Scope and Activation Records

Vitaly Shmatikov
Activation Records for Functions

- Block of information ("frame") associated with each function call, including:
  - Parameters
  - Local variables
  - Return address
  - Location to put return value when function exits
  - Control link to the caller’s activation record
  - Saved registers
  - Temporary variables and intermediate results
  - (not always) Access link to the function’s static parent
Activation Record Layout

- **Return address**
  - Location of code to execute on function return

- **Return-result address**
  - Address in activation record of calling block to receive returned value

- **Parameters**
  - Locations to contain data from calling block
Example

Function

\[ \text{fact}(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n \times \text{fact}(n-1) & \text{else} \end{cases} \]

- Return result address: location to put fact(n)

Parameter

- Set to value of \(n\) by calling sequence

Intermediate result

- Locations to contain value of fact(n-1)
Typical x86 Activation Record

Caller’s activation frame

old sp

Callee’s activation frame

higher memory addresses

caller saves

saved frame pointer

saved registers

caller’s return address

fp + offset

frame pointer

fp - offset

local variables

sp

stack pointer

Next activation frame to be called

temporary storage

lower memory addresses
Run-Time Stack

◦ Activation records are kept on the stack
  • Each new call pushes an activation record
  • Each completing call pops the topmost one
  • Stack has all records of all active calls at any moment during execution (topmost record = most recent call)

◦ Example: fact(3)
  • Pushes one activation record on the stack, calls fact(2)
  • This call pushes another record, calls fact(1)
  • This call pushes another record, resulting in three activation records on the stack
Function Call

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n \times \text{fact}(n-1) & \text{else}
\end{cases}
\]

Return address omitted; would be a pointer into code segment
Function Return

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n \times \text{fact}(n-1) & \text{else}
\end{cases}
\]
Scoping Rules

- **Global and local variables**
  - \(x, y\) are local to outer block
  - \(z\) is local to inner block
  - \(x, y\) are global to inner block

- **Static scope**
  - Global refers to declaration in closest enclosing block

- **Dynamic scope**
  - Global refers to most recent activation record

- **Do you see the difference?** (think function calls)
Static vs. Dynamic Scope

Example

```javascript
var x = 1;
function g(z) { return x + z; }
function f(y) {
    var x = y + 1;
    return g(y * x);
}

f(3);
```

Which x is used for expression \(x + z\)?

- static scope
- dynamic scope
Activation Record For Static Scope

- **Control link**
  - Link to activation record of previous (calling) block
  - Depends on dynamic behavior of the program

- **Access link**
  - Link to activation record of closest lexically enclosing block in program text
  - Is this needed in C? (why?)
  - Depends on the static program text
Static Scope with Access Links

var x = 1;
function g(z) = { return x+z; }
function f(y) =
{ var x = y+1;
  return g(y*x); }
f(3);

Outer block

Use access link to find global variable:
- Access link is always set to frame of closest enclosing lexical block
- For function body, this is the block that contains function definition
Variable Arguments (Redux)

Special functions `va_start`, `va_arg`, `va_end` compute arguments at run-time (how?)

```c
void printf(const char* format, ...) {
    int i; char c; char* s; double d;
    va_list ap; /* declare an “argument pointer” to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != \'\0\'; p++) {
        if (*p == \'%\') {
            switch (++p) {
                case \'d\':
                    i = va_arg(ap, int); break;
                case \'s\':
                    s = va_arg(ap, char*); break;
                case \'c\':
                    c = va_arg(ap, char); break;
            }
        ... /* etc. for each % specification */
    }
    ...

    va_end(ap); /* restore any special stack manipulations */
}
```
Activation Record for Variable Args

- **va_arg(ap,type)** retrieves next arg from offset ap
- **va_start** computes location on the stack past last statically known argument
- **caller** saves
- **callee** saves

**Diagram:***
- Caller's activation frame
  - old sp
  - callee's activation frame
  - va_arg(ap,type) retrieves next arg from offset ap
  - caller's return address
  - saved frame pointer
  - saved registers
  - local variables
  - temporary storage
  - higher memory addresses
  - caller saves
  - fp + offset
  - va_start computes location on the stack past last statically known argument
  - fp - offset
  - sp
  - lower memory addresses
  - Next activation frame to be called
Tail Recursion (first-order case)

- Function g makes a tail call to function f if return value of function f is return value of g
- Example
  
  ```
  fun g(x) = if x>0 then f(x) else f(x)*2
  ```

- Optimization: can pop current activation record on a tail call
  - Especially useful for recursive tail call because next activation record has exactly same form
Example of Tail Recursion

fun f(x, y) = if x > y
  then x
  else f(2*x, y);

f(1, 3) + 7;

Optimization

• Set return value address to that of caller
• Can we do same with control link?
• Avoid return to caller
• Does this work with dynamic scope?

Calculate least power of 2 greater than y
Tail Recursion Elimination

Optimization

- pop followed by push - reuse activation record in place
- Tail recursive function is equivalent to iterative loop

fun f(x, y) = if x>y
  then x
  else f(2*x, y);

f(1,3) + 7;
Tail Recursion and Iteration

fun f(x, y) = if x > y
  then x
  else f(2*x, y);
f(1, y);

function g(y) {
  var x = 1;
  while (!x > y)
    x = 2*x;
  return x;
}
Higher-Order Functions

◆ Function passed as argument
  • Need pointer to activation record “higher up” in stack

◆ Function returned as the result of function call
  • Need to keep activation record of the returning function (why?)

◆ Functions that take function(s) as input and return functions as output are known as functionals
Pass Function as Argument

There are two declarations of \( x \)
Which one is used for each occurrence of \( x \)?
Static Scope for Function Argument

val x = 4;
fun f(y) = x*y;
fun g(h) =
    let
      val x = 7
    in
      h(3) + x;
    g(f);

How is access link for h(3) set?
Closures

- Function value is pair \texttt{closure} = \langle \text{env}, \text{code} \rangle
  - Idea: statically scoped function must carry a link to its static environment with it
  - Only needed if function is defined in a nested block (why?)

- When a function represented by a closure is called...
  - Allocate activation record for call (as always)
  - Set the access link in the activation record using the environment pointer from the closure
Function Argument and Closures

val x = 4;
fun f(y) = x*y;
fun g(h) =
  let
    val x = 7
  in
    h(3) + x;
g(f);

Run-time stack with access links
Summary: Function Arguments

- Use closure to maintain a pointer to the static environment of a function body.
- When called, set access link from closure.
- All access links point “up” in stack.
  - May jump past activation records to find global vars.
  - Still deallocate activation records using stack (last-in-first-out) order.
Return Function as Result

Language feature (e.g., ML)

Functions that return “new” functions

- Example: `fun compose(f,g) = (fn x => g(f x));`
- Function is “created” dynamically
  - Expression with free variables; values determined at run-time
- Function value is closure = \langle env, code \rangle
- Code not compiled dynamically (in most languages)
- Need to maintain environment of the creating function (why?)
Return Function with Private State

fun mk_counter (init : int) =
    let  val count = ref init
        fun counter(inc:int) =
            (count := !count + inc; !count)
        in
            counter
        end;
val c = mk_counter(1);
c(2) + c(2);

• Function to “make counter” returns a closure
• How is correct value of count determined in c(2) ?
Function Results and Closures

fun mk_counter (init : int) = 
  let  val count = ref init 
    fun counter(inc:int) = 
      (count := !count + inc; !count) 
  in 
    counter 
  end;

val c = mk_counter(1);

Call changes cell value from 1 to 3

c(2) + c(2);
Closures in Web Programming

◆ Useful for event handlers

```javascript
function AppendButton(container, name, message) {
    var btn = document.createElement('button');
    btn.innerHTML = name;
    btn.innerHTML = name;
    btn.innerHTML = name;
    btn.onclick = function(evt) { alert(message); } well { alert(message); }
    container.appendChild(btn);
}
```

◆ Environment pointer lets the button’s click handler find the message to display
Managing Closures

- Closures as used to maintain static environment of functions as they are passed around
- May need to keep activation records after function returns *(why?)*
  - Stack (last-in-first-out) order fails! *(why?)*
- Possible “stack” implementation:
  - Put activation records on heap
  - Instead of explicit deallocation, invoke garbage collector as needed
    - Not as totally crazy as it sounds *(may only need to search reachable data)*