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
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

(3) static
symbol table inside proc q inside proc p

(2) dynamic activation records in runtime stack

Lexical scoping

(r) 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

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

loadAI r1, c1 ⇒ r2 : MEM( r1 + c1 ) → r2
Activation Record Details

Where do activation records live?
• If lifetime of AR matches lifetime of invocation, **AND**
• If code normally executes a "return"
  ⇒ 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
  ⇒ ARs must be kept in the heap
• If a procedure makes no calls
  ⇒ 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 (*i.e.*, &\_fee)

- **Local variables**
  
  → Convert to static data coordinate and use ARP + offset

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

{ Must find the right AR
  
  Need links to nameable 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

Some setup cost on each call
Establishing Addressability

Using access links

**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

<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>
</tbody>
</table>
| <1,12> | loadAl r₀, -4 ⇒ r₁  
                    loadAl r₁, 12 ⇒ r₂ |
| <0,16> | loadAl r₀, -4 ⇒ r₁  
                    loadAl r₁, -4 ⇒ r₁  
                    loadAl r₁, 16 ⇒ r₂ |

Access & maintenance cost varies with level
All accesses are relative to ARP $(r₀)$
Establishing Addressability

Using a display
- Global array of pointer to nameable ARs
- Needed ARP is an array access away

- Reference to \(<p,16>\) looks up \(p\)'s ARP in display & adds 16
- Cost of access is constant \((\text{ARP} + \text{offset})\)
Establishing Addressability

Using a display

<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>loadl _disp ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 4 ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 12 ⇒ r₂</td>
</tr>
<tr>
<td>&lt;0,16&gt;</td>
<td>loadl _disp ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 16 ⇒ r₂</td>
</tr>
</tbody>
</table>

Assume

- Current lexical level is 2
- Display is at label _disp

Maintaining access link

- On entry to level $j$
  - Save level $j$ entry into AR (Saved Ptr field)
  - Store ARP in level $j$ slot
- On exit from level $j$
  - Restore level $j$ entry

**Desired AR is at _disp + 4 x level**

**Access & maintenance costs are fixed**

**Address of display may consume a register**
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** ...
Establishing Addressability

Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
  - Overhead only incurred on references & calls
  - If ARs outlive the procedure, access links still work
- Display costs are fixed for all references
  - References & calls must load display address
  - Typically, this requires a register

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

*For either scheme to work, the compiler must insert code into each procedure call & return*
How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - Different calls may be in different compilation units
  - Compiler may not know system code from user code
  - All calls must use the same protocol

Compiler must use a standard sequence of operations
- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement (for interoperability)
Procedure Linkages

Standard procedure linkage

**procedure p**

<table>
<thead>
<tr>
<th>prolog</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>pre-call</td>
<td></td>
</tr>
<tr>
<td>post-return</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td></td>
</tr>
</tbody>
</table>

**procedure q**

<table>
<thead>
<tr>
<th>prolog</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td></td>
</tr>
</tbody>
</table>

Procedure has
- standard **prolog**
- standard **epilog**

Each call involves a
- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters
Procedure Linkages

Pre-call Sequence
• Sets up callee’s basic AR
• Helps preserve its own environment

The Details
• Allocate space for the callee’s AR
  → except space for local variables
• Evaluates each parameter & stores value or address
• Saves return address, caller’s ARP (control link) into callee’s AR
• If access links are used
  → Find appropriate lexical ancestor & copy into callee’s AR
• Save any caller-save registers
  → Save into space in caller’s AR
• Jump to address of callee’s prolog code
Procedure Linkages

Post-return Sequence

- Finish restoring caller’s environment
- Place any value back where it belongs

The Details

- Copy return value from callee’s AR, if necessary
- Free the callee’s AR
- Restore any caller-save registers
- Copy back call-by-value/result parameters
- Continue execution after the call
**Procedure Linkages**

**Prolog Code**
- Finish setting up callee’s environment
- Preserve parts of caller’s environment that will be disturbed

**The Details**
- Preserve any callee-save registers
- If display is being used
  - Save display entry for current lexical level
  - Store current ARP into display for current lexical level
- Allocate space for local data
  - Easiest scenario is to extend the AR
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables
Epilog Code

- Wind up the business of the callee
- Start restoring the caller’s environment

The Details

- Store return value?
  - Some implementations do this on the return statement
  - Others have return assign it & epilog store it into caller’s AR
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller’s ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)
Work on the project!

Some special topics in compilers/programming languages

Q&A