

## Scope and Memory Management (part 2)

CSCI 334  
Stephen Freund

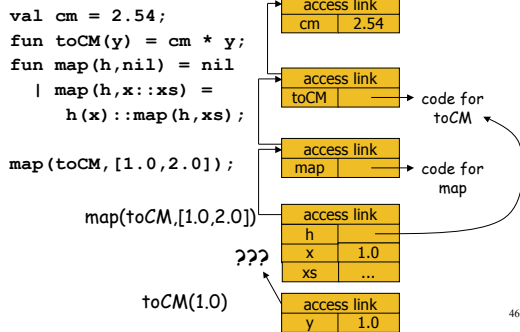
44

## Higher-Order Functions and Storage

- Functions passed as arguments
- Functions that return functions from nested blocks
- Same issues for declarations
  - scope / access
  - lifetime
- Data structures and implementation form foundation for objects

45

## Passing Functions to Functions



46

## Function Values are Closures

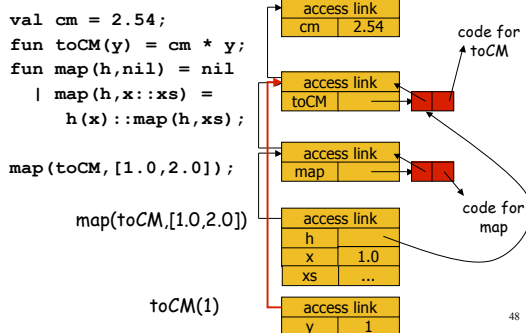
- *closure* =  $(env, code)$ 
  - env is pointer to activation record for scope in which function is declared
  - code is pointer to start of instructions

- When function is called, set the access link using the environment pointer from the closure



47

## Closures



48

## Summary of Function Arguments

- Closure maintains pointer to static environment of a function body
- When called, access link set from closure
- All access links point "up" in the stack
  - can still deallocate activation records in lifo order

49

## makeRand

```
fun makeRand(seed1, seed2) =
  let val generator = Random.rand(seed1, seed2)
  in
    fn (x,y) =>
      Random.randRange(x,y) (generator)
  end;
```

50

## Returning Functions From Functions

```
fun make(seed) =
  let val count = ref seed
      fun next(n) =
          (count := !count + n;
           !count)
      in
        next
      end;
  val c = make(1);
  c(1) -> returns 2
  c(2) -> returns 4
  c(2) -> returns 6
```

51

## Returning Functions From Functions

```
fun make(seed) =
  let val count = ref seed
      fun next(n) =
          (count := !count + n;
           !count)
      in
        next
      end;
  val c = make(1);
  c(1) + c(2);
```

count is free var

- "make" returns a closure
- How is correct value of count found in c(1)?

52

## Function Results and Closures

```
fun make(seed) =
  let val count = ref seed
      fun next(n) =
          (count := !count + n;
           !count)
      in
        next
      end;
  val c = make(1);
  c(1) + c(2);
```

code for make

code for next

make(1)

53

## Function Results and Closures

```
fun make(seed) =
  let val count = ref seed
      fun next(n) =
          (count := !count + n;
           !count)
      in
        next
      end;
  val c = make(1);
  c(1) + c(2);
```

code for make

code for next

Deallocating AR is bad...

54

## Function Results and Closures

```
fun make(seed) =
  let val count = ref seed
      fun next(n) =
          (count := !count + n;
           !count)
      in
        next
      end;
  in
    add
  end;
  val c = make(1);
  c(1) + c(2);
```

code for make

code for next

c(1)

(Right before call to c(1) ends)

55

### Summary of Returning Functions

- Closure maintains static environment
- May need to keep activation records after return
  - Stack (lifo) order fails!
- Possible "stack" implementation
  - Forget about explicit deallocation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as totally crazy as it sounds

56