More on Runtime Environments

• How to **efficiently** implement procedure call and return in the presence of higher-order functions?

  • what are higher-order functions?
  • how to extend stack frames to support higher-order functions?
  • efficiency issues (execution time, space usage)?

• How to **efficiently** support memory allocation and de-allocation?

  • what are the data representations?
  • what are the memory layout?
  • 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 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 ...
      end
  in ...
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
    fun Q() =
      let val u = hd(v)
        fun R() =
          ...
        in ...
      end
  in ...
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() =
          ...
        in ...
      end
  in ...
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
    fun Q() =
      let val u = hd(v)
        fun R() =
          ...
        in ...
      end
  in ...
end
val S = P(BIG,0,0,0)
val result = S()
```
Higher-Order Functions (cont’d)

Q must copy the frame!

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

Applying Higher-Order Functions

Accessing the Closure Q!

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

Nested Higher-Order Functions

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

```haskell
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
    val S = P(BIG,0,0,0)
    val T = S()
    val result = T()
end
```

Fast creation, Slow access!

Flat Closures

```haskell
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
    val S = P(BIG,0,0,0)
    val T = S()
    val result = T()
end
```

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**

```haskell
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
    val S = P(BIG,0,0,0)
    val T = S()
    val result = T()
end
```

Linked Closures: $O(N^2)$

**Flat Closures**

```haskell
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,[])
```

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

Space Leaks for Stack Allocations

fun \( P(x) = \ldots \)

fun \( Q(n) = \begin{array}{l}
\text{let} \ v = P(u) \\
\text{val} \ w = \text{hd}(u) \\
\text{in} \ n > 0 \rightarrow Q(n-1) + v(w) \\
\text{else} \ldots \end{array} \)

val result = \( Q(N) \)

“u” is dead after this call!

Better Space Usage?

• The safe for space complexity rule:

\[ \text{Local variable must be assumed dead after its last use within its scope!} \]

• 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
- **Stack** (code and globals)
- **Heap** (dynamic data)
- **Garbage collector**

---

**Safely Linked Closures**

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

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

**THE TRICK:**
- Variables $w, x, y$ have same life time!

---

**Safely Linked Closures (cont’d)**

**Shorter Access Path!**

\[
\begin{align*}
\text{fun } P(v, w, x, y) &= \text{let fun } Q() = \\
\text{let val } u = \text{hd}(v) \\
\text{fun } R() &= (u, w+x+y+3) \\
\text{in } (S, u) \\
\text{in } R \\
\text{in } Q \\
\text{val } T = P(\text{big}(N), 0, 0, 0)
\end{align*}
\]

The number of links traversed is at most 1.

**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

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

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? No!**

Module FOO: (in file “foo.sml”)

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

"pred" is an escaping function!

Its closure must be built on 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)
  in h(l,[])
end
```

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

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

---

**“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)
  in h(l,[] rev,p)
end
```

known functions can be rewritten into functions that are fully closed!

---

Michelle Zhang, Yale University
“Spilled Activation Records”

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

Must save and load everything here!

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

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

fun f(g, u, v, w) =
let
  x = g(u, v)
  y = g(x, w)
end

in
  x + y + w

end

f return

Callee-save Registers (cont’d)

6 callee-save registers:

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