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 are the memory layout?
  3. explicit vs implicit memory de-allocation? (malloc-free vs. garbage collection)

---

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 results.

- 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 Parameters (in Pascal)

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

  ```pascal
  program main(input, output);
  2
  3    procedure b(function h(n : integer): integer);
  4        var m : integer;
  5        begin
  6            n := 6; writeln(h(2)) end;
  7
  8    procedure c;
  9        var m : integer;
 10       function f(n: integer): integer;
 11           begin
 12              n := m + n end;
 13          begin m := 0; b(f) end;
 14          begin c end.
 15
  Question: how to get the correct environment when calling h inside b?

  Solution: must pass static link along with f as if it had been called at the point it was passed (line 11).
  ```

---

Traditional Stack Scheme
Procedure Activations

**Nested Functions in ML**

```plaintext
val BIG = big(N)

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

**Procedure Activations (cont’d)**

**Nested Functions in ML**

```plaintext
val BIG = big(N)

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

Higher-Order Functions

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

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

**Higher-Order Functions (cont’d)**

**Q lost track of its environment**

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

Q must copy the frame!

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

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

Higher-Order Functions (cont’d)

Q’s environment is in the heap!

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

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

Apply Higher-Order Functions

Accessing the Closure Q!

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

Nested Higher-Order Functions

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

```
Linked Closures

\[
\text{fun } P(v,w,x,y) = \\
\quad \text{let } \\
\quad \quad \text{fun } Q() = \\
\quad \quad \quad \text{let } \text{val } u = \text{hd}(v) \\
\quad \quad \quad \quad \text{fun } R() = \\
\quad \quad \quad \quad \quad \ldots(u,w+x+y+3)\ldots \\
\quad \quad \quad \text{in } R \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{in } Q \\
\text{end} \\
\text{val } S = P(\text{BIG},0,0,0) \\
\text{val } T = S() \\
\text{val } \text{result} = T() \\
\]

Fast creation, Slow access!

Flat Closures

\[
\text{fun } P(v,w,x,y) = \\
\quad \text{let } \\
\quad \quad \text{fun } Q() = \\
\quad \quad \quad \text{let } \text{val } u = \text{hd}(v) \\
\quad \quad \quad \quad \text{fun } R() = \\
\quad \quad \quad \quad \quad \ldots(u,w+x+y+3)\ldots \\
\quad \quad \quad \text{in } R \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{in } Q \\
\text{end} \\
\text{val } S = P(\text{BIG},0,0,0) \\
\text{val } T = S() \\
\text{val } \text{result} = T() \\
\]

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 --- slow 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

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

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

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

**Space Leaks for Stack Allocations**

```plaintext
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

---

**Drawbacks of Stack Allocation**

- inefficient space usage
- slow access to non-local variables
- expensive copying between stack and heap (activation records cannot be shared by closures)
- scanning roots is expensive in generational GC
- very slow first-class continuations (call/cc)
- correct implementation is complicated and messy

---

**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

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

Safely Linked Closures

Safe for Space: use \( O(N) \) space

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

THE TRICK:
- Variables \( w,x,y \) have same life time!

Safely Linked Closures (cont’d)

Shorter Access Path!

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

THE TRICK:
- Variables \( w,x,y \) have same life time!

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 \( R_{p} \ldots R_{p+k-1} \), the rest are passed on the stack.
- Problem: extra memory traffic caused by passing args. in registers

\[
\text{function} \ g(x : \text{int}, y : \text{int}, z : \text{int}) : \text{int} = x*y*z \\
\text{function} \ f(x : \text{int}, y : \text{int}, z : \text{int}) = \\
\text{let} \ a := g(z+3, y+3, x+4) \text{ in} a*x+y+z \text{ end} \\
\]

Suppose function \( f \) and \( g \) pass their arguments in \( R_{1} \), \( R_{2} \), \( R_{3} \), then \( f \) 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 ? Yes !

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

```
"h" is a known function !
Its closure can be put in registers !
(e.g., \{rev,p\})
```

Closures in Registers ? No !

```
Module FOO: (in file “foo.sml”)
fun pred(x) = ...v(w,x) ...
val result = BAR.filter(pred,)
```

```
Module BAR: (in file “bar.sml”)
fun filter(p,l) = let
    fun h(s,z) =
        if (s=[[]]) then rev z
        else
            (let val a = car s
             val r = cdr s
             in if p a then h(r,a::z) else h(r,z)
             end)
        end
    in h(l,[]) end
```

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

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

“Lambda Lifting”

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

known functions can be rewritten into functions that are fully closed !
(i.e. with no free variables !)

Known functions:
functions whose call sites are all known at compile time !
“Spilled Activation Records”

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

```
| r0 | r1 | r2 | r3 | r4 | r5 |
```

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

```
e4, e5, e6
```

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

```
| r0 | r1 | r2 | r3 | r4 | r5 |
```

```
g | u | v | w |
```

```
A | B | C | D | E | F |
```

**Callee-save Registers (cont’d)**

6 callee-save registers:

```
e4, e5, e6, e7, e8, e9
```

```
| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 |
```

```
s | z | rev | p | |
```

```
r0 r1 r2 r3 r4 r5 r6 r7 r8 r9
```

**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