Lecture 26: 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)

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

$x = 0 \ \& \ y = 0$

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

$x = 1 + ... + n$

Fibonacci

x = 0 & y = 1 & z = 1 & 1 ≤ n

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

y = fib n

List length

$x = lst \;\&\; y = 0$

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

$y = len\ lst$

List reverse

x = lst & y = []

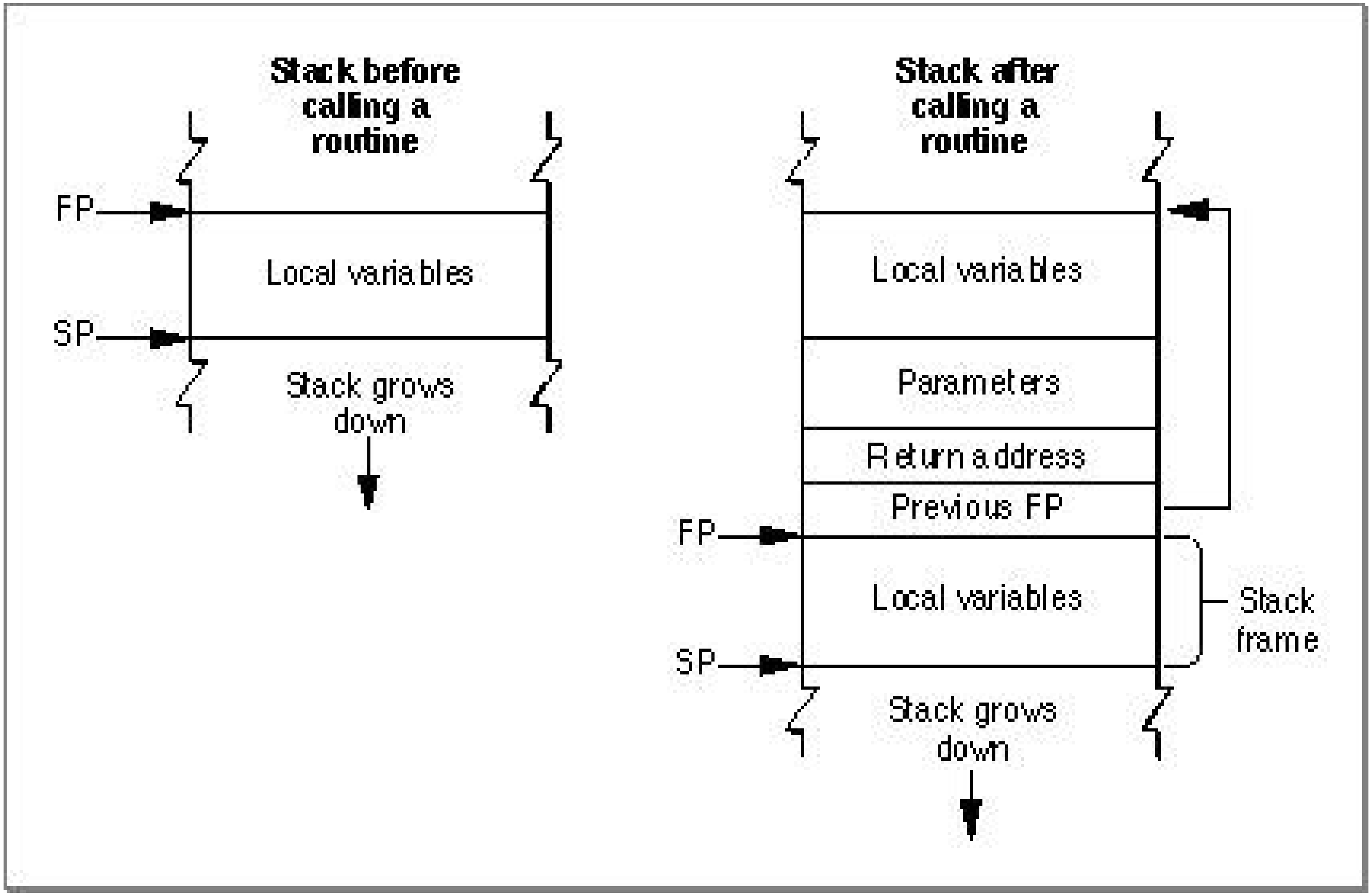
```
{
    while (x ≠ []) {
        y := hd x :: y;
        x := tl x;
    }
}
```

y = rev lst

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)

Implementing higher-order functions – follow op. sem.

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 () { … }
          }
```

D extends C by adding fields

```
class D extends C { int y; }
```

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 () { … } }
```