Compiler Principle

Prof. Dongming LU

Apr. 1st, 2024

Content

6 ACTIVATION RECORD

Function calls behave

- A function may have local variable created upon entry to the function.

```
function f(x:int):
  int =
   let var  y:==
  x+x
     in  if  y<10
         then  f(y)
         else
  return y-1
end
```

- ✓ A new instantiation of x is created (and initialized by f's caller) each time that f is called.
- ✓ There are recursive calls, many of these x's exist simultaneously.

- In many languages (including C, Pascal, and Java), local variables are destroyed when a function returns.

- last-in-first-out (LIFO) fashion， use a LIFO data structure - a stack

HIGHER-ORDER FUNCTIONS

```
fun f(x) =
   let fun g(y) = x+y
      in g
  end
  val h = f(3)
  val j  = f(4)
  val z = h(5)
  val w = j(7)
```

(a) Written in ML

```
int (*)( ) f ( int x)  {
     int g(int y) {return x+y;}
     return g;
}
 int (*h)( ) = f(3);
 int (*j)( ) = f(4);
 int z = h(5);
 int w = j(7);
```

(b) Written in Pseudo-c

- **It is the combination of *nested functions* and *functions returned as results* .**
- **Local variables need lifetimes longer than their enclosing function invocations.**

HIGHER-ORDER FUNCTIONS

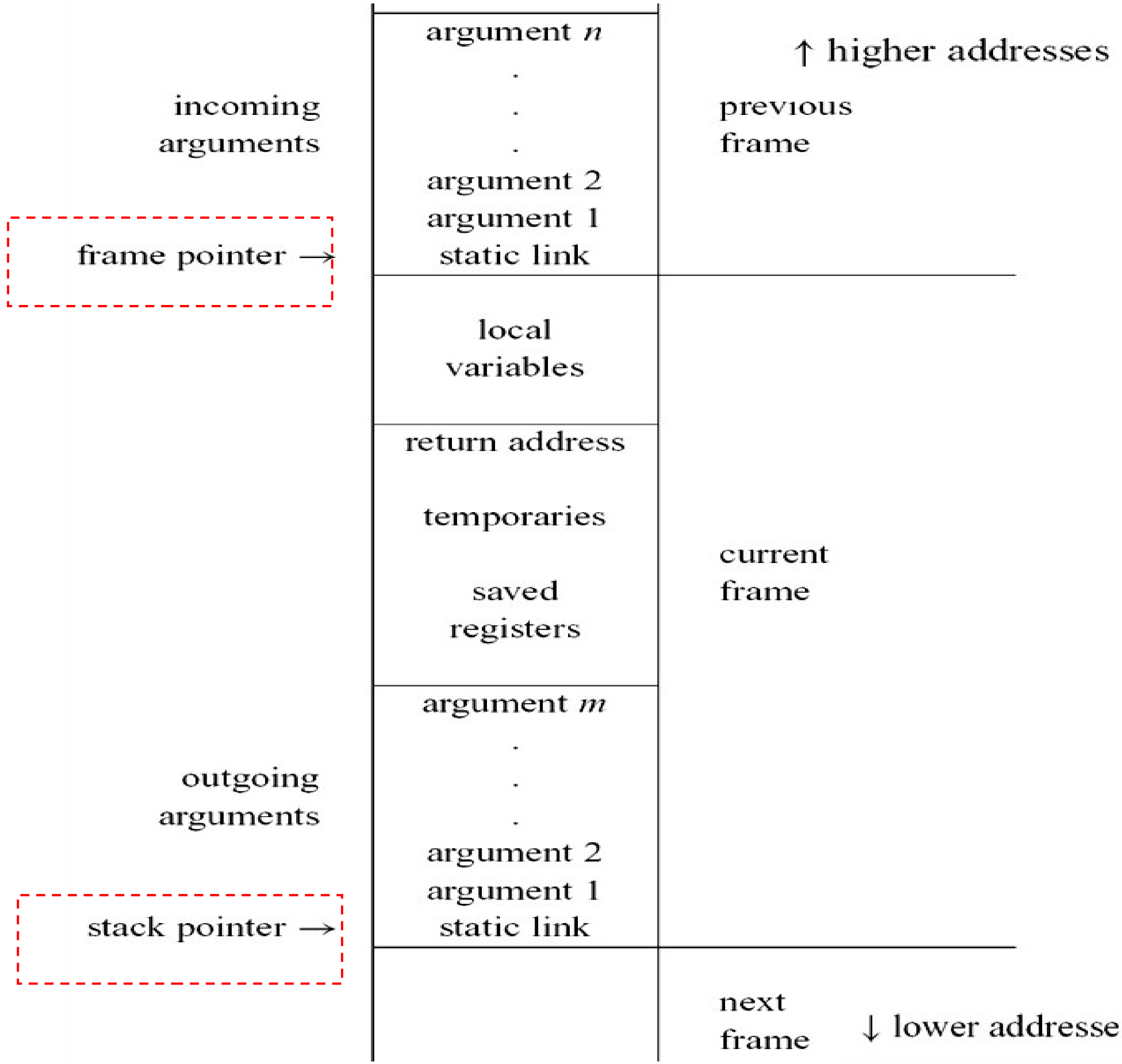
- **Pascal has nested functions, but it does not have functions as returnable values.**
- **C has functions as returnable values, but not nested functions.**
- **Pascal and C can use stacks to hold local variables.**

- **ML, Scheme, and several other languages have both nested functions and functions as returnable values (higher-order functions).**
- **They cannot use stacks to hold all local variables.**

6.1 STACK FRAMES

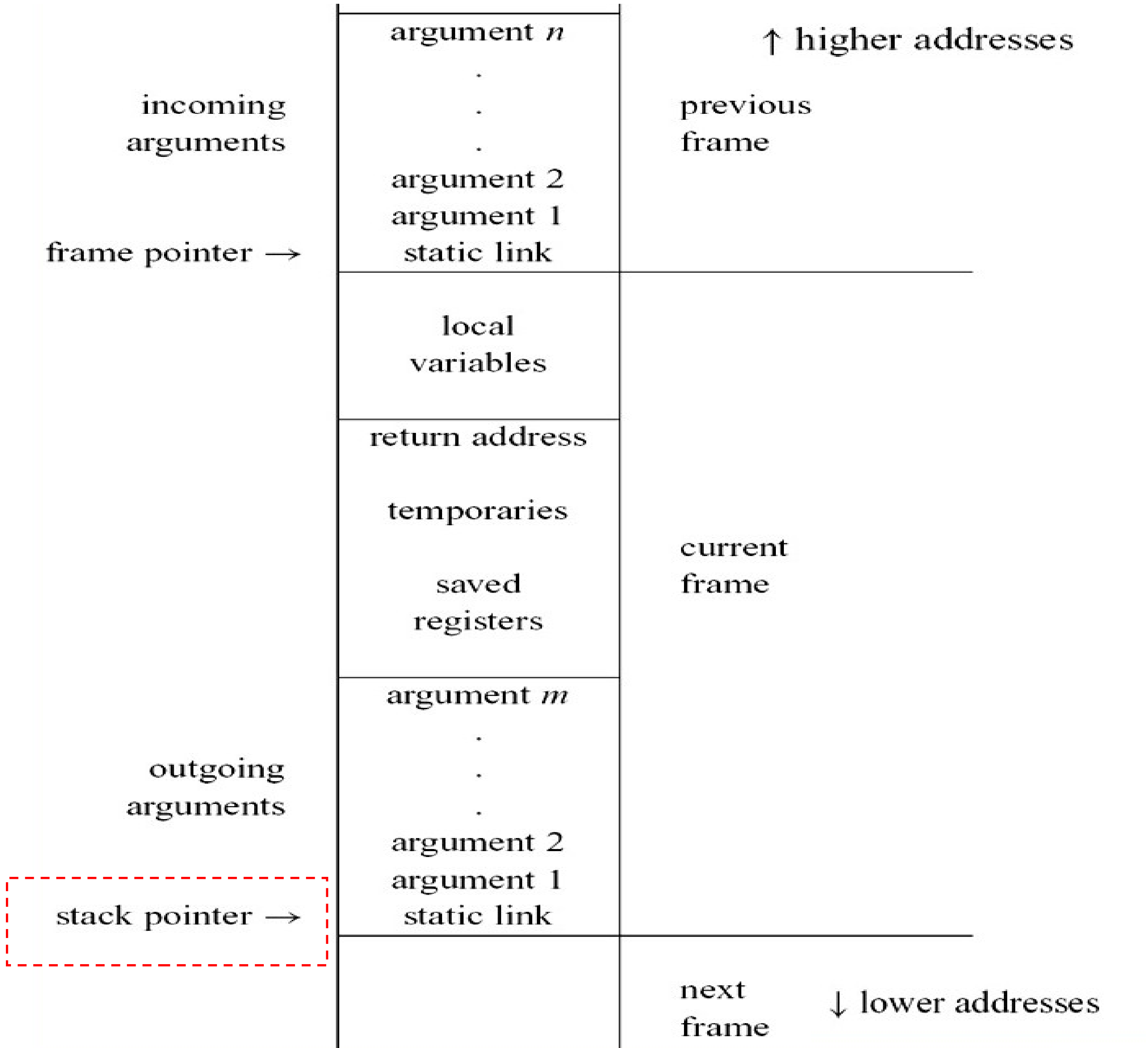
STACK FRAMES

- The area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function.
- Run-time stacks usually start at a high memory address and grow toward smaller addresses.

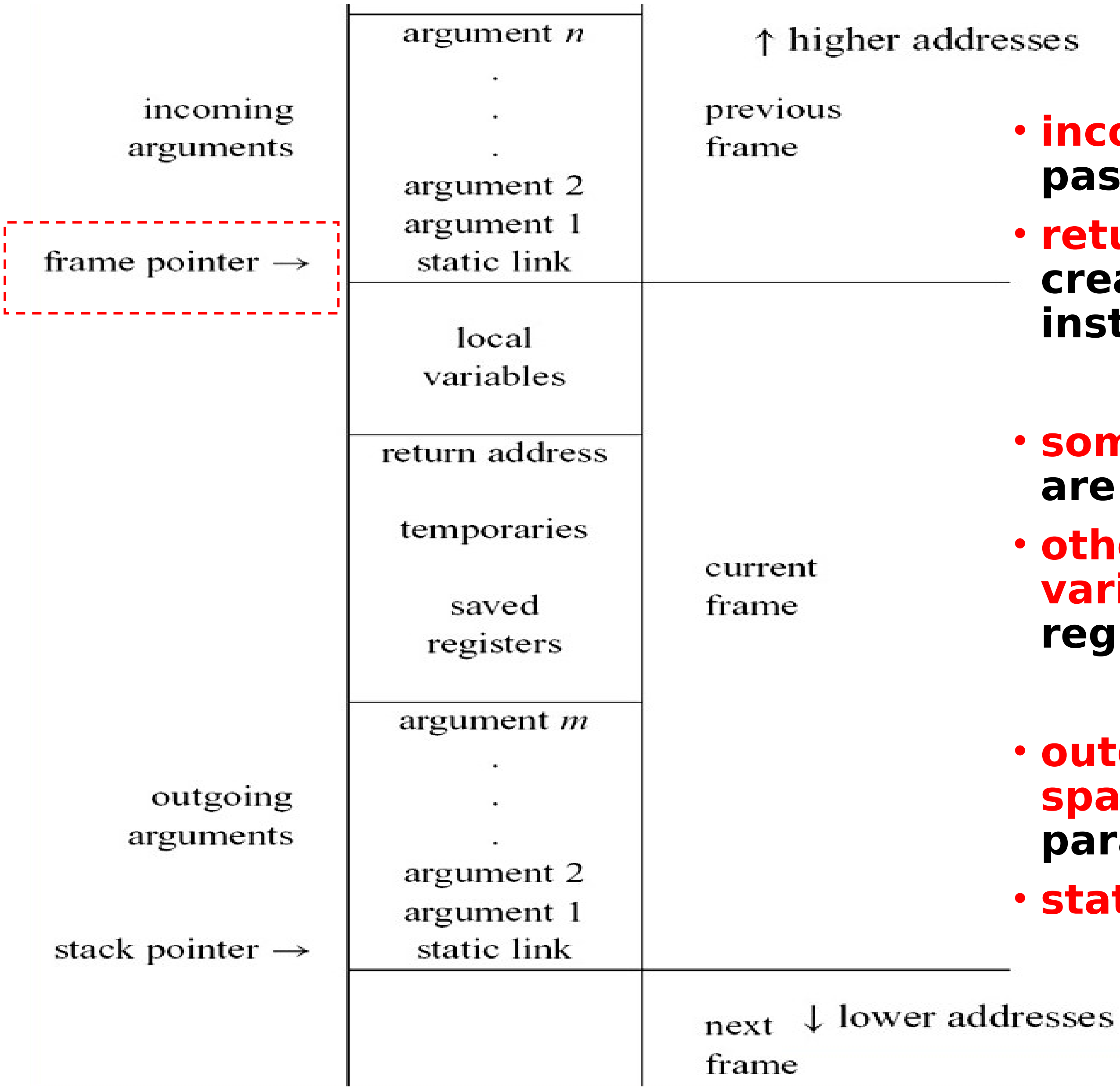


STACK FRAMES

- **The *stack pointer* - that points at some location.**
- **The stack usually grows only at the entry to a function, by an increment large enough to hold all the local variables for that functi**



STACK FRAMES



- **incoming arguments : passed by the caller**
- **return address : created by the CALL instruction**

- **some local variables are in stack frame ,**
- **other temporary local variables are saved in registers**

- **outgoing argument space : pass parameters.**
- **static link**

THE FRAME POINTER

a function *g*(…) calls the function *f*(a_1, …, a_n)

- ***g* is the *caller***
- ***f* is the *callee***

When g calls f

- **The stack pointer SP points to the first argument that g passes to f**
- ***f* allocates a frame by simply subtracting the frame size from the stack pointer SP**

THE FRAME POINTER

When enter f

- **The old SP becomes the current *frame pointer* FP**
- **The old value of FP is saved in memory (in the frame) and the new FP becomes the old SP**

When *f* exits

- **Just copies FP back to SP and fetches back the saved FP.**

REGISTERS

- A modern machine has a large set of ***registers***
- Many different procedures and functions need to use registers

- Suppose: a function ***f*** is using register ***r*** to hold a local variable and calls procedure ***g***, which also uses ***r*** for its own calculations.
  - ✓ Then ***r*** must be saved (stored into a stack frame) before ***g*** uses it
  - ✓ and restored (fetched back from the frame) after ***g*** is finished using it.

REGISTERS

- **When *f*(…) calls the function *g***
  - ✓ ***r*** **is a *caller-save* register if the caller (in this case, *f*) must save and restore the register.**
  - ✓ ***r*** **is *callee-save* if it is the responsibility of the callee (in this case, *g*).**

PARAMETER PASSING

- **Parameter-passing conventions for modern machines specify :**
  - ✓ **The first k arguments (for k = 4 or k = 6, typically) of a function are passed in registers r_p, …, r_{p+k-1},**
  - ✓ **And the rest of the arguments are passed in memory.**

PARAMETER PASSING

Example: **suppose $f(a_1, \ldots, a_n)$ (which received its parameters in $r_1, \ldots, r_n$, for example) calls $h(z)$.**

1. **Some procedures don't call other procedures - these are called *leaf procedures*.**
   - Leaf procedures need not write their incoming arguments to memory.
2. **Some optimizing compilers use *interprocedural register allocation*,**
   - Assign different procedures different registers in which to receive parameters and hold local variables.

PARAMETER PASSING

3. **Parameter *x* is a dead variable at the point where *h* is called. Then *f* can overwrite r_1 without saving it.**

4. **Some architectures have *register windows*, so that each function invocation can allocate a fresh set of registers without memory traffic.**

RETURN ADDRESSES

- **If the *call* instruction within *g* is at address *a*, then (usually)**
  - ✓ **The right place to return to is *a* + 1, the next instruction in *g*.**
  - ✓ **This is called the *return address*.**

- **On modern machines, the *call* instruction merely puts the return address in a designated register.**

- **A nonleaf procedure will have to write it to the stack (unless interprocedural register allocation is used)，a leaf procedure will not.**

FRAME-RESIDENT VARIABLES

The register holding the variable is needed for a specific purpose。

- **function parameters in registers,**
- **pass the return address in a register**
- **return the function result in a register**

FRAME-RESIDENT VARIABLES

The reasons of values are written to memory (in the stack frame) :

- **The variable will be passed by reference, so it must have a memory address .**
- **The variable is accessed by a procedure nested inside the current one.**
- **The value is too big to fit into a single register.**
- **The variable is an array, for which address arithmetic is necessary to extract components.**
- **There are so many local variables and temporary values that they won't all fit in registers, some of them are "spilled" into the frame.**

STATIC LINKS

- **The inner functions may use variables declared in outer functions.**
- **This language feature is called *block structure.***

STATIC LINKS

- Whenever a function f is called, it can be passed a pointer to the frame of the function statically enclosing f
  - ✓This pointer is the static link.
- A global array can be maintained .
  - ✓This array is called a display.
- When g calls f, each variable of g that is actually accessed by f (or by any function nested inside f) is passed to f as an extra argument.
  - ✓This is called lambda lifting.

STATIC LINKS

```
type tree = {key: string, left: tree, right: tree}

function prettyprint(tree: tree) : string =
 let
    var output := " "

   function write(s: string) =
      output := concat(output,s)

   function show(n:int, t: tree) =
     let function indent(s: string) =
        (for i := 1 to n
            do write(" ");
           output := concat(output, s); write("\n"))
    in if t=nil then indent(".")
           else (indent(t.key);
                 show(n+1,t.left);
                 show(n+1,t.right))
      end

     in show(0,tree); output
end
```

STATIC LINKS

21 prettyprint calls show, passing prettyprint's own frame pointer as show's static link.

10 show stores its static link (the address of prettyprint's frame) into its own frame.

15 show calls indent, passing its own frame pointer as indent's static link.

17 show calls show, passing its own static link (not its own frame pointer) as the static link.

STATIC LINKS

12 indent uses the value *n* from show's frame. To do so, it fetches at an appropriate offset from indent's static link (which points at the frame of show).

13 indent calls write. It must pass the frame pointer of prettyprint as the static link. To obtain this, it first fetches at an offset from its own static link (from show's frame), the static link that had been passed to show.

14 indent uses the variable output from prettyprint’s frame.   To do so it starts with its own static link, then fetches show's, then fetches output.[

The end of Chapter 6(1)