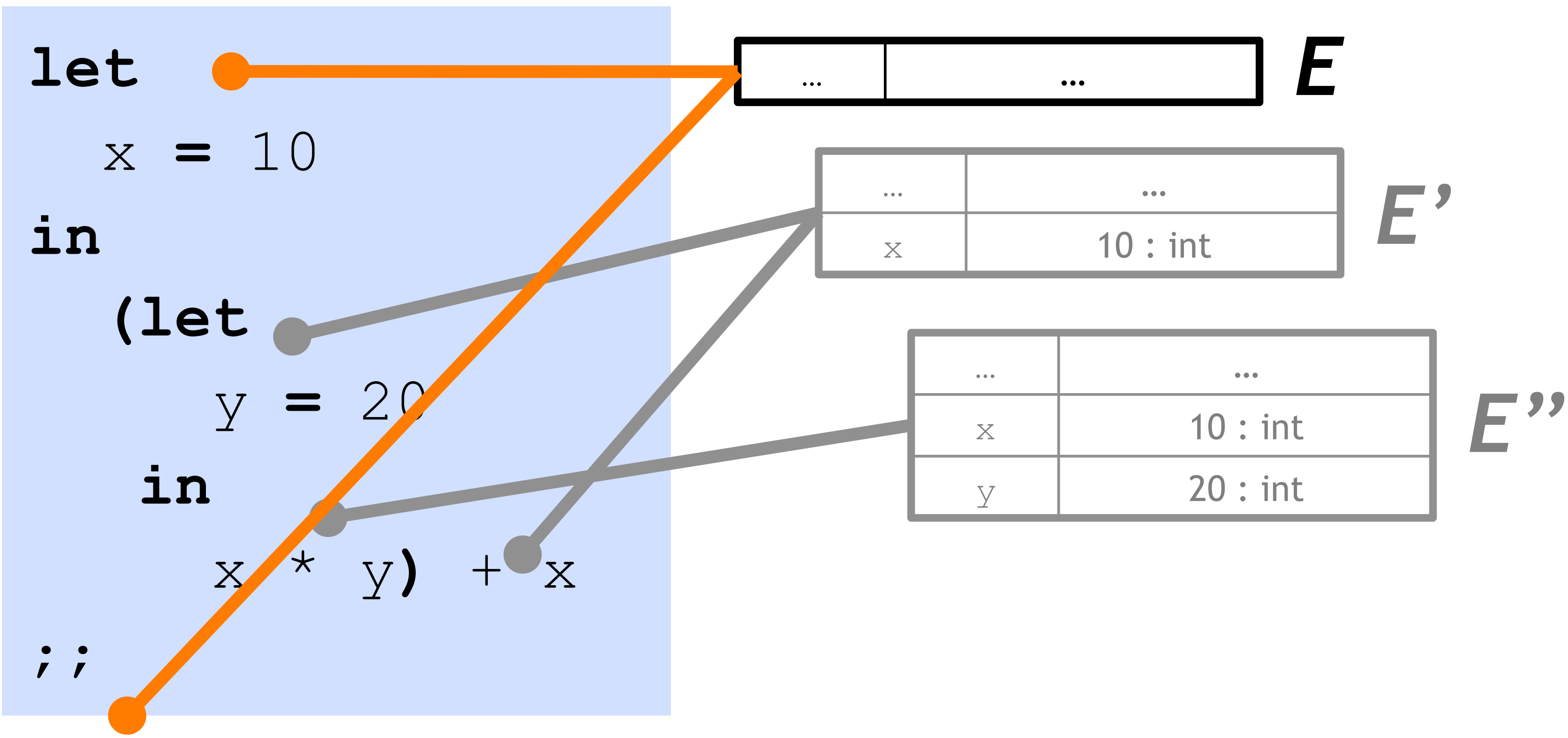
1. Evaluate expr ***e1*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**x** |-> ***v1*** : ***t1***] to evaluate ***e2***



Nested Bindings

Evaluating **`let-in`** in env ***E***:

1. Evaluate expr ***`e1`*** in env ***E*** to get value ***v1*** : ***t1***
2. Use extended ***E'*** = ***E*** [**`x`** |-> ***v1*** : ***t1***] to evaluate ***`e2`***



Nested Bindings

```
let
   x = 10
in
   let
      y = 20
   in
      x * y
;;
```

BAD Formatting

```
let x =
   10
in
   let y =
      20
   in
      x * y
;;
```

BAD Formatting

(except for HW 2 ☺)

```
let x = 10 in
let y = 20 in
   x * y
;;
```

GOOD Formatting

Example

```
let rec filter f xs =
  match xs with
  | []      -> []
  | x::xs’ -> let ys  = if f x then [x] else [] in
              let ys’ = filter f xs              in
              ys @ ys’
```

Recap: Environments

Variables are names for values

- Environment is dictionary/phonebook
- Most recent binding used
- Entries never change
- New entries added

Recap: Environments

Variables are names for values

Big expressions with local bindings

- **`let-in`** expression
- Variable "in scope" in **`in`**-expression
- Outside, variable not "out of scope"

Recap: Environments

Variables are names for values

Big expressions with local bindings

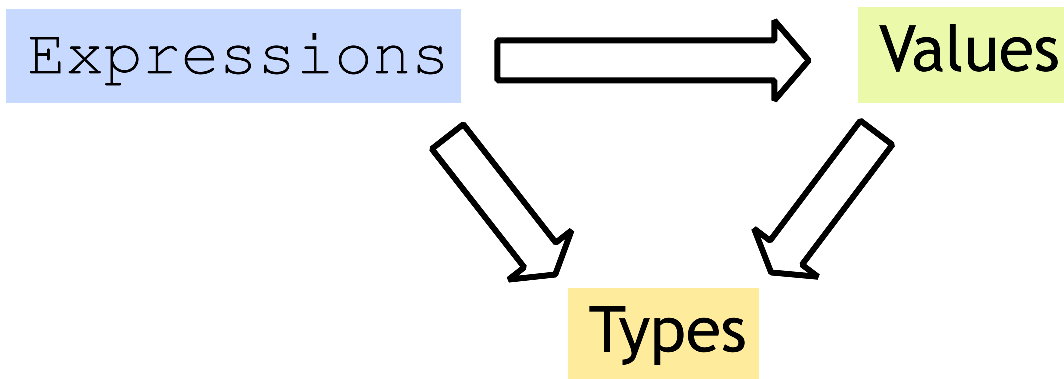
Env frozen at function definition

- Re-binding vars cannot change function
- Identical I/O behavior at every call
- Predictable code, localizes debugging

Recap: Environments

How is environment
“frozen into”
`<fun>`
?

Functions



Q: What's the value of a function ?

Functions

`Expressions`

Two ways of writing function expressions:

1. Anonymous functions:

Parameter (formal) | Body Expr

```
let fname = fun x -> e
```

2. Named functions:

Parameter (formal) | Body Expr

```
let fname x = e
```

Functions

Expressions

`x` is "in scope" in ***e***

```
fun x -> e === fun x -> (e)
```

```
fun x -> fun y -> e
===
fun x -> (fun y -> (e))
```

Function Application (or “Call”)

`Expressions`

“Apply” function value ***e1*** to argument ***e2***

```
e1 e2
```

Calls associate to the **left**:

```
e1 e2 === (e1 e2)
e1 e2 e3 === ((e1 e2) e3)
e1 (e2 e3) === (e1 (e2 e3))
```

Functions

Types

Every function has a type of the form:

- *T1* : the type of the "input"
- *T2* : the type of the "output"

$T1 \rightarrow T2$

```
let fname = fun x -> e
```

$T1 \rightarrow T2$

$T1 \rightarrow T2$

```
let fname x = e
```

Functions

Types

Every function has a type of the form:

- *T1* : the type of the "input"
- *T2* : the type of the "output"

$T1 \rightarrow T2$

T1, *T2* can be any types, including function types!

Whats an example of ?

- *int -> int*
- *int * int -> bool*
- *(int -> int) -> (int -> int)*

Functions

Types

Every function has a type of the form:

- *T1* : the type of the "input"
- *T2* : the type of the "output"

$$T1 \rightarrow T2$$

Function types associate to the **right**:

$$T1 \rightarrow T2 \rightarrow T3 \;\; === \;\; (T1 \rightarrow (T2 \rightarrow T3))$$

$$(T1 \rightarrow T2) \rightarrow T3 \;\; === \;\; ((T1 \rightarrow T2) \rightarrow T3)$$

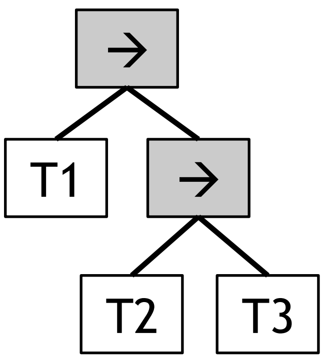
$$(T1 \rightarrow T2) \rightarrow T3 \rightarrow T4 \;\; === \;\; ((T1 \rightarrow T2) \rightarrow (T3 \rightarrow T4))$$

Functions

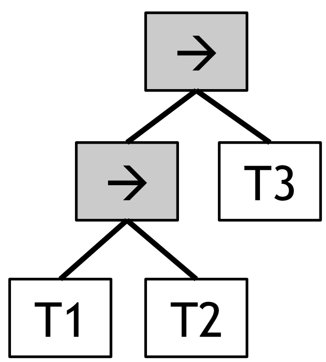
Types

Think of function types as trees

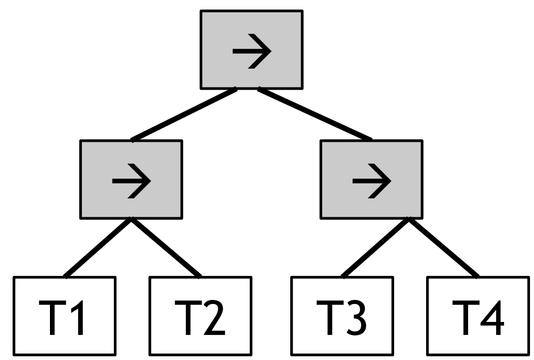
T1 → T2 → T3



(T1 → T2) → T3



(T1 → T2) → T3 → T4



Type of Function Application

`(e1 e2)`

"Apply" function *e1* to argument *e2*

or "Call" function *e1* with argument *e2*

or "Pass" argument *e2* to function *e1*

$$\frac{e1 : T1 \rightarrow T2 \qquad e2 : T1}{(e1\ e2) : T2}$$

Argument must have same type as "input" *T1*

Result has the same type as "output" *T2*

Functions

Values

Two questions:

What is the value:

1. … of a function ? `fun x -> e`

2. … of a function “application” (call) ? `(e1 e2)`

Values of Functions = “Closures”

Two questions:

What is the value:

1. … of a function ? `fun x -> e`

Closure =

Code of Fun. (formal **x** + body **e**)

\+ Environment at Fun. Definition

Values of Functions = “Closures”

- “Closure” = <code + environment at definition>
- Type checking when function is defined
- Body not evaluated until application

```
# let x = 2+2;;
val x : int = 4
# let f = fun y -> x + y;;
val f : int -> int = fn
# let x = x + x;;
val x : int = 8
# f 0;;
- : int = 4
```

Binding used to evaluate `f`

…	…
x	4 : int
f	<fun> : int -> int
x	8 : int

<fun>
“closes over”
all previous
bindings

Binding for subsequent uses of `x`

Q: Which vars in closure of f ?

```
let x   = 2 + 2 ;;
let f y = x + y ;;
let z   = x + 1 ;;
```

(a) `x`

(b) `y`

(c) `z`

(d) `x y z`

(e) None

Q: Which vars in closure of f ?

```
let a = 20;;

let f x =
   let y    = 1 in
   let g z = y + z in
     a + (g x)
;;
```

(a) a y

(b) a

(c) y

(d) z

(e) y z

Functions

Values

Two questions:

What is the value:

1. … of a function ? `fun x -> e`

2. … of a function “application” (call) ? `(e1 e2)`

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Environment frozen inside function definition

Used to evaluate function application `(e1 e2)`

Which vars are needed from frozen env?

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Inside a function,

A "bound" occurrence:

1. Formal variable
2. Variable bound in `let-in`

`x`, `y`, `g` are "bound" inside `f`

A "free" occurrence:

- An unbound variable

`a` is "free" inside `f`

Frozen Environment

needed for values of free vars

Free vs. Bound Variables

```
let a = 20;;

let f x =
  let y = 1 in
  let g z = y + z in
    a + (g x)
;;

f 0;;
```

Bound values determined when function is evaluated ("called")

- Arguments
- Local variable bindings

Values of Function Application

Value of a function "application" (call) `(e1 e2)`

1. Evaluate ***e1*** in current-env to a **closure**
   = **code** (formal **x** + body ***e***) **+ closure-env** ***E***

2. Evaluate ***e2*** in current-env to get (argument) *v2*

3. Evaluate body ***e*** in env ***E*** extended with **x** := *v2*

Q: What is the value of `res` ?

```
let f g =
   let x = 0 in
   g 2
;;

let x = 100;;

let h y = x + y;;

let res = f h;;
```

(a) Syntax Error
(b) 102
(c) Type Error
(d) 2
(e) 100

Example

```
let f g =
   let x = 0 in
   g 2
;;

let x = 100;;

let h y = x + y;;

f h;;
```

Immutability: The Colbert Principle

"A function behaves the same way on Wednesday as it behaved on Monday, *no matter what happened on Tuesday!*"

Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment
- Static/Lexical: Determined from the program text
  - Without executing the program
- Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence