More on Runtime Environments

• How to efficiently implement procedure call and return in the presence of higher-order functions?
  1. what are higher-order functions?
  2. how to extend stack frames to support higher-order functions?
  3. efficiency issues (execution time, space usage)?

• How to efficiently support memory allocation and de-allocation?
  1. what are the data representations?
  2. what is the memory layout?
  3. explicit vs implicit memory de-allocation? (malloc-free vs. garbage collection)

Procedure Parameters (in Pascal)

• Procedure parameters permit procedures to be invoked “out-of-scope”:

1. program main(input, output);
2. procedure b(function h(n: integer): integer);
3. var n: integer;
4. begin n := 6; writeln(h(2)) end;
5. procedure c;
6. var n: integer;
7. function f(n: integer): integer;
8. begin f := n + n end;
9. begin n := 0; b(f) end;
10. begin c end.

• Question: how to get the correct environment when calling h inside b?
• Solution: must pass static link along with f as it had been called at the point it was passed (line 11).

Restrictions in C & Pascal

• C does not allow nested procedures --- names in C are either local to some procedure or are global and visible in all procedures. Procedures in C can be passed as arguments or returned as results.

• Pascal (or Modula-2, Modula-3, Algol) allows procedure declarations to be nested, but procedure parameters are of restricted use, and procedures cannot be returned as result.

• Functional languages (e.g. ML, Haskell, Scheme, Lisp) support higher-order functions --- supporting both nested procedures and procedures passed as parameters or returned as results. supporting it is a big challenge to the compiler writers!
### Procedure Activations

#### Nested Functions in ML

```ml
val BIG = big(N)
fun P(v,w,x,y) =
  let
    fun Q() =
      let val u = hd(v)
      fun R() =
        P(v,u,u,y)
      in
        R()
      end
  in
    Q()
  end
val result = P(BIG,0,0,0)
```

### Procedure Activations (cont’d)

#### Nested Functions in ML

```ml
val BIG = big(N)
fun P(v,w,x,y) =
  let
    val u = hd(v)
    fun R() =
      P(v,u,u,y)
  in
    R()
  end
val result = P(BIG,0,0,0)
```

### Higher-Order Functions

#### How to create a closure for Q?

```ml
fun P(v,w,x,y) =
  let
    fun Q() =
      let val u = hd(v)
      fun R() =
        (u,w+x+y+3)
      in
        R()
      end
  in
    Q()
  end
val S = P(BIG,0,0,0)
val result = S()
```

### Higher-Order Functions (cont’d)

#### Q lost track of its environment

```ml
fun P(v,w,x,y) =
  let
    val u = hd(v)
    fun R() =
      (u,w+x+y+3)
  in
    R()
  end
val S = P(BIG,0,0,0)
val result = S()
```
Higher-Order Functions (cont’d)

Q must copy the frame!

```latex
fun P(v,w,x,y) = 
    let
        fun Q() = 
            let val u = hd(v)
            fun R() = 
                ... (u, w+x+y+3) ...
            in 
                ... R() ...
            end
        in
            ... R() ...
        end
    in
        ... R() ...
    end

val S = P(BIG,0,0,0)
val result = S()
```

Higher-Order Functions (cont’d)

Q’s environment is in the heap!

```latex
fun P(v,w,x,y) = 
    let
        fun Q() = 
            let val u = hd(v)
            fun R() = 
                ... (u, w+x+y+3) ...
            in 
                ... R() ...
            end
        in
            ... R() ...
        end
    in
        ... R() ...
    end

val S = P(BIG,0,0,0)
val result = S()
```

Applying Higher-Order Functions

Accessing the Closure Q!

```latex
fun P(v,w,x,y) = 
    let
        fun Q() = 
            let val u = hd(v)
            fun R() = 
                ... (u, w+x+y+3) ...
            in 
                ... R() ...
            end
        in
            ... R() ...
        end
    in
        ... R() ...
    end

val S = P(BIG,0,0,0)
val result = S()
```

Nested Higher-Order Functions

```latex
fun P(v,w,x,y) = 
    let
        fun Q() = 
            let val u = hd(v)
            fun R() = 
                ... (u, w+x+y+3) ...
            in 
                ... R() ...
            end
        in
            ... R() ...
        end
    in
        ... R() ...
    end

val S = P(BIG,0,0,0)
val result = S()
```
Linked Closures

fun P(v,w,x,y) = let
  fun Q() = let val u = hd(v)
  fun R() = ...(u,w+x+y+3)...
  in R
  end
in Q
end
val T = P(BIG,0,0,0)
val result = T()

Fast creation, Slow access!

Flat Closures

fun P(v,w,x,y) = let
  fun Q() = let val u = hd(v)
  fun R() = ...(u,w+x+y+3)...
  in R
  end
in Q
end
val S = P(BIG,0,0,0)
val T = S()
val result = T()

Slow creation, Fast access!

Better Representations?

- Closures cannot point to stack frame
  (different life time, so you must copy.)
- Linked closures --- fast creation, slow access
  Flat closures --- fast creation, fast access
- Stack frames with access links are similar to linked closures
  (accessing non-local variables is slow.)

GOAL: We need good closure representations that have both fast access and fast creation!

Space Usage

Space Leaks for Linked Closures

fun P(v,w,x,y) = let
  fun Q() = let val u = hd(v)
  fun R() = (u,w+x+y+3)
  in R
  end
in Q
end
fun loop (n,res) = if n<1 then res
  else (let val S = P(big(N),0,0,0)
         val T = S()
         in loop(n-1,T::res)
         end)
val result = loop(N,[])

Linked Closures: \(O(N^2)\)

Flat Closures: \(O(N)\)
Space Usage (cont’d)

Space Leaks for Stack Allocations

fun P(x) = ......
fun Q(n) = let
  val u = big(n)
  val v = P(u)
  val w = hd(u)
in if n > 0
  then Q(n-1)+v(w)
  else ...
end
val result = Q(N)

“u” is dead after this call!

Better Space Usage?

• The safe for space complexity rule:

  Local variable must be assumed dead after its last use within its scope!

• Stacks and linked closures are NOT safe for space
• Flat closures are safe for space
• SML/NJ: unsafe version = (2 to 80) x safe version

Efficient Heap-based Compilation

An efficient heap-based scheme has the following advantages:

• very good space usage (safe for space complexity !)
• very fast closure creation and closure access
• closures can be shared with activation records
• fast call/cc and fast generational GC
• simple implementation
Pure Heap-based Scheme

**Main Ideas:**
- no runtime stack!
- safely linked closures
- good use of registers

**Memory Layout**

- **STATIC**
  - code (code and globals)
- **HEAP**
  - (dynamic data)
  - (activation records)
- **REGISTERS**
  - garbage collector

---

**Safely Linked Closures**

*Safe for Space: use $O(N)$ space*

```plaintext
fun P(v, w, x, y) = 
  let
    fun Q() = 
      let
        val u = hd(v)
      in
        R()
      end
    in Q end
  in Q end

fun loop (n, res) = 
  if n<1 then res
  else (let
        val T = S()
    in loop(n-1, T::res)
  end)

val result = loop(N, [])
```

---

**Safely Linked Closures (cont'd)**

*Shorter Access Path!*

```plaintext
fun P(v, w, x, y) = 
  let
    fun Q() = 
      let
        val u = hd(v)
      in
        R()
      end
    in Q end
  in Q end

val T = P(big(N), 0, 0, 0)
```

---

**Good Use of Registers**

- To avoid memory traffic, modern compilers often pass arguments, return results, and allocate local variables in machine registers.
- Typical parameter-passing convention on modern machines:
  
  - the first $k$ arguments ($k = 4$ or $6$) of a function are passed in registers $R_p,...,R_p+k-1$, the rest are passed on the stack.
- Problem: extra memory traffic caused by passing args. in registers

```plaintext
function g(x : int, y : int, z : int) : int = x*y*z

function f(x : int, y : int, z : int) = 
  let
    val a = g(z+3, y+3, x+4) 
  in
    a*x+y+z end
```

Suppose function $\xi$ and $g$ pass their arguments in $R_p,R_{p+1},R_{p+2}$; then $\xi$ must save $R_1,R_2,$ and $R_3$ to the memory before calling $g$. 
Good Use of Registers (cont’d)

how to avoid extra memory traffic?

- **Leaf procedures** (or functions) are procedures that do not call other procedures; e.g., the function `exchange`. The parameters of leaf procedures can be allocated in registers without causing any extra memory traffic.
- Use **global register allocation**, different functions use different set of registers to pass their arguments.
- Use register windows (as on SPARC) --- each function invocation can allocate a fresh set of registers.
- Allocate **closures** in registers or use callee-save registers.
- When all fails --- save to the stack frame or to the heap.

Closures in Registers? No!

Module FOO: (in file “foo.sml”)

```plaintext
fun pred(x) = ...v(w,x) ...
val result = BAR.filter(pred, ...)
```

"pred" is an escaping function!
Its closure must be built on the heap!

Module BAR: (in file “bar.sml”)

```plaintext
fun filter(p,l) = let
  fun h(s,z,
      rev, p) = 
    if (s=[]) then rev z
    else
      (let val a = car s
          in if p a then h(r,a::z)
          else h(r,z)
      end)
    end
  in h(l,[])
end
```

Escaping functions:
functions whose call sites are not all known at compile time!

Closures in Registers? Yes!

```plaintext
fun filter(p,l) = let
  fun h(s,z) = 
    if (s=[]) then rev z
    else
      (let val a = car s
          in if p a then h(r,a::z)
          else h(r,z)
      end)
    end
in h(l,[])
end
```

Known functions:
functions whose call sites are all known at compile time!

“Lambda Lifting”

```plaintext
fun filter(p,l) = let
  fun h(s,z,rev,p) = 
    if (s=[]) then rev z
    else
      (let val a = car s
          in if p a then h(r,a::z,rev,p)
          else h(r,z,rev,p)
      end)
    end
  in h(l,[])
end
```

known functions can be rewritten into functions that are fully closed!
(i.e., with no free variables!)
“Spilled Activation Records”

We do not know how “p” treats the registers!

<table>
<thead>
<tr>
<th>z</th>
<th>rev</th>
<th>p</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Y</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Must save and load everything here!

Callee-save Registers

Convention:
Reserve k special registers!

Every function promises to always preserve these registers!

Example: k=3 (r4, r5, r6)

```
fun f(g, u, v, w) = let val x = g(u, v) val y = g(x, w) in x+y+w end
```

Callee-save Registers (cont’d)

6 callee-save registers:

<table>
<thead>
<tr>
<th>z</th>
<th>rev</th>
<th>p</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

no need to save and load anymore!

Summary: A Uniform Solution

Take advantage of variable life time and compile-time control flow information!

“Spilled activation records” are also thought as closures!

- no runtime stack ---------- everything is sharable
- all use safely-linked closures --------- to maximize sharing
- pass arguments and return results in registers
- allocating most closures in registers
- good use of callee-save registers