Example: Swap Two Values

- **ML**
  ```ml
  fun swap(x,y) = 
    let val z = !x in x := !y; y := z end;
  val swap = fn : 'a ref * 'a ref -> unit
  ```

- **C++**
  ```cpp
  template <typename T>
  void swap(T& x, T& y){
    T tmp = x;  x=y;  y=tmp;
  }
  ```

  Declarations look similar, but compiled very differently

Topics

- **Block-structured languages and stack storage**
  - 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

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
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

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

Environment Pointer

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

These are same until we consider function calls.

Functions and procedures

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

- **Parameter**
  - set to value of \( n \) by calling sequence

- **Intermediate result**
  - locations to contain value of \( \text{fact}(n-1) \)
**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

**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);
```

**activation records**

```
f(y)  y  0
f(x)  y  1
Control link
Return, result addr
x
```

**Access to global variables**

- **Two possible scoping conventions**
  - Static scope: refer to closest enclosing block
  - Dynamic scope: most recent activation record on stack

- **Example**

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

Static scope with access links

Control link
- Control link
- Access link
- Return address
- Return result addr
- Parameters
- Local variables
- Intermediate results

Environment
Pointer

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
g(12)
f(3)
control link
access link

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
Pass function as argument

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

Pseudo-JavaScript
{ var x = 4;
  { function f(y) {return x*y};
    fun f(y) = x*y;
  }
  fun g(h) =
    let
      int x = 7;
      return h(3) + x;
    }
  }
g(f);

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;
    return h(3) + x;
  }
g(f);

Result of function call

Code for f
Code for g
Code for h

follow access link
local var

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

- int x = 4;
  fun f(y) = x*y;
- fun g(h) =
  let
    int x=7
  in
    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
  - function compose(f,g)
    {return function(x) { return g(f(x)) }};

Example: Return fctn with private state

ML:

fun mk_counter (init : int) =
  let
    val count = ref init
  in
    fun counter(inc:int) =
      (count := !count + inc; !count)
  end;
val c = mk_counter(1);
c(2) + c(2);
Example: Return function with private state

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

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