Lecture 25: Proving termination/implementing functions

- Termination of loops
- Function calls
  - Conventional (review)
  - Higher-order functions
  - Virtual functions (object-oriented languages)

Proving termination

Weird property of the Hoare proof system: It is possible to "prove" non-terminating programs.

Can prove: *a = a0 & b = b0* { while (a != b)

gcd(a,b) = gcd(a0,b0) →

if (a<b) b := b-a;

} *a = gcd(a0,b0)*

Judgments in Hoare logic are assertions about partial correctness: P{A}Q means "if the state satisfies P, then after executing A, *if A terminates*, the state will satisfy Q." If A doesn't terminate the judgment is vacuously true.

Proving termination

Total correctness means A will satisfy its specification (i.e. its partial correctness formula) *and* will definitely terminate.

Total correctness is usually proven in two separate steps: (1) Prove partial correctness; (2) Prove termination.

Proving termination of loops

Obviously, the only place where non-termination is possible is in loops.

To prove termination of a loop: Define a function ϕ: program states $\rightarrow$ non-negative integers. Prove: For every iteration of the loop, ϕ(the current state) < ϕ(the previous state). As long as ϕ is correctly defined as a function whose values are non-negative integers, then the loop cannot go on forever.

Termination proof examples

- sum of n
- fibonacci
- list append
- list reverse

Sum of n

Assume $n \geq 0$:

$x = 0$ & $y = 0$

```
{
  while (y < n) {
    y := y + 1;
    x := x + y
  }
}
```

$x = 1 + \ldots + n$

$\varphi(x, y, n) = n - y$ ✓

- $\varphi(x, y, n) \geq 0$
- If x_0, y_0, n_0 are value at start of loop body, and x, y, n value at end, $\varphi(x, y, n) < \varphi(x_0, y_0, n_0)$

Fibonacci

$x = 0$ & $y = 1$ & $z = 1$ & $1 \le n$

```
{
   while (z < n) {
      y := x + y;
      x := y – x;
      z := z + 1;
   }
}
```

$y = fib\ n$

$\varphi(x, y, n, z) = n - z$

- $\varphi(\quad) \geq 0$
- $\varphi(x, y, n, z) < \varphi(x_0, y_0, n_0, z_0)$

List length

x = lst & y = 0

```
{
  while (x ≠ []) {
    x := tl x;
    y := y + 1;
  }
}
```

y = len lst

$\varphi(x, y, lst)$

$= |x|$

List reverse

x = lst & y = []

```
{
   while (x ≠ []) {
      y := hd x :: y;
      x := tl x;
   }
}
```

y = rev lst

$\varphi(x, y, lst)$

$= |lst| - |y|$

or $= |x|$

flatten : (int list) list → int list

(underbrace under "(int list) list": x)

flatten [] = []

| (x :: xs) :: ys = x :: flatten (xs :: ys)

| [] :: ys = flatten ys

flatten [[0, 1, 2], [3, 4, 5], ...]

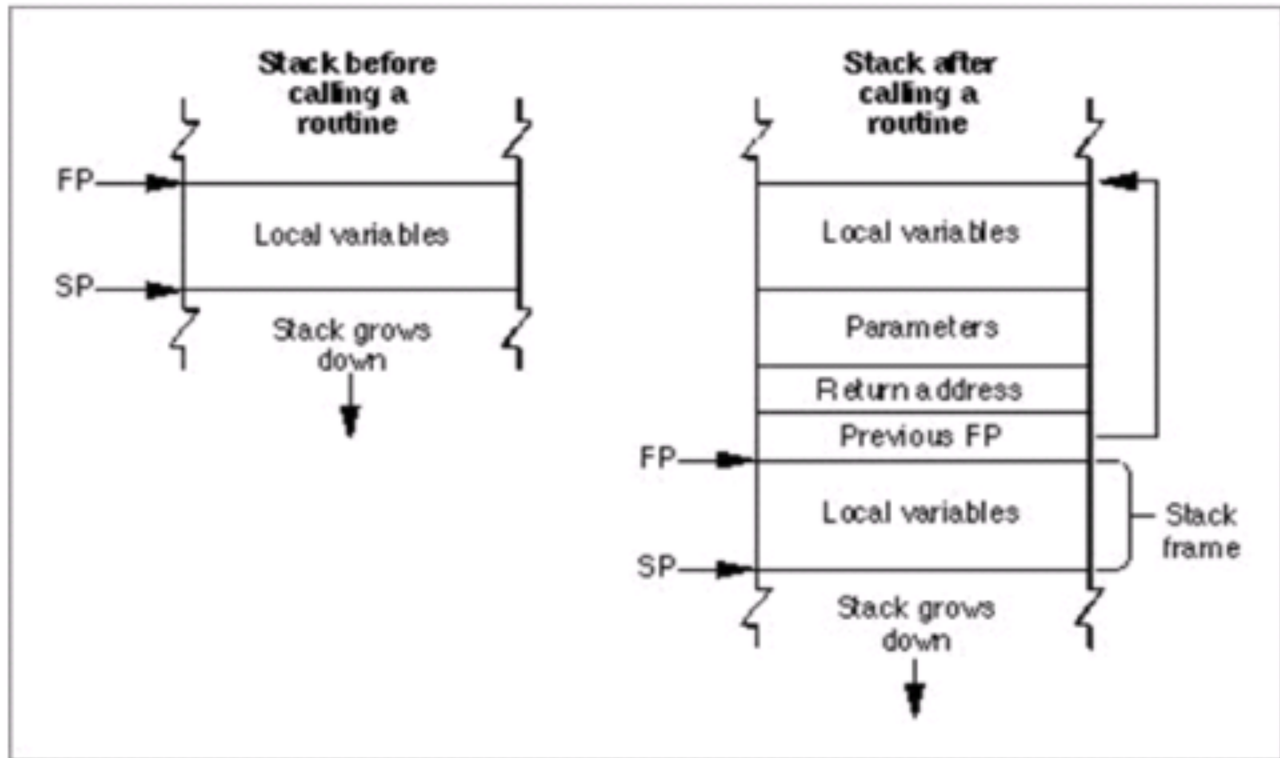
→ 0 : flatten [[1, 2], [3, 4, 5], ...]

Q (x) = ~~|x|~~ = total # of int's in x

Function calls

- Conventional functions:
  - Stack-like function call/return
  - Stack frame contains: parameters, local variables, return address, etc.
  - Offsets of variables within stack frame known statically
- Higher-order functions: environment (bindings of variables) outlives function call
- Virtual functions: bound at run time

Run-time environment – stack structure



Higher-order functions

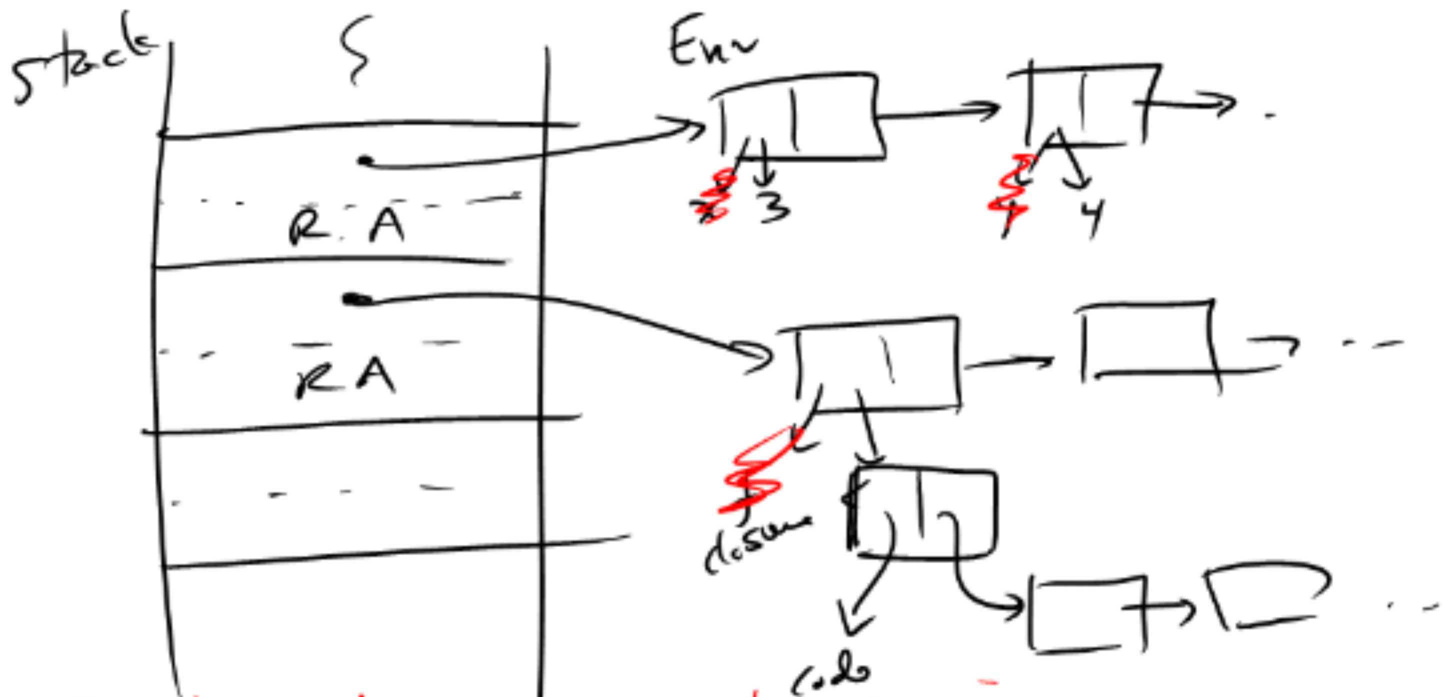
(For simplicity, assume one argument per function, no local variables – i.e. like OCaml.)

Use implementation that mimics operational semantics:

- Environments are pairs stored in heap – value of parameter and pointer to previous environment
- Closures are pairs stored in heap – address of code for function and pointer to environment
- Stack frame contains just return address and address of environment (also saved registers, but we'll ignore those)

((fun x -> fun y -> x + y) 3) 4

Implementing higher-order functions – follow op. sem.



Can't put env on stack in a language that has higher-order functions

Closure semantics

Follow env pointers a number of times = distance between ref. and declaration

Variable reference: $\eta, x \Downarrow \eta x$

→ Find x at "correct location" in current location

Closure formation: η, fun $x \to e \Downarrow$ <fun $x \to e, \eta$>

→ Allocate pair, fill first half w/ address of compiled code for fun $x \to e$, second half with current env.

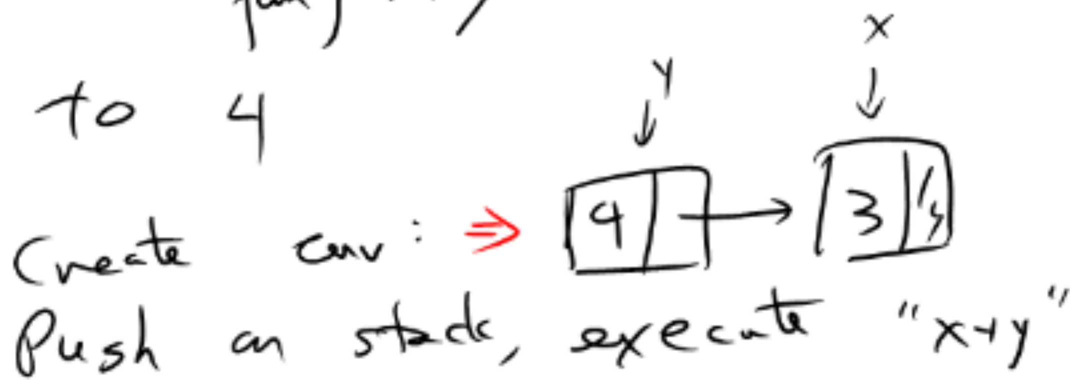
Application: Given address c of closure [f | η] and value v: Form new env: [v |] → η, push on stack along w/ r.a., jump to f.

Locations of variables:

((fun x → fun y → x+y) 3) 4

fun y → x+y

to 4



Push on stack, execute "x+y"

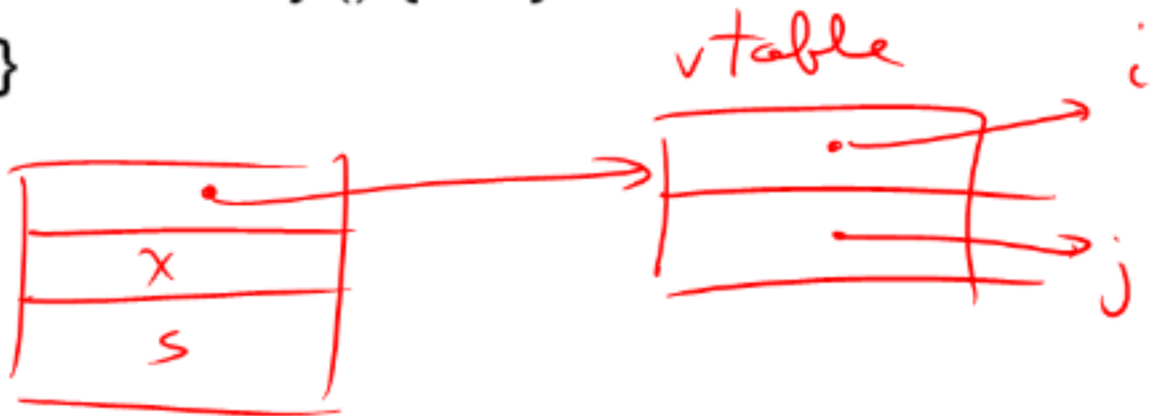
Implementing virtual functions

- Calls to virtual functions (those declared "virtual" in C++, all methods in Java) are always indirect – they go through a "virtual function table."
- Objects contain fields and pointer to virtual function table.
- Key point: *when compiling any method, the location of every other method's pointer, within its v.f.t., is a static number.*

A class C

```
class C { int x; String s;
          method  i () { … }
          method  j () { … }
        }
```

a C object



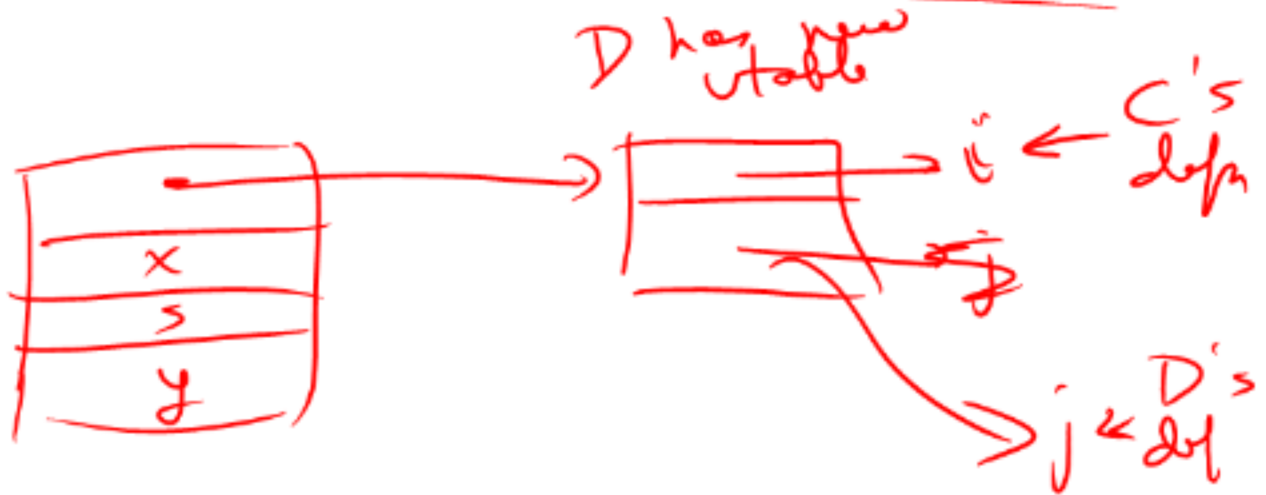
D extends C by adding fields

class D extends C { int y; }

a D object

D extends C by adding fields, and redefining methods

```
class D extends C { int y;   method j () {…} }
```



D extends C by adding fields, redefining methods, and adding new methods

```
class D extends C { int y;
                    method j  () { … }
                    method k () { … } }
```