CS415 Compilers

Procedure abstractions

part 3

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Project 3 has been posted and is due Wednesday, May 8

• Homework #8 has been posted and is due Monday, May 6

• Final exam: May 14, noon - 3:00pm, SEC-118 (our current room)

CONFLICTS?
If you have a conflict, please send me the details of your conflict:
   class, email of instructor, time of scheduled exam

NEED TO KNOW BY MONDAY, MAY 6.
Activation Record Basics

- **parameters**: Space for parameters to the current routine
- **register save area**: Saved register contents
- **return value**: If function, space for return value
- **return address**: Address to resume caller
- **addressability**: Help with non-local access
- **caller’s ARP**: To restore caller’s AR on a return (control link)
- **local variables**: Space for local values & variables (including spills)

One AR for each invocation of a procedure
Example: Dynamic vs. Static Views

```c
int r (...) { // declaration
    int d, s;

    int q (x,y) // declaration
        int x,y;
        {
            return x + y + d ;
        }

    int p (a,b,c) // declaration
        int a, b, c;
        {
            int d;
            if (...)
                d = q (c,b); // call
            else
                d = p (a, d, c); // call
        }

    s = p(10, d, s); // call
    s = p(11, s, d); // call
}
```
Example: Dynamic vs. Static Views

(1) dynamic activation tree

(2) dynamic activation records in runtime stack

(3) static symbol table inside proc q

symbol table inside proc p

d:0.0
d:0.0
s:0.1
s:0.1
x:1.0
a:1.0
y:1.1
b:1.1
c:1.2
d:1.3

nesting level

offset

lexical scoping

```
int r (...) { // declaration
    int d, s;

    int q (x,y) // declaration
        int x,y;
        { //
            return x + y + d;
        }

    int p (a,b,c) // declaration
        int a, b, c;
        {
            int d;
            if (...) // call
                d = q(c,b);
            else // call
                d = p(a, d, c);
        }

    s = p(10, d, s); // call
    s = p(11, s, d); // call
```
Lexical Scoping: Translating Local Names

How does the compiler represent a specific instance of $x$?

- **Name is translated into a *static coordinate***
  - $\langle \text{level}, \text{offset} \rangle$ pair
  - "level" is lexical nesting level of the procedure
  - "offset" is *unique* within that scope
- **Subsequent code will use the static coordinate to generate addresses and references**
- "level" is a function of the table in which $x$ is found
  - Stored in the entry for each $x$
- "offset" must be assigned and stored in the symbol table
  - Assigned at compile time
  - Known at compile time
  - Used to generate code that executes at run-time
Activation Record Details

How does the compiler find the variables?
• They are at known offsets from the AR pointer
• The static coordinate leads to a “loadAI” operation
  → Level specifies an ARP, offset is the constant

Variable-length data
• If activation record (AR) can be extended, put it below local variables
• Leave a pointer at a known offset from ARP
• Otherwise, put variable-length data on the heap

Initializing local variables
• Must generate explicit code to store the values
• Among the procedure’s first actions
Where do activation records live?

- If lifetime of AR matches lifetime of invocation, **AND**
- If code normally executes a “return”
  \[ \Rightarrow \text{Keep ARs on a stack} \]

- If a procedure can outlive its caller, **OR**
- If it can return an object that can reference its execution state
  \[ \Rightarrow \text{ARs must be kept in the heap} \]

- If a procedure makes no calls
  \[ \Rightarrow \text{AR can be allocated statically} \]

Efficiency prefers static, stack, then heap
Establishing Addressability

Must create base addresses

- **Global & static variables**
  - Construct a label by mangling names (\textit{i.e., }\&\_fee)

- **Local variables**
  - Convert to static data coordinate and use \texttt{ARP} + offset

- **Local variables of other procedures**
  - Convert to static coordinates (level, offset)
  - Find appropriate \texttt{ARP}
  - Use that \texttt{ARP} + offset

\{ Must find the right \texttt{AR} \\
Need links to nameable \texttt{ARs} \}
Establishing Addressability

Using access links (static links)

- Each AR has a pointer to most recent AR of immediate lexical ancestor (mylevel - 1)

- Lexical ancestor need not be the caller

Reference to \(<p,16>\) runs up access link chain to \(p\)

Cost of access is proportional to lexical distance
Establishing Addressability

Using access links

<table>
<thead>
<tr>
<th>SC</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2,8&gt;</td>
<td>loadAl r₀, 8 ⇒ r₂</td>
</tr>
<tr>
<td>&lt;1,12&gt;</td>
<td>loadAl r₀, -4 ⇒ r₁&lt;br&gt;loadAl r₁, 12 ⇒ r₂</td>
</tr>
<tr>
<td>&lt;0,16&gt;</td>
<td>loadAl r₀, -4 ⇒ r₁&lt;br&gt;loadAl r₁, 12 ⇒ r₁&lt;br&gt;loadAl r₁, 16 ⇒ r₂</td>
</tr>
</tbody>
</table>

Assume
- Current lexical level is 2
- Access link is at ARP - 4

Maintaining access link
- Calling level \( k+1 \) (\( k \) is current level)
  → Use current ARP as link in new AR
- Calling level \( j < k \)
  → Find ARP for \( j - 1 \)
  → Use that ARP as link in new AR

Access & maintenance cost varies with level
All accesses are relative to ARP (\( r₀ \))
Most languages provide a parameter passing mechanism
⇒ Expression used at “call site” becomes a variable in callee

Two common binding mechanisms
• **Call-by-reference** passes a pointer to actual parameter
  → Requires slot in the AR (for address of parameter)
  → Multiple names with the same address (aliasing)?

• **Call-by-value** passes a copy of its value at time of call
  → Requires slot in the AR
  → Each name gets a unique location
  → Arrays are mostly passed by reference, not value

• **Can always use global variables** ...
Work on the project!

Wrapping up procedure linkage

Q&A