Scope, Function Calls and Storage Management

John Mitchell

Revised class schedule

- Friday Oct 17
  - No lecture; discussion section as usual
- Friday Oct 24
  - No section
- Monday Oct 27
  - Review section during class meeting time, Gates B01
- Wednesday Oct 29
  - No lecture
  - Evening exam: 7PM, Gates B01 and B03

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 \(\Rightarrow\) implementation complexity

Block-Structured Languages

- Nested blocks, local variables
  - Example
    
    ```
    \{ int x = 2;
    \{ int y = 3;
    x = y + 2;
    \}
    \}
    ```
  - Storage management
    - Enter block: allocate space for variables
    - Exits block: some or all space may be deallocated

Examples

- Blocks in common languages
  - C { ... }
  - 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 vars

Simplified Machine Model

- Registers
- Code
- Data
- Stack
- Heap
- Program Counter
- Environment Pointer
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 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

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

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

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

```c
{ 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)
  - `procedure P (<pars>) { <local vars> <proc body> }` for Algol.
  - `function f(<pars>) { <local vars> <function body> }` for C.

- Activation record must include space for:
  - parameters
  - return address
  - return value (an intermediate result)

Example

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

- 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

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

- Return address omitted; would be ptr into code segment

Topics for first-order functions

- Parameter passing
  - use ML reference cells to describe pass-by-value, pass-by-reference

- 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
ML imperative features (review)

- 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

- ML reference cells and assignment
  - Different types for location and contents
    - $x : \text{int}$ non-assignable integer value
    - $y : \text{int ref}$ location whose contents must be integer
    - $\text{ref } x$ expression creating new cell initialized to $x$
  - ML form of assignment
    - $y := x + 3$ place value of $x + 3$ in location (cell) $y$
    - $y := !y + 3$ add 3 to contents of $y$ and store in location $y$

Parameter passing

- Pass-by-reference
  - Caller places L-value (address) of actual parameter in activation record
  - Function can assign to variable that is passed

- Pass-by-value
  - Caller places R-value (contents) of actual parameter 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 : int = 0; print f(y)+y;`

- Standard ML
  - `fun f (x : int ref) =
    ( x := !x+1; !x ;
    y := ref 0 : int ref;
    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

```ml
int x=1;
function g(z) = x+z;
function f(y) =
{ int 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
- Environment
  - Parameters
  - Intermediate results
  - Local variables
- Control link
  - Depends on dynamic behavior of program
- Access link
  - Depends on static form of program text

Complex nesting structure

- Simplify to
  - `int x=1;` function $g(z) = x+z;`
  - `function f(y) =
    { int x = y+1;
      return g(y*x) ;
    }`

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

**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**
  ```plaintext```
  ```text
  fun g(x) = if x>0 then f(x) else f(x)*2
  ```
  ```plaintext```
- **Optimization**
  - Can pop activation record on a tail call.
  - Especially useful for recursive tail call.
  - Next activation record has exactly same form.

**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.
Example

Why this example here at this point in the lecture???

◆ Map function
  \[
  \text{fun map (f, nil) = nil | map(f, x:xs) = f(x) :: map(f, xs)}
  \]

◆ Modify repeated elements in list
  \[
  \text{fun modify(x) =}
  \begin{align*}
  &\text{let c = ref (hd l)} \\
  &\text{fun f(y) = ((if y = !c then c:=y+1 else c:=y); !c)} \\
  &\text{in} \\
  &\text{(hd l) :: map(f, tl l)} \\
  &\text{end;}
  \end{align*}
  \]
  \[
  \text{modify [1,2,2,3,4] \Rightarrow [1,2,3,4,5]}
  \]

Exercise: pure functional version of modify

Pass function as argument

\[
\begin{align*}
\text{val x = 4;} \\
\text{fun f(y) = x*y;} \\
\text{fun g(h) = let} \\
\text{val x=7 in} \\
\text{h(3) + x;} \\
\text{g(f);} \\
\text{g(f);} \\
\end{align*}
\]

There are two declarations of \( x \)

Which one is used for each occurrence of \( x \)?

Static Scope for Function Argument

\[
\begin{align*}
\text{val x = 4;} \\
\text{fun f(y) = x*y;} \\
\text{fun g(h) = let} \\
\text{val x=7 in} \\
\text{h(3) + x;} \\
\text{g(f);} \\
\end{align*}
\]

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

Closures

◆ Function value is pair closure = (env, code)

◆ 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

Run-time stack with access links
Function Argument and Closures

Run-time stack with access links

```plaintext
(int x = 4;

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

(int g(int h) {
    int x = 7;
    return h(3)+x;

}, g(f);
)

})
```

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
  ```plaintext
  fun compose(f,g) = (fn x => 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

```plaintext
fun mk_counter (int init : int) = 
    let
        val count = ref init
        fun counter(int inc:int) { return count := !count + inc; !count}
    in
        counter
    end;

val c = mk_counter(1);
print c(2) + c(2);
```

Function Results and Closures

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

val c = mk_counter(1);
print c(2) + c(2);
```

Example: Return fctn with private state

```plaintext
(int→int mk_counter (int init) {
    int count = init;
    int counter(int inc) { return count += inc;}
    return counter
}

(int→int c = mk_counter(1);
print 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

```c
int mk_counter(int init) {
    int count = init;    int counter(int inc) { return count+=inc;}
}
int c = mk_counter(1);    print c(2) + c(2);
```

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

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