Scope, Function Calls and Storage Management

John Mitchell

Reading: Chapter 7, Concepts in Programming Languages

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

Scope, Function Calls and Storage Management

John Mitchell

Reading: Chapter 7, Concepts in Programming Languages

Announcements

• Midterm exam
  – Wed 10/26, 7-9PM, Room TBA
  – Local SCPD students are required to come to campus
  – Closed book, one page of notes allowed (tentatively)
• Homework 1
  – Due today 5PM
  – Turn in two parts separately
    • Printed on paper: all except 3 code solutions to Haskell problems
    • Electronically on CourseWare: 3 code solutions to Haskell problems
Turn in paper solutions now in class or in homework drop box by 5PM. Since this is first electronic submission, we will allow code solutions to be submitted up to 8PM tonight.
  – See web site

Block-Structured Languages

• Nested blocks, local variables
  – Example
    { int x = 1;
    { int y = 5;
      y = x;
    }
    
  – 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, Haskell 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

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

\begin{verbatim}
{ int x = ...;
 int y = ...;
 { int z = (x+y)*(x-y);
 };
{ int x = ...;
 int y = ...;
 { int z = (x+y)*(x-y);
 };
\end{verbatim}

Line declaration of x hides outer one.
Called "hole in scope"
Lifetime of outer x includes time when inner block is executed
Lifetime \neq 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

```
{ 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

Scoping rules

- Global and local variables
  \begin{verbatim}
  x, y are local to outer block
  z is local to inner block
  x, y are global to inner block
  \end{verbatim}

- 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)
  
  **Procedure P (<pars>)**
  
  ```
  procedure P (<pars>)
  begin
  <local vars>
  <proc body>
  end;
  ```

  **Function f(<pars>)**
  
  ```
  function f(<pars>)
  begin
  {<local vars>}
  <func 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

Example

- **Function**
  
  ```
  fact(n) = if n <= 1 then 1 else n * fact(n-1)
  ```

  - **Parameter**
    - set to value of n by calling sequence
  
  - **Intermediate result**
    - locations to contain value of fact(n-1)

Function call

- **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

Function return

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 function**

```plaintext
function f(x) =
    { x = x + 1; return x; }

var y = 0;
print(f(y) + y);
```

**Activation records**

- **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 program
  - Access link depends on static form of program text

**Static scope with access links**

- **Example**
  - Function \( g \) makes a *tail call* to function \( f \) if
    - Return value of function \( f \) is return value of \( g \)
  - Example tail call
    - \( g(x) = \text{if } x > 0 \text{ then } f(x) \text{ else } f(x)^2 \)
  - 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*x, y);

f(1,3) + 7;

Tail recursion and iteration

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

Tail recursion elimination

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

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

Complex nesting structure

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

function g(y) {
  var x = 1;
  ...}

Simplified code has same block nesting,
if we follow convention that each
declaration begins a new block.

JavaScript blocks and scopes

{ ( ) groups JavaScript statements
  — Does not provide a separate scope

• Blocks w/scope can be expressed using function
  — (function( ){} ) - create function of no args and call
  — Example
    var y=0;
    (function() {
    ...
    ...
    return y; 
    })();

Write as

RECAP
Translating examples to JS

Example and HW convention:
Each new declaration begins a new scope

Pass function as argument

There are two declarations of x
Which one is used for each occurrence of x?

Static Scope for Function Argument

How is access link for h(3) set?

Result of function call

Closures

- Function value is pair closure = \langle env, 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

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

```
Code for f
f
access
Run-time stack with access links
x
access
Code for g
```

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
  - 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

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

«ML»

```
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);
```

«JS»

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

```ml
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);
```

Closures in Web programming

- Useful for event handlers in Web programming:
  ```js
  function AppendButton(container, name, message) {
    var btn = document.createElement('button');
    btn.innerHTML = 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 activ 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