Runtime Support for Algol-Like Languages
## Lab 3 Schedule

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>Docs Available Lab Lecture</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start work</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>Tutorial 4PM</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Complete &amp; Correct Dependence Graph</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Correct Schedule</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Improve Schedules</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>Thanksgiving Break</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lab 3 Due Date</td>
<td>24</td>
<td>Late day</td>
<td>Not a late day</td>
</tr>
</tbody>
</table>
Support for Name Spaces

In an ALL, the compiler needs
- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

In an OOL, the compiler needs
- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

This lecture focuses on runtime support for ALLs.

COMP 412, Fall 2018
Where Do Variables Live?

We have seen ARs, static data areas, global data areas, ...

How does the compiler decide where to place each variable?
• A combination of visibility and lifetime determines that placement

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>Scope</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>local</td>
<td>declaring procedure’s AR</td>
</tr>
<tr>
<td>static</td>
<td>any</td>
<td>named static data area for its lifetime</td>
</tr>
<tr>
<td>dynamic</td>
<td>any</td>
<td>heap</td>
</tr>
</tbody>
</table>

Variable length items?
• Keep descriptor in its “natural” place & allocate space in AR or heap
  – Requires one level of indirection (a “pointer”)
  – Allows uniform addressability
  – In AR if AR is extendible
  – On heap if AR is not extendible

Automatic: Lifetime matches procedure activation
Static: Lifetime may be as long as entire execution
Dynamic: Lifetime is under program control & not known at compile time
How Do We Address Variables?

Local variables
- Need a mechanism to locate the local data area in appropriate AR
- Represent the variable as a static coordinate: <level, offset>
  - Level is the lexical nesting level of the procedure that declares the variable
  - Offset is the offset within the AR’s local data area for that procedure
- Mechanism takes static coordinate to run-time address

Static variables (including global)
- Need a base-address, offset pair
  - Emit a load of the base address
  - Emit an add of offset and a load
  - Must know that it is non-local case
- mangled name for base address
  - compile time
  - loadA if offset is small enough
  - represent differently

Dynamic variables
- Programmer manages access through pointers, names, ...
  - not the compiler’s problem
How Do We Address Variables?

**Local variable of current procedure**
- Convert to static coordinate
- Add *offset* to \( r_{ARP} \) and load

**Local variable in surrounding lexical scope**
- Convert to static coordinate
- Convert *level* to \( ARP \) for procedure at that level
- Add *offset* to that \( ARP \) and load

Local variables of surrounding scopes, static variables at various scopes, and global variables are often called “free variables” in the PL literature.
Finding the Correct AR

Each AR has a field for addressability

One common scheme uses access links

- Each AR has a link to the AR of its lexical ancestor (surrounding scope)
- Lexical ancestor need not be the procedure that called it
- Compiler can use static coordinate to generate code that chases the chain of access links to find the correct AR
  - Reference to \(<p, 16>\) runs up chain to find level \(p\)'s ARP, then adds 16
  - Number of dereferences is \((c - p)\), where \(c\) is the current lexical level
- Cost of access is proportional to lexical distance between \(c\) and \(p\)

\(r_{\text{ARP}}\) is a physical register, dedicated to holding the ARP

COMP 412, Fall 2018
Using Access Links

Finding the Correct AR

Static Coord | Generated Code
---|---
<3, 8> | loadAI r_{ARP}, 8 => r_{10}
<2,12> | loadAI r_{ARP}, 4 => r_{10}
   | loadAI r_{10}, 12 => r_{10}
<1,16> | loadAI r_{ARP}, 4 => r_{10}
   | loadAI r_{10}, 4 => r_{10}
   | loadAI r_{10}, 16 => r_{10}

procedure fee {
    procedure fie {
        procedure foe {
            call foe()
        }
        call fie()
    }
Finding the Correct AR

Maintaining Access Links

At each call:

- In pre-call sequence, caller must find the appropriate ARP
- Assume caller is at level $k$
  - Callee at level $k+1$
    - Use current ARP as link
  - Callee at level $j$, $0 < j \leq k$
    - Walk up access link chain to level $j-1$
    - Use that ARP as link
  - Callee is global (e.g., at level 0)
    - Use NULL as link
    - Caller & callee share no lexical context
- As long as the code is correct
  - Caller’s chain will be long enough
  - Callee cannot walk off end of chain

### Access Registers
- parameters
- registers
- return value
- return address
- caller’s ARP
- addressability
- local variables
Finding the Correct AR

Each AR has a field for addressability

The Linkage Can Also Use A Display

- Global vector of pointers to the ARs of surrounding procedures
- Any nameable AR can be reached in constant time
  - Current AR is reached through ARP
  - Surrounding scopes are reached by treating display as a vector
- Reference to \(<p, 16>\), for \(p < k\), looks up the level \(p\) ARP in the display and adds 16 to it
- Cost of access is constant
Finding the Correct AR

procedure fee {
    procedure fie {
        procedure foe {
            call foe()
        }
        call fie()
    }
}

<table>
<thead>
<tr>
<th>Static Coord</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;3, 8&gt;</td>
<td>loadAI $r_{ARP}, 8 \rightarrow r_{10}$</td>
</tr>
</tbody>
</table>
| <2,12>       | loadAI `_display_ \rightarrow r_{10}$
|              | loadAI $r_{10}, 4 \rightarrow r_{10}$
|              | loadAI $r_{10}, 12 \rightarrow r_{10}$ |
| <1,16>       | loadI `_display_ \rightarrow r_{10}$
|              | load $r_{10} \rightarrow r_{10}$
|              | loadAI $r_{10}, 16 \rightarrow r_{10}$ |
Finding the Correct AR

Maintaining A Display

- Display maintenance is simple, as long as the compiler writer is willing to “waste” one display slot

- In prolog of each level $j$ procedure
  - Save the level $j$ display entry into the procedure’s own AR (addressability slot)
  - Store current ARP into level $j$ display slot

- In epilog of each level $j$ procedure
  - Restore old level $j$ display entry from the current AR’s addressability slot

Simple, clean, and constant time
Access Links versus Display

Access Links

- Maintenance adds overhead to each call
  - Overhead depends on nesting
- Adds overhead on each reference to a local in surrounding scope
  - Overhead depends on nesting
- If ARs outlive procedure, access links still work
  - Access links between ARs ensures that they are live (& uncollected)

Display

- Maintenance adds overhead to each call & return
  ♦ Overhead is constant
- Adds overhead on each reference to a local in surrounding scope
  ♦ Overhead is constant
- If ARs outlive procedure, display links are not preserved
  ♦ Need to preserve them in a way that ensures ARs are live (collector issue)

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Depends on demand for registers

*(available register to hold _display_ ?)*
Access Links and Displays

The Meta Issue

• Decision to use one or the other is made when the *linkage is designed*
  — May be beyond the control of the compiler writer

• Lexical levels for procedures and variables are determined in parsing
  — Requires the use of a lexically scoped symbol table

• Code to maintain access link or display is emitted at compile time
  — When compiler generates a procedure or a call, it emits the code
  — **Access links:** Lexical levels are known at compile time, so compiler can
    generate the code to find a level \( j \) display (*knows how far to run up the chain*)
  — **Display:** Lexical levels are known at compile time, so compiler can generate
    the save (in prolog) and restore (in epilog)

• Either an access link or a display is a runtime object
  — Exists at runtime,
  — Points to ARs, which are runtime objects
Other Runtime Services

We have talked about storage for ARs, for static & global data, for heap allocated data. What other runtime services are needed?

• Input & output subsystems, including files systems, sockets, streams, pipes, DMA, network connections, and the like
• Process management, including concurrency
• Interfaces to accelerators, such as GPUs, FPGAs, other processors
• Heap management, including allocation, deallocation, and collection
COMP 412, MidTerm Exam, Fall 2018

Max: 97.5
Min: 23
Mean: 66.7
Median: 73
Std Dev.: 19.5