CS415 Compilers

Procedure abstractions

Part 3

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

• Third project – Local dead code pass for ILOC has been posted. Due on Monday, May 3

• Seventh homework has been posted. Due on Monday, May 3

• Fourth quiz has been posted. Due on Friday, April 30; topics are: program representations, syntax-directed translation schemes, and type systems; you have 60 minutes and two tries; last try counts

• “Finals” quiz during finals week: Friday, May 7 at noon to Tuesday, May 11, at noon; four top quiz scores count, i.e., “finals” quiz is optional; will cover procedure abstractions and runtime environments

• No regular lecture on Thursday; open floor for questions; will not be recorded
Announcements

- **Overall grade for class:**
  - Homeworks: Only count 6 best out of 7: \(6 \times 3\% = 18\%\)
  - Projects: \(3 \times 15\% = 45\%\)
  - Quizzes: Only count 4 best out of 5: \(4 \times 10\% = 40\%\)

\[103\%\]

- **Basic grading scale (may push thresholds “down”, but not “up”):**
  - A \(\geq 90\%\)
  - B+ \(\geq 87\%\)
  - B \(\geq 80\%\)
  - C+ \(\geq 77\%\)
  - C \(\geq 70\%\)
  - D \(\geq 60\%\)
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
}

(1) dynamic activation tree
(2) dynamic activation records in runtime stack
```c
int r (...) { // declaration
    int d:(0,4), s:(0,8);
}

int q (x,y) // declaration
    int x: (1,-8), y: (1,-12);
    {
        return x + y + d;
    }

int p (a,b,c) // declaration
    int a: (1,-8), b: (1,-12), c: (1,-16);
    {
        int d: (1,4);
        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
```

**(1) dynamic activation tree**

**Example: Dynamic vs. Static Views**

- **Dynamic activation records** in runtime stack
- **Lexical scoping**

**Static symbol table inside proc q**
- d: 0.0
- s: 0.1
- x: 1.0
- y: 1.1

**Dynamic symbol table inside proc p**
- d: 1.3
- c: 1.2
- b: 1.1
- a: 1.0

**Nesting level and offset**
Lexical Scoping: Translating Local Names

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

- Name is translated into a *static coordinate* $<\text{level},\text{offset}>$
  - "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
  - Known at compile time
  - 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
How does the compiler find the variables?

- They are at known offsets from the AR pointer
- The static coordinate leads to a “loadAI” operation

\[ \text{Level specifies an ARP, offset is the constant} \]

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
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
Most languages provide a parameter passing mechanism:
actual parameters are mapped to formal parameters

Common binding mechanisms:
- **Call-by-reference** passes a pointer to actual parameter
  - Requires slot in the AR (for *address* of parameter)
  - Expression used at “call site” becomes a variable in callee
  - Multiple names with the same address (aliasing)?
    
    e.g: `call fee(x,x,x)`

- **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
Most languages provide a parameter passing mechanism where actual parameters are mapped to formal parameters.

- **Call-by-value-result** passes the value of and a pointer to the actual parameter; at the end of the call, value of formal parameter is copied back into actual parameter.
  - Requires two slots in the AR
  - During execution of procedure body, formal parameter is treated as a call-by-value parameter,
  - Order of write-back is important

- Can always use global variables, which makes reasoning about programs harder.
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

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

Using access links

<table>
<thead>
<tr>
<th>SC</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2,8&gt;</td>
<td>loadAI r₀, 8 ( \Rightarrow r₂ )</td>
</tr>
</tbody>
</table>
| <1,12> | loadAI r₀, -4 \( \Rightarrow r₁ \)  
|       | loadAI r₁, 12 \( \Rightarrow r₂ \) |
| <0,16> | loadAI r₀, -4 \( \Rightarrow r₁ \)  
|       | loadAI r₁, -4 \( \Rightarrow r₁ \)  
|       | loadAI r₁, 16 \( \Rightarrow r₂ \) |

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

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 \(_\text{disp} + 4 \times \text{level}\)

Access & maintenance costs are fixed

Address of display may consume a register
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
• 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

prolog

pre-call

post-return

epilog

procedure q

prolog

epilog

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 and/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
**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!

Thursday Q&A session (not recorded)

Don’t forget to take the fourth quiz! Deadline: Friday, April 30

Good luck!