CS415 Compilers

Procedure Abstraction
Chapter 6.1 - 6.5

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

• Homework 8 due today, Wednesday, April 25
• Second project new due date: Friday, April 27
• Third project (local CSE) due Wednesday, May 2
• Homework 9: will be posted by tomorrow; not graded

Final exam: Tuesday, May 8, noon-3:00pm, Physics Lecture Hall, Busch Campus

Conflicts?

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

(1) dynamic
activation tree

(2) dynamic activation records
in runtime stack

(3) static
symbol table inside proc q 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       c:1.2
d:1.3       d:1.3

naming level  offset

lexical scoping
```
Each procedure creates its own name space
  • Any name (almost) can be declared locally
  • Local names obscure identical non-local names
  • Local names cannot be seen outside the procedure
    → Nested procedures are “inside” by definition
  • We call this set of rules & conventions “lexical scoping”

Examples
  • C has global, static, local, and block scopes (Fortran-like)
    → Blocks can be nested, procedures cannot
  • Scheme has global, procedure-wide, and nested scopes (let)
    → Procedure scope (typically) contains formal parameters
Why introduce lexical scoping?

- Provides a compile-time mechanism for binding “free” variables
- Simplifies rules for naming & resolves conflicts

How can the compiler keep track of all those names?

The Problem

- At point \( p \), which declaration of \( x \) is current?
- At run-time, where is \( x \) found?
- As parser goes in & out of scopes, how does it delete \( x \)?

The Answer

- Lexically scoped symbol tables

(see § 5.7.3)
OS needs a way to start the program’s execution

- Programmer needs a way to indicate where it begins
  - The “main” procedure in most languages

- When user invokes “grep” at a command line
  - OS finds the executable
  - OS creates a process and arranges for it to run “grep”
  - “grep” is code from the compiler, linked with run-time system
    - Starts the run-time environment & calls “main”
    - After main, it shuts down run-time environment & returns

- When “grep” needs system services
  - It makes a system call, such as fopen()
Where Do All These Variables Go?

Automatic & Local
• Keep them in the procedure activation record or in a register
• Automatic $\Rightarrow$ lifetime matches procedure’s lifetime

Static
• Procedure scope $\Rightarrow$ storage area affixed with procedure name
• File scope $\Rightarrow$ storage area affixed with file name
• Lifetime is entire execution

Global
• One or more named global data areas
• One per program, ...
• Lifetime is entire execution
Placing Run-time Data Structures

Classic Organization

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack &amp; Global</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
</table>

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved

Single Logical Address Space

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a virtual address space
How Does This Really Work?

The Big Picture

Compiler's view

Virtual address spaces

OS's view

Hardware's view

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

- Name is translated into a *static coordinate*
  - `< level,offset >` 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 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
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

\[
\text{loadAI } r1, c1 \Rightarrow r2 : \text{MEM}( r1 + c1 ) \rightarrow r2
\]
Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, AND
- If code normally executes a “return”
  \[ \Rightarrow \] 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
  \[ \Rightarrow \] ARs must be kept in the heap

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

Efficiency prefers static, stack, then heap
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 ...

**e.g:** call fee(x,x,x);
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 \(\langle p,16\rangle\) 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

<table>
<thead>
<tr>
<th>SC</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2,8&gt;</td>
<td>loadAI r_0, 8 \rightarrow r_2</td>
</tr>
</tbody>
</table>
| <1,12> | loadAI r_0, -4 \rightarrow r_1  
      | loadAI r_1, 12 \rightarrow r_2 |
| <0,16> | loadAI r_0, -4 \rightarrow r_1  
      | loadAI r_1, -4 \rightarrow r_1  
      | loadAI r_1, 16 \rightarrow r_2 |

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 \((r_0)\)
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 \((\text{ARP} + \text{offset})\)

Some setup cost on each call
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>\text{loadAl } r_0, 8 \Rightarrow r_2</td>
</tr>
</tbody>
</table>
| <1,12> | \text{loadl } \_\text{disp} \Rightarrow r_1  \\
|       | \text{loadAl } r_1, 4 \Rightarrow r_1  \\
|       | \text{loadAl } r_1, 12 \Rightarrow r_2 |
| <0,16> | \text{loadl } \_\text{disp} \Rightarrow r_1  \\
|       | \text{loadAl } r_1, 16 \Rightarrow r_2 |

Assume

- Current lexical level is 2
- Display is at label \_\text{disp}

Maintaining access link

- On entry to level \text{j}
  - Save level \text{j} entry into AR (Saved Ptr field)
  - Store ARP in level \text{j} slot
- On exit from level \text{j}
  - Restore level \text{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
  → 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
Procedure Linkages

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 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
**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
Procedure Linkages

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.)
Things to do and next class

Work on the project!

GPU and CUDA programming or

Dependence analysis and automatic parallelization?