Compiler Construction
Lent Term 2013
Lecture 12 (of 16)

- **Implementing Slang.2 functions in the VSM**
- **L3-specific details require some extra effort**
  - **e1(e2)**
- **Why VRM is more difficult**
  - **Register allocation**
  - **Calling conventions**

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

Slang.2 concrete syntax
(Extend BAD SYNTAX from Lecture 11)

NEW

```
expr := simple
| set expr := expr
| while expr do expr
| if expr then expr else expr
| begin expr expr_list
| let var : type_expr = expr in expr end
| fn var : type_expr => expr
| fun var (var : type_expr) : type_expr = expr in expr end

                    . . .

 factor := identifier
 | integer | - expr | ~ expr | true| false
 | skip | ref expr | ! Expr | ( expr ) | print expr
 | apply expr expr  (* ugly? yes! *)
```

Calling functions : direct vs. closure

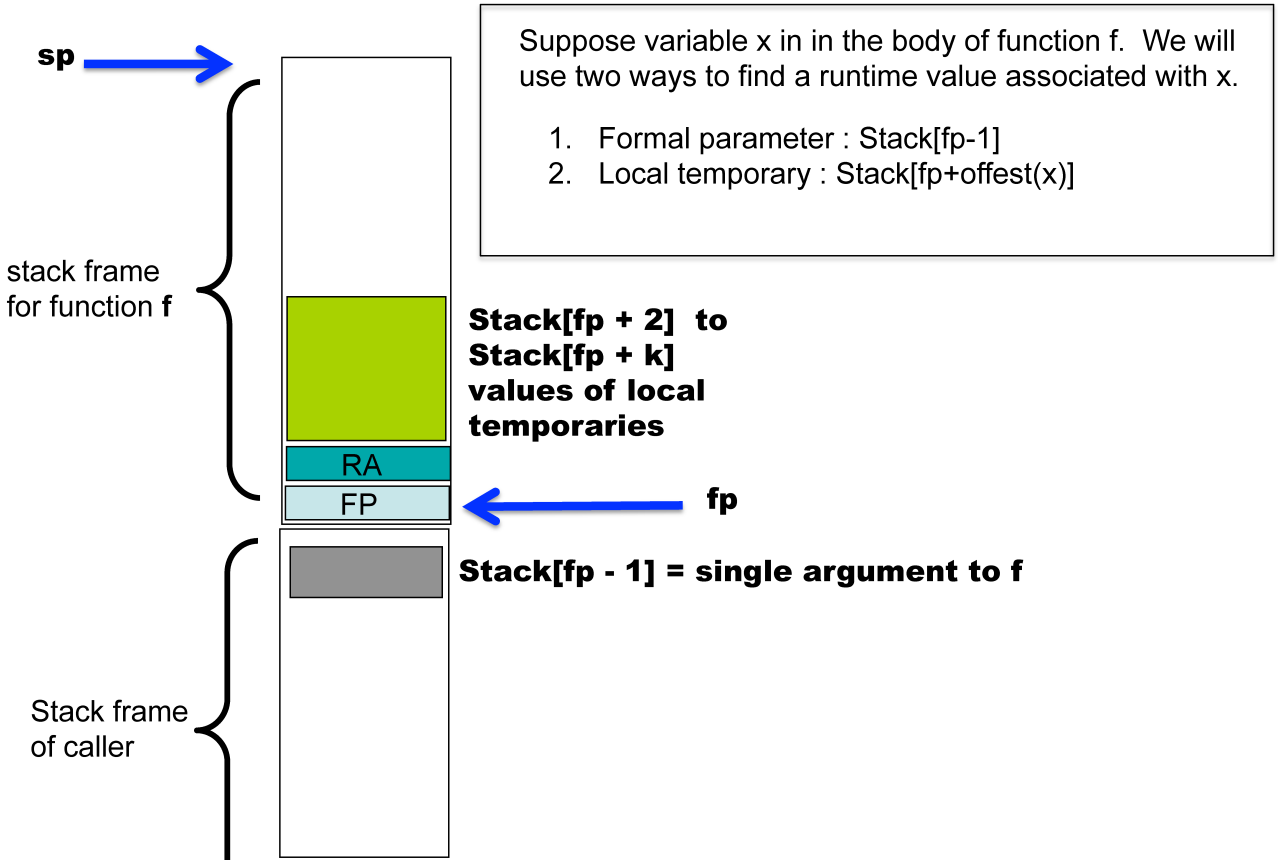
```
fun f(a : int) : int -> int =

   fun g(x :int) : int = a + x
   in g end
in

   let add21 : int -> int  = apply f 21
   in
      let add17 : int -> int  = apply f 17
      in
         (apply add17 3) + (apply add21 -1)
      end
   end
end
```

Note that calls to `f` are “direct” --- there is no need to build a closure on heap since the body of f has no free variables (other than the formal parameter)

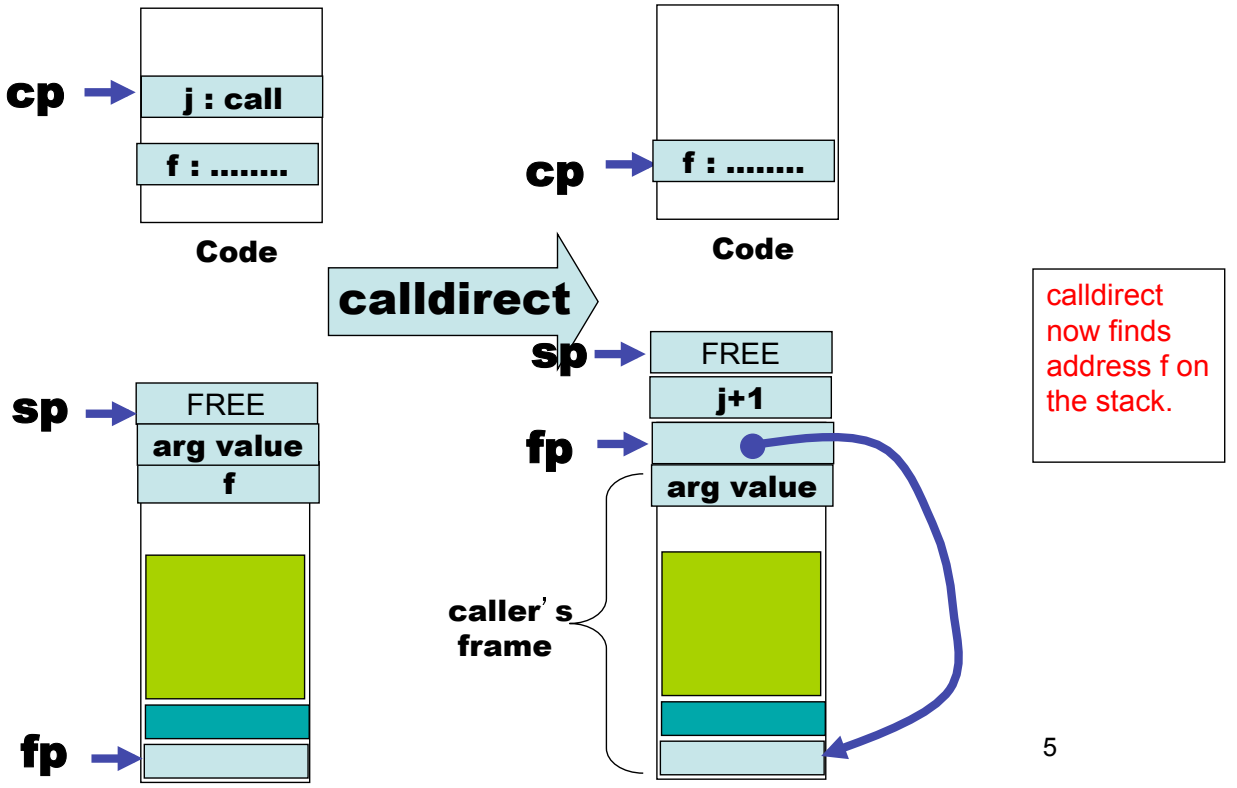
The applications of `add17` and `add21` are different --- they invoke closures stored in the heap

Let us assume that most functions are direct, and that we don’t want to build closures on the heap for such functions. } Not so easy!

Slang.2 VSM Call Stack for a direct function f

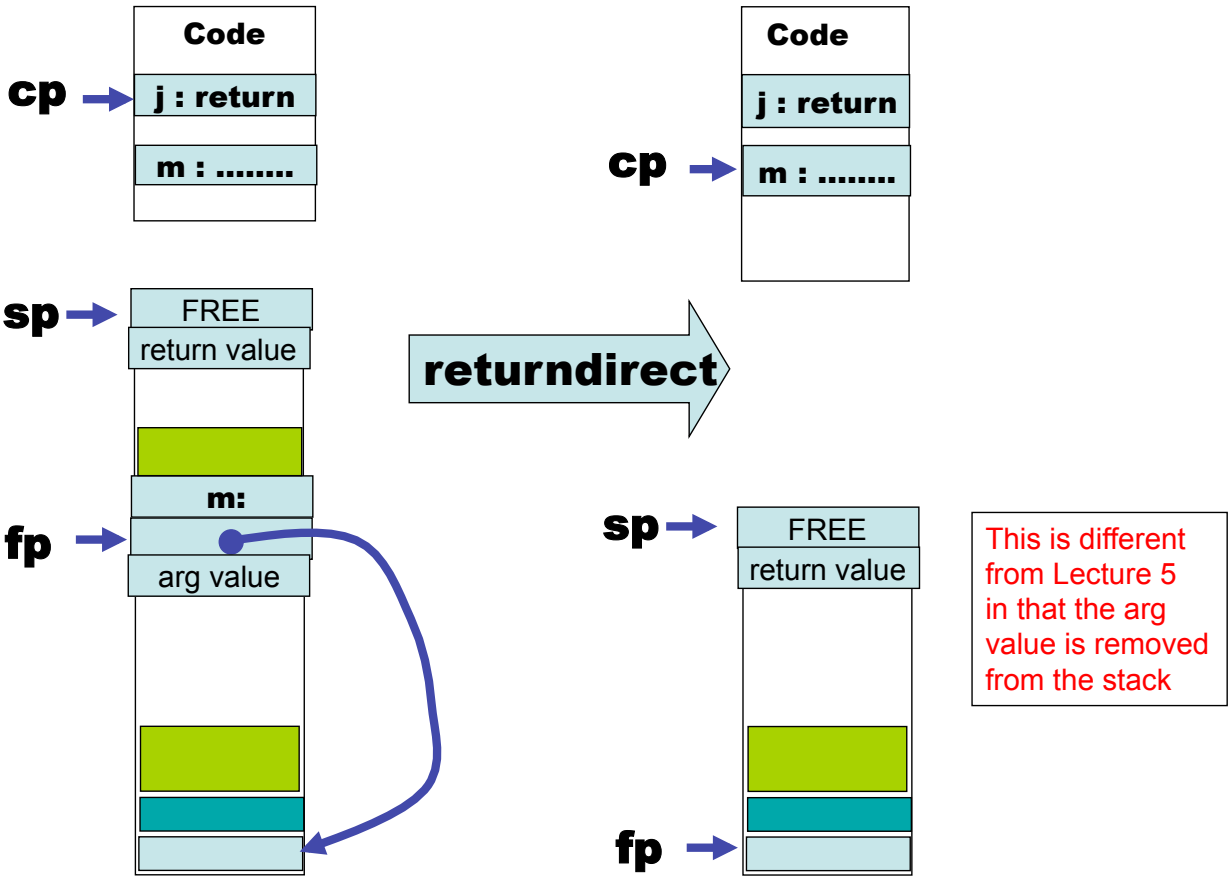


Call (modified from Lecture 5)



calldirect now finds address f on the stack.

returndirect



This is different from Lecture 5 in that the arg value is removed from the stack

Simple function call

calldirect f e

push f	Put address f on stack
code for e	Leave argument value of e on top of stack
calldirect	

New call works well with function-valued expressions

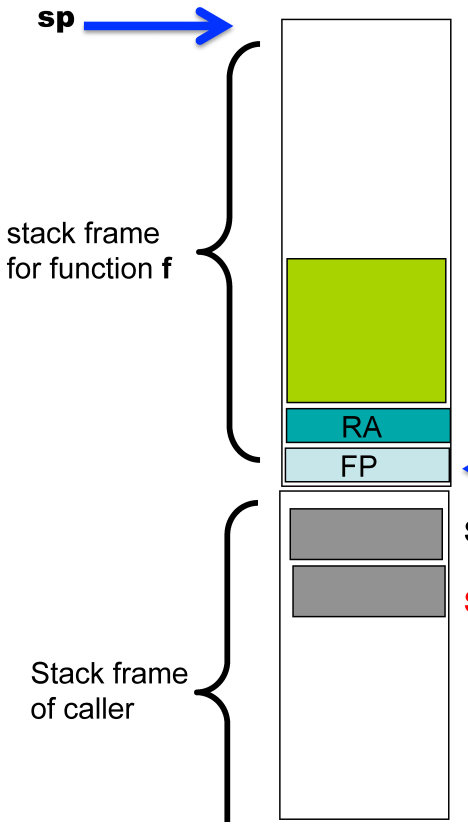
calldirect e1 e2

code for e1	leave an address on stack
code for e2	Leave argument value of e on top of stack
calldirect	

Why is address of function below argument value on stack?

Remember : left-to-right evaluation

Slang.2 VSM Call Stack for closure function f



Suppose variable x in in the body of function f. We will use three ways to find a runtime value associated with x.

1. Formal parameter : Stack[fp-1]
2. Local temporary : Stack[fp+offest(x)]
3. Non-local (in-closure) : Heap[Stack[fp-2] +offset(x)]

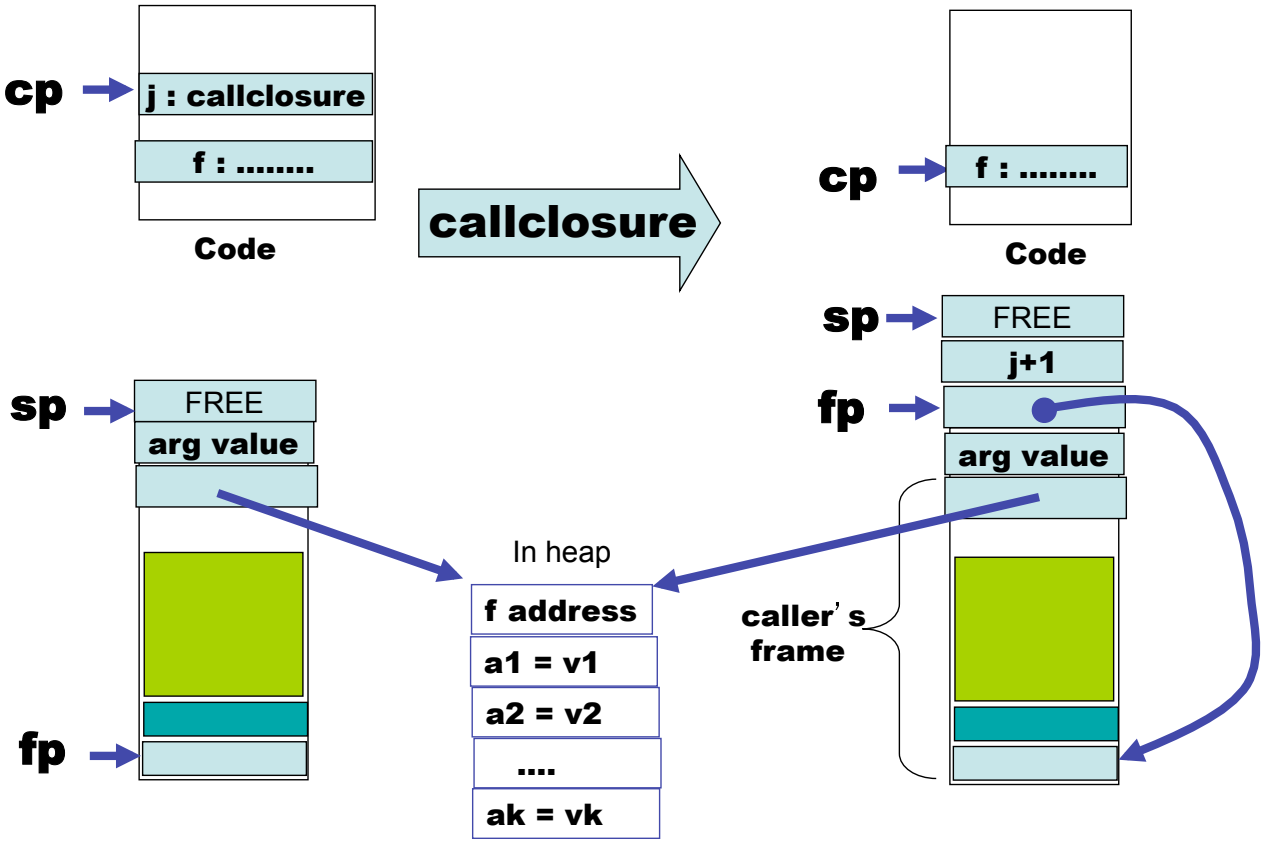
Note : We could devise more elaborate mechanisms. See Appel, Chapter 15, for an example.

Stack[fp + 2] to Stack[fp + k] values of local temporaries

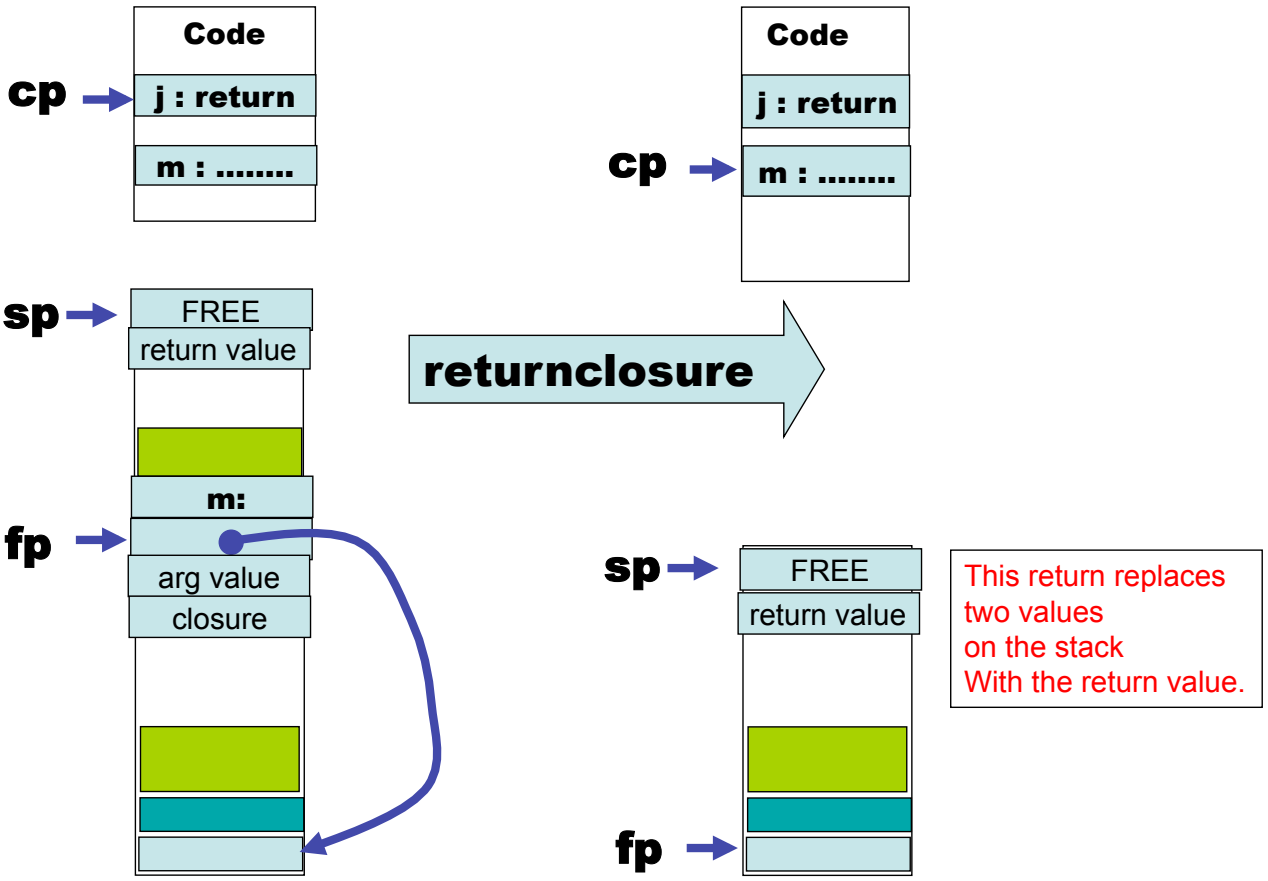
Stack[fp - 1] = single argument to f

Stack[fp - 2] = pointer to closure for instance of f

Callclosure



returndirect



Calling a closure

callclosure e1 e2

code for e1	**Leave a pointer into the heap on stack**
code for e2	**Leave argument value on top of stack**
callclosure	

Problem

How can we compile the following expression?

```
apply e1 e2
```

We do not know until run time if `e1` will need a calldirect or a callclosure. For example, suppose h is a bound to a direct function and f is bound to a closure in the following:

```
apply (if e then h else f) e2
```

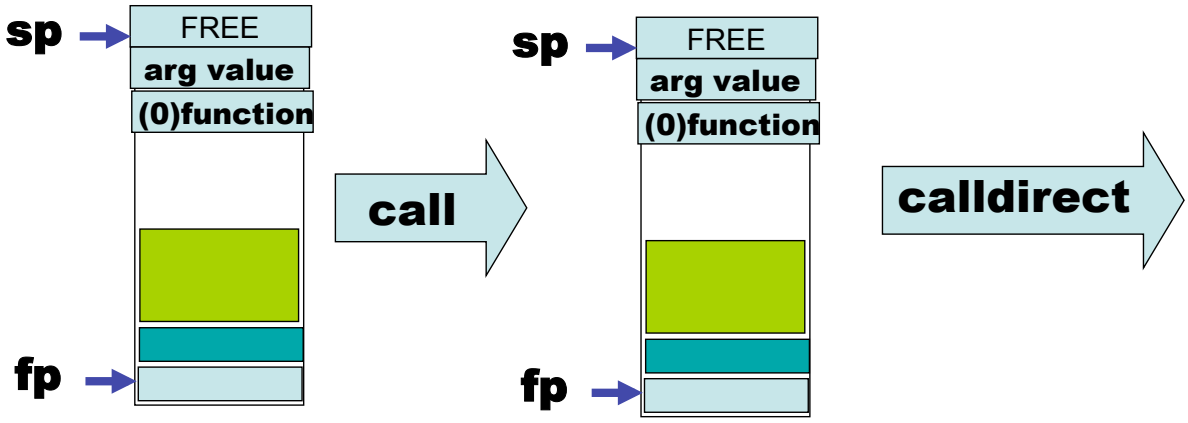
Solution : functional values need to identify themselves at run-time as being direct or closure.
We will use the first bit of the word for a function location: 0 for direct, 1 for closure. Note that this reduces our address space for functions by ½.

call

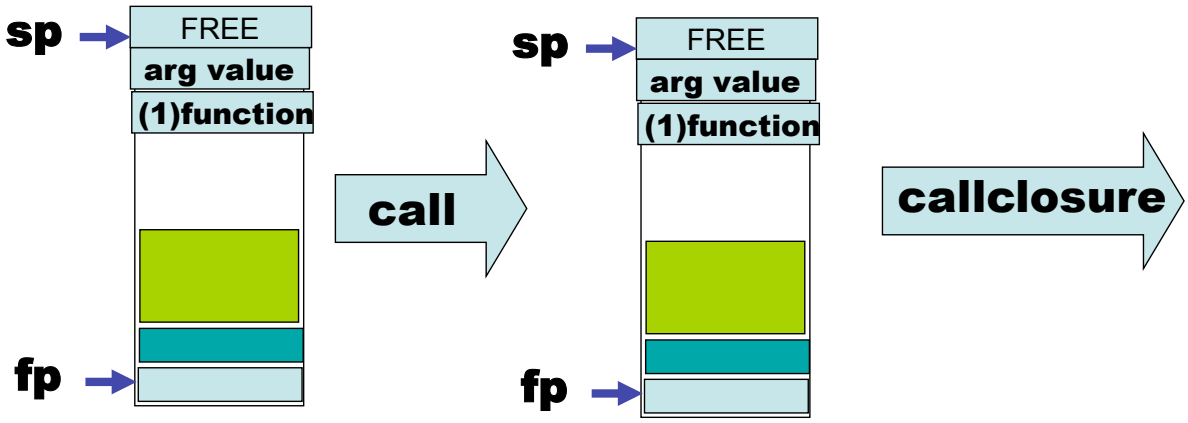
call e1 e2

code for e1	**Leaves functional value on stack: the first bit is either a 0 (for direct) or a 1 (for closure).**
code for e2	**Leave argument value on top of stack**
call	**This now does either calldirect or callclosure depending On the value 0 or 1 associated with function part.**

call (with 0)



call (with 1)



apply e1 e2

If we know at compile-time that f is direct.

If we know at compile-time that f is a closure.

If we don't know much about e1 at compile-time.

We might want to first apply rewriting, such as

What is the "register allocation problem"?

At some point in the back-end, the compiler must confront the fact that the target machine does not have an infinite number of registers.

A solution will

- Assign temporaries to finite number of registers
- Attempt to assign source and target of "move" instructions to same register so that the move can be eliminated

Of course the "live" temporaries at a given point in a program may not fit in the available registers, so the associated values must be "spilled" into memory (into a stack frame, or onto the heap).

Good solutions to this problem require the kind of "dataflow analysis" that is covered in Optimising Compilers (Part II). In the meantime, if you are curious see Appel Chapters 10 and 11.