Scope, Function Calls and Storage Management

John Mitchell
Topics

◆ Block-structured languages and stack storage
◆ In-line Blocks
  • activation records
  • storage for local, global variables
◆ First-order functions
  • parameter passing
  • tail recursion and iteration
◆ Higher-order functions
  • deviations from stack discipline
  • language expressiveness => implementation complexity
Nested blocks, local variables

Example

```java
{ int x = 2;
{ int y = 3;
  x = y+2;
}
}
```

New variables declared in nested blocks

Storage management
- Enter block: allocate space for variables
- Exits block: some or all space may be deallocated
Examples

◆ Blocks in common languages
  • C, JavaScript: `{ ... }`
  • Algol: `begin ... end`
  • ML: `let ... in ... end`

◆ Two forms of blocks
  • In-line blocks
  • Blocks associated with functions or procedures

◆ Topic: block-based memory management, access to local variables, parameters, global variables

* JavaScript functions provide blocks
Simplified Machine Model

Registers

Code

Data

Program Counter

Environment Pointer

Stack

Heap
Interested in Memory Mgmt Only

- **Registers, Code segment, Program counter**
  - Ignore registers
  - Details of instruction set will not matter

- **Data Segment**
  - Stack contains data related to block entry/exit
  - Heap contains data of varying lifetime
  - Environment pointer points to current stack position
    - Block entry: add new activation record to stack
    - Block exit: remove most recent activation record
Some basic concepts

◆ Scope
  • Region of program text where declaration is visible

◆ Lifetime
  • Period of time when location is allocated to program

```c
{ int x = ... ;
  { int y = ... ;
    { int x = ... ;
      ....
    };
  };
};
```

• Inner declaration of x hides outer one.
• Called “hole in scope”
• Lifetime of outer x includes time when inner block is executed
• Lifetime ≠ scope
• Lines indicate “contour model” of scope.
In-line Blocks

◆ Activation record
  • Data structure stored on run-time stack
  • Contains space for local variables

◆ Example

```c
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
    }
};
```

Push record with space for x, y
Set values of x, y
Push record for inner block
Set value of z
Pop record for inner block
Pop record for outer block

May need space for variables and intermediate results like (x+y), (x-y)
Activation record for in-line block

- **Control link**
  - pointer to previous record on stack

- **Push record on stack:**
  - Set new control link to point to old env ptr
  - Set env ptr to new record

- **Pop record off stack**
  - Follow control link of current record to reset environment pointer

Can be optimized away, but assume not for purpose of discussion.
Example

{ int x=0;
    int y=x+1;
    { int z=(x+y)*(x-y);
    };
};

Push record with space for x, y
Set values of x, y
    Push record for inner block
    Set value of z
    Pop record for inner block
Pop record for outer block

Control link
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>

Control link
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>-1</td>
</tr>
<tr>
<td>x+y</td>
<td>1</td>
</tr>
<tr>
<td>x-y</td>
<td>-1</td>
</tr>
</tbody>
</table>

Environment
Pointer
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

  ```
  { int x=0;
    int y=x+1;
    { int z=(x+y)*(x-y);
    }
  }
  ```

◆ Static scope
  - global refers to declaration in closest enclosing block

◆ Dynamic scope
  - global refers to most recent activation record

These are same until we consider function calls.
Functions and procedures

- Syntax of procedures (Algol) and functions (C)
  ```plaintext
  procedure P (<pars>)            <type> function f(<pars>)
  begin                                {
    <local vars>                          <local vars>
    <proc body>                         <function body>
  end;                                  }
  ```

- Activation record must include space for:
  - parameters
  - return address
  - local variables, intermediate results
  - return value (an intermediate result)
  - location to put return value on function exit
Activation record for function

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

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

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

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

  - 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)
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 ptr into code segment
Function return

fact(n) = if n <= 1 then 1 else n * fact(n-1)
Topics for first-order functions

◆ Parameter passing
  • pass-by-value: copy value to new activation record
  • pass-by-reference: copy ptr to new activation record

◆ Access to global variables
  • global variables are contained in an activation record
  higher “up” the stack

◆ Tail recursion
  • an optimization for certain recursive functions

See this yourself: write factorial and run under debugger
Parameter passing

◆ General terminology: L-values and R-values
  • Assignment \( y := x + 3 \)
    – Identifier on left refers to location, called its L-value
    – Identifier on right refers to contents, called R-value

◆ Pass-by-reference
  • Place L-value (address) in activation record
  • Function can assign to variable that is passed

◆ Pass-by-value
  • Place R-value (contents) in activation record
  • Function cannot change value of caller’s variable
  • Reduces aliasing (alias: two names refer to same loc)
Example

pseudo-code

function f (x) =
    { x = x+1; return x;  }
var y = 0;
print (f(y)+y);
Access to global variables

Two possible scoping conventions

- Static scope: refer to closest enclosing block
- Dynamic scope: most recent activation record on stack

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?
Activation record for static scope

- **Control link**
  - Link to activation record of previous (calling) block

- **Access link**
  - Link to activation record of closest enclosing block in program text

- **Difference**
  - Control link depends on dynamic behavior of prog
  - Access link depends on static form of program text
Static scope with access links

Use access link to find global variable:
- Access link is always set to frame of closest enclosing lexical block
- For function body, this is block that contains function declaration

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

\[
\text{fun } g(x) = \text{if } x > 0 \text{ then } f(x) \text{ else } f(x) \times 2
\]

◆ Optimization
  • Can pop activation record on a tail call
  • Especially useful for recursive tail call
    – next activation record has exactly same form
Example

Calculate least power of 2 greater than $y$

fun $f(x, y) =$ if $x > y$
then $x$
else $f(2 \times x, y)$;

$f(1, 3) + 7$;

Optimization
- Set return value address to that of caller

Question
- Can we do the same with control link?

Optimization
- avoid return to caller
Tail recursion elimination

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

f(1,3);  

Optimization
- pop followed by push = reuse activation record in place

Conclusion
- Tail recursive function equiv to iterative loop
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;
}

initial value

control

return val

x | 1
y | 3

control

return val

x | 2
y | 3

control

return val

x | 4
y | 3

loop body

test
Higher-Order Functions

◆ Language features
  • Functions passed as arguments
  • Functions that return functions from nested blocks
  • Need to maintain environment of function

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

◆ More complicated second case
  • Function returned as result of function call
  • Need to keep activation record of returning function
Pass function as argument

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

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

How is access link for h(3) set?
Static Scope for Function Argument

```javascript
{ var x = 4;
  { function f(y) {return x*y};
    { function g(h) {
        int x=7;
        return h(3) + x;
      }
    }
  }
  g(f);
}
```

How is access link for h(3) set?
Result of function call

```javascript
var x = 4;

function f(y) { return x*y; }

function g(h) {
    var x = 7;
    return h(3) + x;
}

g(f);
```

19

```javascript
js>
```
Closures

- Function value is pair $\text{closure} = \langle \text{env}, \text{code} \rangle$
- 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

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

Run-time stack with access links

- `x`: 4
- `access f`
- `access g`
- `access h`
- `access x`: 7
- `access y`: 3

Code for `f`

Code for `g`

access link set from closure
Function Argument and Closures

```javascript
{ var x = 4;
  { function f(y) { return x * y; }
    { function g(h) {
      int x = 7;
      return h(3) + x;
    }
    g(f);
  }
}}
```

Run-time stack with access links

- `x = 4` accessed from `f`
- `h(3)` accessed from `g`
- `x = 7` accessed from `g`
- `y = 3` accessed from `g`
- `access link set from closure`
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 activ records to find global vars
  - Still deallocate activ records using stack (lifo) order
Return Function as Result

◆ Language feature
  • Functions that return “new” functions
  • Need to maintain environment of function

◆ Example
  ```javascript
  function compose(f, g)
  {
    return function(x) { return g(f(x)) };
  }
  ```

◆ Function “created” dynamically
  • expression with free variables
    values are determined at run time
  • function value is closure = ⟨env, code⟩
  • code not compiled dynamically (in most languages)
Example: Return fctn with private state

```ml
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)?
Example: Return fctn with private state

```javascript
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return count};
    return counter;
}
var c  = mk_counter(1);
c(2) + c(2);

Function to “make counter” returns a closure
How is correct value of count determined in call 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 
end;
val c = mk_counter(1);
c(2) + c(2);

Call changes cell value from 1 to 3
function mk_counter (init) {
    var count = init;
    function counter(inc) {count=count+inc; return count};
    return counter;
}
var c  = mk_counter(1);
c(2) + c(2);
Closures in Web programming

💡 Useful for event handlers in Web programming:

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

💡 Environment pointer lets the button’s click handler find the message to display
Summary: Return Function Results

- Use closure to maintain static environment
- May need to keep activation records after return
  - Stack (lifo) order fails!
- Possible “stack” implementation
  - Forget about explicit deallocation
  - Put activation records on heap
  - Invoke garbage collector as needed
  - Not as totally crazy as it sounds
    May only need to search reachable data
Summary of scope issues

- Block-structured lang uses stack of activation records
  - Activation records contain parameters, local vars, ...
  - Also pointers to enclosing scope
- Several different parameter passing mechanisms
- Tail calls may be optimized
- Function parameters/results require closures
  - Closure environment pointer used on function call
  - Stack deallocation may fail if function returned from call
  - Closures *not* needed if functions not in nested blocks