CS 162 Programming languages

Lecture 14: Environment and Closure

Yu Feng

Winter 2021

Variables and bindings

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

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

“Bind the value of expression e to the variable x”

Variables and bindings

```
# 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]
```

Most recent “bound” value used for evaluation

Sounds like C/Java? No!

Environment and evaluation

ML begins in a "top-level" environment

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

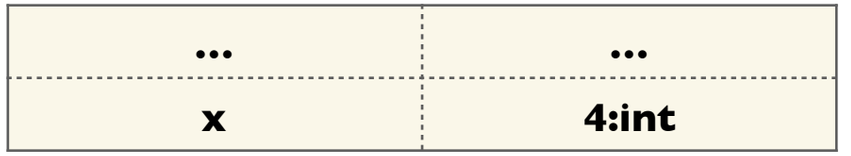
ML program = Sequence of variable bindings

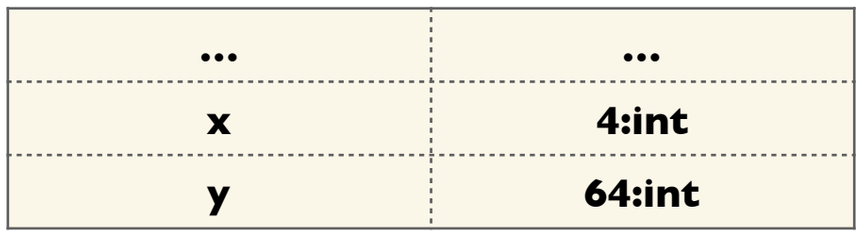
Program evaluated by evaluating bindings in order

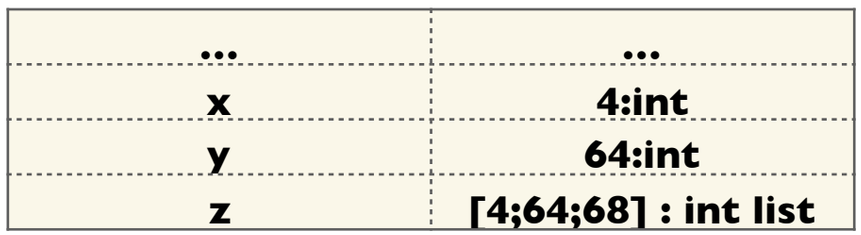
1. Evaluate expr e in current env to get value v
2. Extend env to bind x to v
3. Repeat with next binding

Example

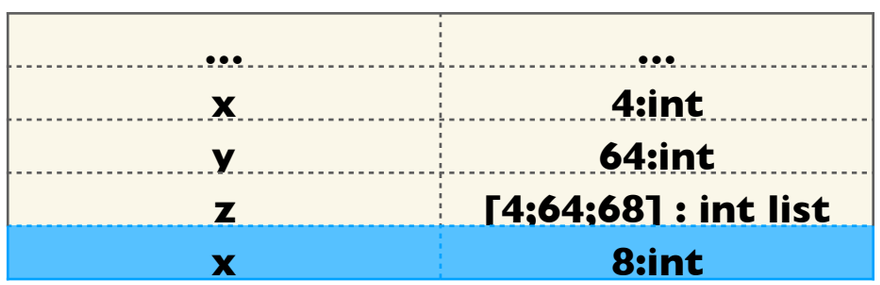
```
# let x = 2+2;;
val x : int = 4


# 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
```







New binding! →



Environments

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

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

```
# 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;
val it : int = 4
```

...	...
x	4:int

...	...
x	4:int
f	fn <code, *>: int->int

New binding:

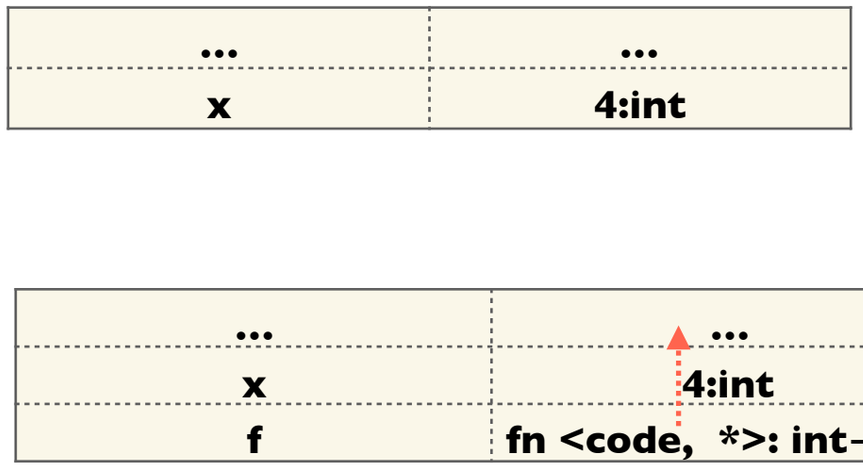
- No change or mutation
- Old binding frozen in **f**

Environments

1. Evaluate: Use most recent bound value of var
2. Extend: Add new binding at end

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

```
# 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;
val it : int = 4
```



Cannot change the world

Cannot "assign" to variables

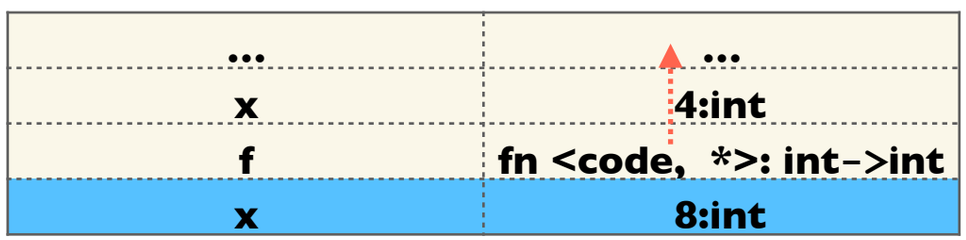
- Can extend the env by adding a fresh binding
- Does not affect previous uses of variable

Environment at fun declaration frozen inside fun "value"

- Frozen env used to evaluate application (f …)

Why is this a good thing?

```
# 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;
val it : int = 4
```



Cannot change the world

Function behavior frozen at declaration

Q: Why is this a good thing?

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

Functions

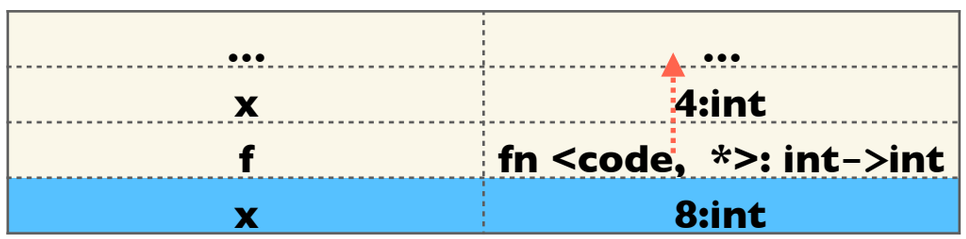
Type `f : T1 -> T2`

f takes a value of type T1
and returns a value of type T2

Value of function: Closure

- "Body" expression not evaluated until application
  - but type-checking takes place at compile time
  - i.e. when function is defined
- Function value =
  - (code + environment at definition)
    - "closure"

```
# 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;
val it : int = 4
```



Values of function application

Application: function call or β-reduction

$(e_1\ e_2)$

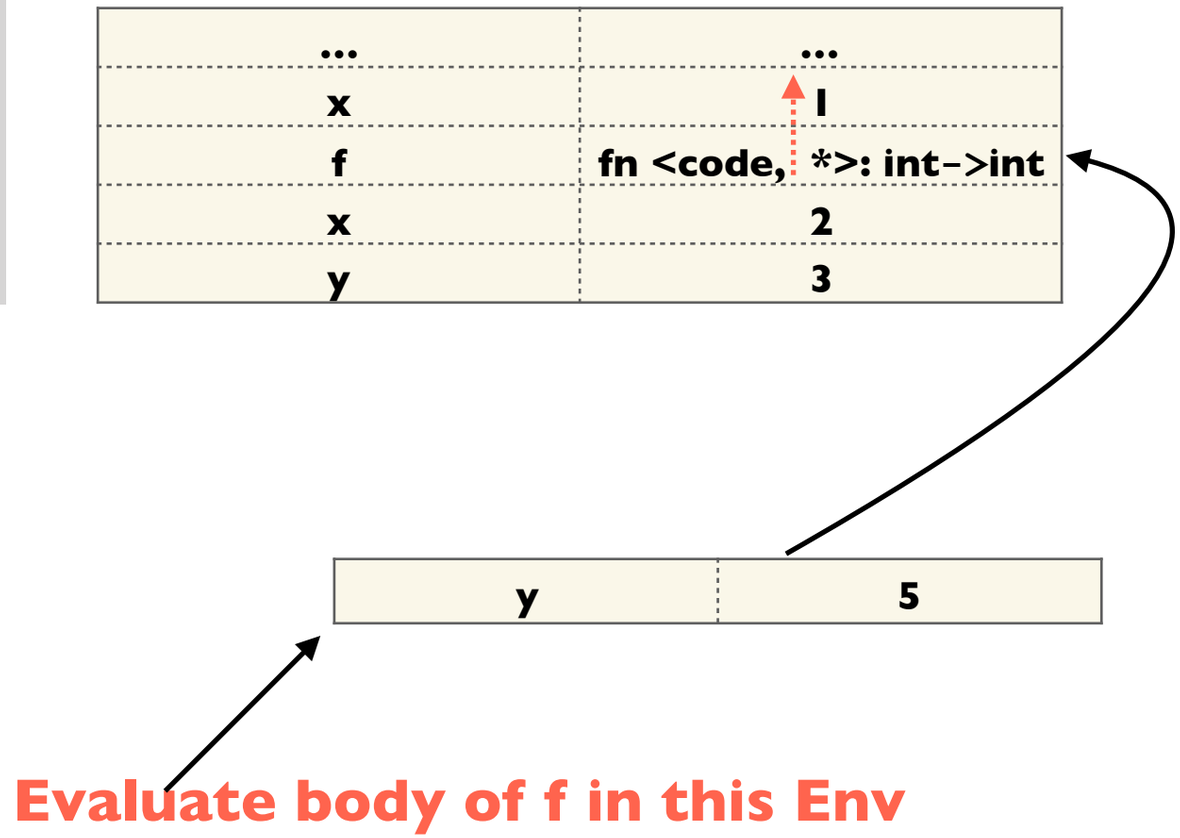
Apply argument e_2 to function e_1

Application Value:

1. Evaluate e_1 in current env to get (function) v_1
   - v_1 is code + env
   - code is (formal x + body e) , env is E
2. Evaluate e_2 in current env to get (argument) v_2
3. Evaluate body e in env E extended by binding x to v_2

Example 1

```
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```



Example 2

```
let x = 1;;
let f y =
  let x = 2 in
  fun z -> x + y + z
;;

let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
```