Activation Records/Stack Frames

COS 320

Compiler Implementation

Princeton University
Spring 2006

Prof. David August (guest)

Activation Records (ARs)

- Modern imperative PLs typically have local variables
  - Created upon call to function (or entry to region of code)
  - Destroyed upon return of function (or exit of region of code)
- Each invocation has own instance of locals
  - Recursive calls require several instances to exist simultaneously
  - Function instance dies only after all callees have died (LIFO)
  - Need LIFO structure to hold each instance: Stack
- The portion of “The Stack” used for an invocation of a function is called the stack frame or activation record
- Callee/Caller terminology?

The Stack

- Essentially:
  - A large (resizable) array
  - Grows downward (upward) in memory addresses
  - Shrinks upward (downward)
- push(r1):
  - stack_pointer--;
  - M[stack_pointer] = r1;
- r1 = pop():
  - r1 = M[stack_pointer];
  - stack_pointer++;
- Notes:
  - Push and pop entire activation records?
  - Previous activation records need to be accessed? Implications?
# Stack Frame Example

```plaintext
let
  function g(x:int) =
    let
      var y := 10
    in
      x + y
    end
  function h(y:int):int =
    y + g(y)
  in
    h(4)
end
```

# Recursive Example

```plaintext
let
  function fact(n:int):int =
    if n = 0 then 1
    else n * fact(n - 1)
  in
    fact(3)
end
```
What about Functional Languages?

Some functional PLs (ML, Scheme) cannot use a stack

```plaintext
fun f(x) =
  let
    fun g(y) = x + y
  in
    g
  end

Consider:
- val z = f(4)
- val w = z(5)
```

Assume variables are stack-allocated.

---

Functional Languages

Combination of nested functions and nested returned results (higher-order functions):

1. Requires locals to remain after enclosing function returns
2. Activation records must be allocated on heap, not stack

Concentrate on languages using the stack...

Prof. Walker adds:

Comment that I already talked about closure conversion, which deals with the problem of creating "activation records" (closures) for ML-style nested functions (or at least reduces it to the problem of creating activation records for C).

---

Stack Frame Organization

- In isolation, compiler can use any layout scheme
- Microprocessor manufacturers specify standards
  - Called: Calling Conventions
  - Allows code from different compilers to work together
  - Essential for library interaction
**Typical Calling Convention**

- *Frame Pointer* points to top (bottom) of previous frame
- *Stack Pointer* points to slot above (below) current frame

**Stack Frame Example**

Suppose \( f(a_1, a_2) \) calls \( g(b_1, b_2, b_3) \)

**Step 1:**

- Frame Pointer (FP) \( \Rightarrow \)
- Stack Pointer (SP) \( \Rightarrow \)
- Previous Frame
- Frame for \( f \)
- Garbage

**Step 2:**

- Frame Pointer (FP) \( \Rightarrow \)
- Stack Pointer (SP) \( \Rightarrow \)
- Previous Frame
- Frame for \( f \)
- Garbage

**Step 3:**

Dynamic Link (AKA Control Link) points to AR of the caller
- Optional if size of caller AR is static and known
- Used to restore stack pointer during return sequence
Parameter Passing

\[ f(a_1, a_2, \ldots, a_n) \]

- Registers are faster than memory.
- Compiler should keep values in register whenever possible.
- Modern calling convention: rather than placing \( a_1, a_2, \ldots, a_n \) on stack frame, put \( a_1, \ldots, a_k \) (\( k = 4 \)) in registers \( r_0, r_{p+1}, r_{p+2}, r_{p+3} \) and \( a_{k+1}, a_{k+2}, a_{k+3}, \ldots, a_n \).
- If \( r_0, r_{p+1}, r_{p+2}, r_{p+3} \) are needed for other purposes, callee function must save incoming argument(s) in stack frame.
- C language allows programmer to take address of formal parameter and guarantees that formals are located at consecutive memory addresses.
  - If address argument has address taken, then it must be written into stack frame.
  - Saving it in “saved registers” area of stack won’t make it consecutive with memory resident arguments.
  - Space must be allocated even if parameters are passed through register.

Parameter Passing

If register argument has address taken,

\( \text{callee materializes it on the stack} \)

Frame Pointer (FP) –

\[ a(0) \]
\[ a(k-1) \]
space for \( a(k) \)

Stack Pointer (SP) –

space for \( a(2) \)

space for \( a(1) \)

Garbage

Registers

- Compilers typically place a variable on stack until it can determine whether or not it can be promoted to a register (e.g. no references)

- The assignment of variables to registers is done by the Register Allocator

Registers

- Some Parameters
- Return Value
- Local Variables
- Intermediate results of expressions (temperaries)

Stack Frame holds:

- Variables passed by reference or have their address taken (&)
- Variables that are accessed by procedures nested within current one.
- Variables that are too large to fit into register file
- Array variables (address arithmetic needed to access array elements).
- Variables whose registers are needed for a specific purpose (parameter passing)
- Spilled registers. Too many local variables to fit into register file, so some must be stored in stack frame.
### Registers

Register's value must be saved before callee can reuse

Calling convention defines two types of registers:

- **Caller-save registers** are responsibility of the caller
  - Caller-save register values saved only if used after call/return
  - The callee function can use caller-saved registers with concern

- **Callee-save register** are the responsibility of the callee
  - Values must be saved by callee before they can be used
  - Caller can assume that these registers will be restored

Allocation of variables to callee-saved vs. caller-saved done by register allocator

### Return Address and Return Value(s)

**Return Address**

- A called function must be able to return to caller
- Return address is address of instruction following call
- Return address can be placed on the stack or register
- A call instruction (if present in ISA) places return address in a designated register
- The return address is written to stack by callee in non-leaf functions

**Return Value** is placed in designated register or on stack

### Frame Resident Variables

- A variable *escapes* if:
  - it is passed by reference,
  - its address is taken, or
  - it is accessed from a nested function

- Variables cannot be assigned a location at declaration time
  - Escape conditions not known
  - Assign provisional locations, decide later if variables can be promoted to registers

- *escape* set to true by default

### Static Links

In languages that allow nested functions, functions must access other function’s stack frame.

```javascript
let
function f():int = let
  var a = 5
  function g(y:int):int = let
    var b = 10
    function h(z:int):int =
      if z > 10 then h(z / 2)
      else z + b * a <- b, a of outer fn
    in
      y + a + h(16) <- a of outer fn
    end
  in
    g(10)
  end
in f() end
```
Whenever f is called, it is passed a pointer to most recent AR of g that immediately encloses f in program text → Static Link (AKA Access Link)

- Need a chain of indirect memory references for each variable access
  - Example: M[M[F]]
- Number of indirect references = difference in nesting depth between variable declaration function and use function