Lecture #25: Runtime Structures for Functions, Contd.

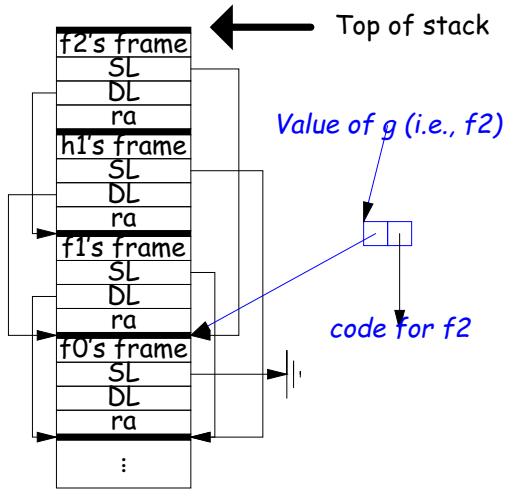
5. Allow Function Values, Properly Nested Access

- In C, C++, no function nesting.
- So all non-local variables are global, and have fixed addresses.
- Thus, to represent a variable whose value is a function, need only to store the address of the function's code.
- But when nested functions possible, function value must contain more.
- When function is finally called, must be given value of its static link is.
- Assume first that access is properly nested: variables accessed only during lifetime of their frame.
- So can represent function with address of code + the address of the frame that contains that function's definition.
- It's environment diagrams again!!

Function Value Representation

```
def f0 (x):
   def f1 (y):
     def f2 (z):
       return x + y + z
     print h1 (f2)
   def h1 (g): g (3)
   f1 (42)
```

- Call `f0` from the main program; look at the stack when `f2` finally is called (see right).
- When `f2`'s value (as a function) is computed, current frame is that of `f1`. That is stored in the value passed to `h1`.
- Easy with static links; global display technique does not fare as well [why?]

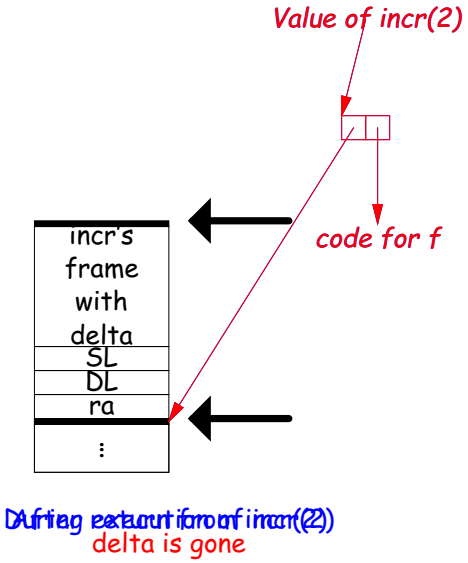


6. General Closures

- What happens when the frame that a function value points to goes away?
- If we used the previous representation (#5), we'd get a *dangling pointer* in this case:

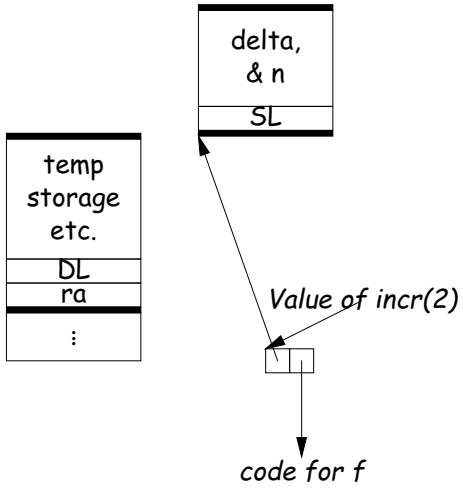
```
def incr (n):
   delta = n
   def f (x):
     return delta + x
   return f

p2 = incr(2)
print p2(3)
```



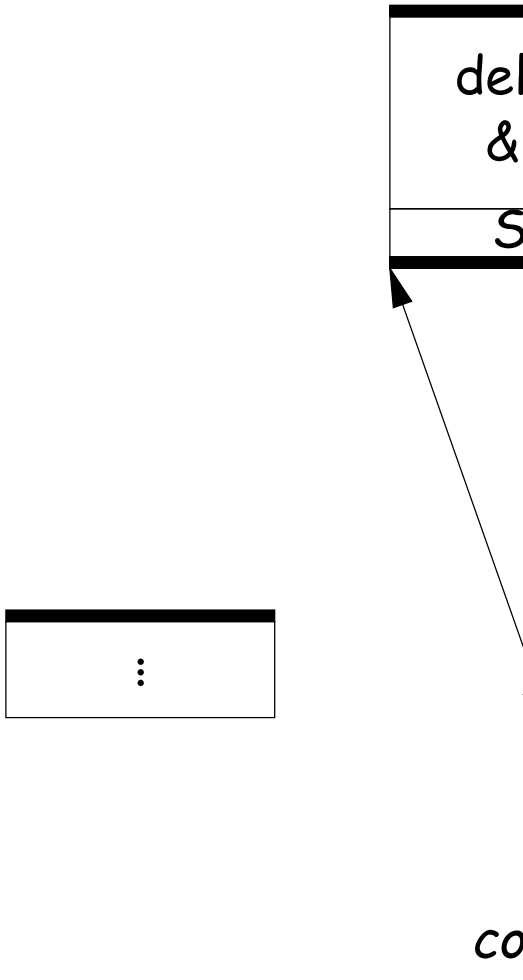
Representing Closures

- Could just forbid this case (as some languages do):
  - Algol 68 would not allow pointer to f (last slide) to be returned from incr.
  - Or, one could allow it, and do something random when f (i.e. via delta) is called.
- Scheme and Python allow it and do the right thing.
- But must in general put local variables (and a static link) in a record on the heap, instead of on the stack.



Representing Closures

- Could just forbid this case (as some languages do):
  - Algol 68 would not allow pointer to f (last slide) to be returned from incr.
  - Or, one could allow it, and do something random when f (i.e. via delta) is called.
- Scheme and Python allow it and do the right thing.
- But must in general put local variables (and a static link) in a record on the heap, instead of on the stack.
- Now frame can disappear harmlessly.



7. Continuations

- Suppose function return were not the end?

```
def f (cont): return cont
x = 1
def g (n):
  global x, c
  if n == 0:
     print "a", x, n,
     c = call_with_continuation (f)
     print "b", x, n,
  else: g(n-1); print "c", x, n,
g(2); x += 1; print; c()
```

```
# Prints:
#  a 1 0 b 1 0 c 1 1 c 1 2
#  b 2 0 c 2 1 c 2 2
#  b 3 0 c 3 1 c 3 2
...
```

- The *continuation*, c, passed to f is "the function that does whatever is supposed to happen after I return from f."
- Can be used to implement exceptions, threads, co-routines.
- Implementation? Nothing much for it but to put all activation frames on the heap.
- Distributed cost.
- However, we can do better on special cases like exceptions.

Summary

Problem	Solution
1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler, first-class function values.	Use inline expansion or use static variables to hold return addresses, locals, etc.
2. #1 + recursion	Need stack.
3. #2 + Add variable-sized unboxed data	Need to keep both stack pointer and frame pointer.
4. #3 – first-class function values + Nested functions, up-level addressing	Add static link or global display.
5. #4 + Function values w/ properly nested accesses: functions passed as parameters only.	Static link, function values contain their link. (Global display doesn't work so well)
6. #5 + General closures: first-class functions returned from functions or stored in variables	Store local variables and static link on heap.
7. #6 + Continuations	Put everything on the heap.