Scope and Code Generation

Pat Morin
COMP 3002
Scope

- Scoping rules define how variable names are looked up when a program is run or compiled
- We have seen how to implement scoping rules in a typechecker
- How does it work in a code generator?
Run-time Environments

- During execution, each time a function call is made, a new *stack frame* is created to hold all the parameters and local variables for that function call.
- Local variables are assigned a position within their stack frame.
- A *frame pointer* (fp) keeps track of the top of the current stack frame.
Example of stack frame layout

```c
int factorial(int n) {
    if (n == 1) return 1;
    int t = factorial(n-1);
    return n * t;
}
```

n (4 bytes)  t (4 bytes)
### A Runtime Example

<table>
<thead>
<tr>
<th>n</th>
<th>5</th>
<th>t undefined</th>
<th>factorial(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>4</td>
<td>t undefined</td>
<td>factorial(4)</td>
</tr>
<tr>
<td>n</td>
<td>3</td>
<td>t undefined</td>
<td>factorial(3)</td>
</tr>
<tr>
<td>n</td>
<td>2</td>
<td>t 1</td>
<td>factorial(2)</td>
</tr>
</tbody>
</table>

fp
Discussion

- The compiler assigns, to each variable and parameter, a location within the current stack frame
- Operations on local variables are compiled into operations on memory locations relative to the frame pointer (fp)
- But now all variable references are to local variables
  - We assume *static lexical scoping*
int odd_factorial(int q) {
    int factorial(int n) {
        if (n == 1) return q;
        int t = factorial(n-1);
        return n * t;
    }
    if (q % t == 0)
        return t;
    return factorial(t);
}
## A Runtime Example

- How do we access q within factorial?

<table>
<thead>
<tr>
<th>q</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>5</td>
</tr>
<tr>
<td>t</td>
<td>undefined</td>
</tr>
</tbody>
</table>

| n  | 4 |
| t  | undefined |

| n  | 3 |
| t  | undefined |

| n  | 2 |
| t  | 1 |

| n  | 1 |
| t  | undefined |

even_factorial(5)

factorial(5)

factorial(4)

factorial(3)

factorial(2)

fp
Solution 1

• Each function has a static level of scope
  – Global scope - level 0
  – even_factorial – level 1
  – factorial - level 2

• Each stack frame contains an extra pointer fpp that points to the stack frame at the next highest level (fpp is actually an implicit parameter)
A Runtime Example

- Now we know how to find \( q \) from within any recursive call
  - \( q \) is at memory location \( \text{fpp} + 0 \)

```plaintext
q 3
fpp n 3 t undefined
fpp n 2 t 1
fpp n 1 t undefined

\text{even\_factorial}(3)
\text{factorial}(3)
\text{factorial}(2)
\text{factorial}(1)
```

```plaintext
\text{fp}
```
Solution 2

• The problem with solution 2 is that it becomes increasingly expensive to access elements that are further away in scope
  – Current level $i$
  – Variable to access is at level $j>i$
  – We must follow $j-i$ fpp pointers

• To speed this up, we can use a global array `frame_pointers`
  – `frame_pointers[i]` is the frame pointer to the currently active level $i$ frame
Frame pointer array example

```
Frame pointer array example

q 3
n 3 t undefined
n 2 t 1
n 1 t undefined

even_factorial(3)
factorial(3)
factorial(2)
factorial(1)
fp
```
Solution 2 (Cont'd)

- Within a function at level $i$
  - Save $\text{tmp} = \text{frame\_pointers}[i]$
  - Set $\text{frame\_pointer}[i] = \text{fp}$ (current frame pointer)
  - Before returning, restore $\text{frame\_pointers}[i] = \text{tmp}$

- When accessing a variable at level $i$ from a level $j > i$ we can get the correct frame pointer just by looking at $\text{frame\_pointers}[i]$
**Solution 1 versus Solution 2**

- Whether to use Solution 1 or 2 depends on how often variables at higher levels of scope are accessed
  - Solution 1 is more costly when accessing variables that are at much higher scope levels
  - Solution 2 increases the cost of every function call but makes all variable accesses constant time
What About Objects?

• For compilers, objects are just structures
• When calling a method on an object, an implicit pointer to the object is passed (*this* or *self*) to the method
• Inheritance is handled by having the child class inherit the structure of the parent and then add on its own elements
**Inheritance Example**

- Any method that assumes the memory layout of a Book can be used on a Novel or a Collection

<table>
<thead>
<tr>
<th>Book</th>
<th>Novel</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>title -&gt; String</td>
<td>author -&gt; String</td>
<td>editor -&gt; String</td>
</tr>
<tr>
<td>title (4 bytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>author (4 bytes)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>editor (4 bytes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**"Virtual" Methods**

- For each "virtual" object method, a new instance variable can be created
- When a child class overrides a method in a parent class, the instance variable is just overridden

<table>
<thead>
<tr>
<th>Book</th>
<th>Novel</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>title (4 bytes)</td>
<td>title (4 bytes)</td>
<td>title (4 bytes)</td>
</tr>
<tr>
<td>fnPrint -&gt; printBook</td>
<td>fnPrint -&gt; printNovel</td>
<td>fnPrint -&gt; printColl</td>
</tr>
<tr>
<td>author (4 bytes)</td>
<td></td>
<td>editor (4 bytes)</td>
</tr>
</tbody>
</table>
"Virtual" Methods (Cont'd)

• Virtual methods require two extra levels of indirection
  – Lookup the function address in this or self (1 level)
  – Load the function address and call it

• For this reason, some languages (C++ and Java) mix "virtual" and non-virtual functions
  – In C++ the virtual keyword is used to specify virtual functions (all others are non-virtual)
  – In Java, the final keyword is used to specify non-virtual functions (these can't be overridden by a subclass)
Summary

• A compiler must resolve occurrences of a variable to the memory location of that variable

• For static lexical scoping, this is done using parent frame pointers (fpp)
  – 2 solutions:
    • 1 - slower lookup for deeply nested functions
    • 2 - slower function calls but faster lookup

• For objects, this is even easier
  – Objects inherit their structure from their parents
  – "Virtual" functions are just instance variables