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

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

Block-Structured Languages

- Nested blocks, local variables
  - Example
    ```
    int x = 2;
    int y = 3;
    new variables declared in nested blocks
    x = y + 2;
    }
    ```
  - Storage management
    - Enter block: allocate space for variables

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 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

```
int x = ...;
int y = ...;

void f() {
  int z = (x + y) * (x - y);
}
```

- 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

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

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);
  };
}
```

Scoping rules

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

- 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> <type> function f<pars>
  begin
  <local vars> <local vars>
  <proc body> <function body>
  end;
  }

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

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

• 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

- 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 \( y \)

ML examples

- Create cell and change contents
  \( \text{val} \ \text{x} = \text{ref} \text{"Bob"}; \)
  \( x := \text{"Bill"}; \)
- Create cell and increment
  \( \text{val} \ \text{y} = \text{ref} 0; \)
  \( y := y + 1; \)
- While loop
  \( \text{val} \ i = \text{ref} 0; \)
  \( \text{while} \ i < 10 \ \text{do} \ i := i + 1; \)

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)

Access to global variables

- Two possible scoping conventions
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack
- Example
  \[
  \text{val} \ x = 1; \\
  \text{function} \ g(x) \ { \text{return} \ x + 1; } \\
  \text{function} \ f(y) \ { \\
    \text{let} \ x = \ y + 1; \\
    \text{return} \ g(y*x); \\
  } \\
  \]

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
Complex nesting structure

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

Tail recursion

Function g makes a *tail call* to function f if:
- Return value of function f is return value of g

Example

```
f(1,3)
```

```
control
return val
x 1
y 3
```

```
fun g(x) = if x>0 then f(x) 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

Tail recursion elimination

```
f(1,3)
```

```
control
return val
x 1
y 3
```

```
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

```
f(1,3)
```

```
control
return val
x 1
y 3
```

```
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
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

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

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

Static Scope for Function Argument

Static Scope for Function Argument

Result of function call

Closures

• Function value is pair \( \langle \text{env}, \text{code} \rangle \)
• 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

val x = 4;
fun f(y) = x*y;
fun g(h) =
  let
  val x = 7
  in
  h(3) + x
  end;
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 active records to find global vars
  - Still deallocate active records using stack (lifo) order

Return Function as Result

- Language feature
  - Functions that return "new" functions
  - Need to maintain environment of function
- Example
  fun compose(f,g) = (fn x => g(f(x)));
  g(h) =
  let
  val x = 7
  in
  h(3) + x
  end;
g(f);

Example: Return fctn 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);

Example: Return fctn with private state

(int--int).mk_counter (int init) {
  int count = init;
  int counter(int inc) { return count += inc; }
  return counter
}
(int--int) c = mk_counter1;
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

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

```
c
```

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

Closures in Web programming

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