Lecture Outline

- Allocating temporaries in the Activation Record
  - Let’s optimize `cgen` a little
- Code generation for OO languages
  - Object memory layout
  - Dynamic dispatch
- Parameter passing mechanisms
  - call-by-value, call-by-reference, call-by-name

An Optimization: Allocate Space for Temporaries in the Activation Record (AR)

Topic I

Review

- The stack machine has activation records and intermediate results interleaved on the stack

```
An Optimization:
Allocate Space for Temporaries in the Activation Record (AR)

Topic I
```

```
cgen(e_1 + e_2) =
cgen(e_1) ; eval e_1
sw $a0 0($sp) ; save its value
addiu $sp $sp - 4 ; adjust $sp (!)
cgen(e_2) ; eval e_2
lw $t1 4($sp) ; get e_1
add $a0 $t1 $a0 ; $a0 = e_1 + e_2
addiu $sp $sp 4 ; adjust $sp (!)
```
Improved Code

<table>
<thead>
<tr>
<th>Old method</th>
<th>New idea</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgen(e1 + e2)</code> = <code>cgen(e1)</code> + <code>cgen(e2)</code></td>
<td><code>cgen(e1)</code> + <code>cgen(e2)</code></td>
</tr>
<tr>
<td><code>sw $a0 0($sp)</code></td>
<td><code>sw $a0 8($fp)</code></td>
</tr>
<tr>
<td><code>addiu $sp $sp - 4</code></td>
<td><code>addiu $sp $sp - 4</code></td>
</tr>
<tr>
<td><code>lw $t1 4($sp)</code></td>
<td><code>lw $t1 8($fp)</code></td>
</tr>
<tr>
<td><code>add $a0 $t1 $a0</code></td>
<td><code>add $a0 $t1 $a0</code></td>
</tr>
</tbody>
</table>

Example

```python
def add(x,y,z) =
x + (y + (z + w.f(3)))
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How many Stack Slots?

- Let $NS(e) = \#$ of slots needed to evaluate $e$
  - Includes slots for arguments to methods
- E.g: $NS(e1 + e2)$
  - Needs at least as many slots as $NS(e1)$
  - Needs at least one slot to hold value of $e1$, plus as many slots as $NS(e2)$, i.e., $1 + NS(e2)$
- Space used for temporaries in $e1$ can be reused for temporaries in $e2$

The Equations

- $NS(e1 + e2) = \max(NS(e1), 1 + NS(e2))$
- $NS(e1 - e2) = \max(NS(e1), 1 + NS(e2))$
- $NS(if e1 = e2 then e3 else e4) = \max(NS(e1), 1 + NS(e2), NS(e3), NS(e4))$
- $NS(f(e1, ..., en)) = \max(NS(e1), 1 + NS(e2), 2 + NS(e3), ..., (n-1) + NS(en), n)$
- $NS(int) = 0$
- $NS(id) = 0$

Rule for $f(e1, ..., en)$: Each time we evaluate an argument, we put it on the stack.

The Revised Activation Record

- For a function definition $f(x1, ..., xn) = e$ the AR has $2 + NS(e)$ elements
  - Return address
  - Frame pointer
  - $NS(e)$ locations for intermediate results
- Note that $f$'s arguments are now considered to be part of its caller's AR

Picture: Activation Record

```
\begin{itemize}
  \item popped by callee
  \item pushed by callee
  \item Increasing values of addresses
  \item direction of stack growth
\end{itemize}
```
Revised Code Generation

- Code generator must know how many slots are in use at each point.
- Add a new argument to code generator: the position of the next available slot.
  - The slots for temporary values are still used like a stack, but we predict usage at compile time.
  - This saves us from doing that work at run time.
  - Allocate all needed slots at the start of method.

Improved Code

<table>
<thead>
<tr>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cgen(e1 + e2) = cgen(e1) sw $a0 0($sp) addiu $sp $sp - 4 cgen(e2)</code></td>
<td><code>cgen(e1, ns) = cgen(e1, ns) sw $a0 ns($fp)</code></td>
</tr>
<tr>
<td><code>cgen(e1 + e2, ns) = cgen(e1 + e2, ns)</code></td>
<td><code>cgen(e2, ns - 4)</code></td>
</tr>
<tr>
<td><code>lw $t1 4($sp) add $a0 $t1 $a0 addiu $sp $sp 4</code></td>
<td><code>lw $t1 ns($fp) add $a0 $t1 $a0</code></td>
</tr>
</tbody>
</table>

OO code generation and memory layout

- How are objects represented in memory?
- How is dynamic dispatch implemented?

- OO Slogan: If C (child) is a subclass of P (parent), then an instance of class C can be used wherever an instance of class P is expected.
- This means that P’s methods should work with an instance of class C (code reuse).

Object Representation

```java
class P {
    x : Int <- 3;
    y : String <- "Hi";
    f() : Int { x };
    g() : String { y };
}
```

- Why method pointers?
- Why the tag?

Subclass Representation

```java
class P { .. (same) .. };

class C inherits P {
    w : Int <- 6; // new
    f() : Int (w); // override
    h() : Bool { z }; // new
}
```

- Idea: Append new fields

To call f:
```
lw $t1 12($s0)
jalr $t1
```
Subclasses (Cont.)

- The offset for an attribute is the same in an instance of a class and all of its subclasses.
- Any method for an A₁ can be used on a subclass A₂.
- Consider layout for A₁ < ... < A₃ < A₂ < A₁.

<table>
<thead>
<tr>
<th>Header</th>
<th>A₁ object</th>
<th>A₂ object</th>
<th>A₃ object</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁ attrs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₂ attrs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₃ attrs</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about multiple inheritance?

- Simple
  - Just append subclass fields
- Efficient
  - Code can ignore dynamic type -- just act as if it is the static type
- Supports overriding of methods
  - Just replace the appropriate dispatch pointers
- We implement type conformance (compile-time concept) with representation conformance (run-time concept)

An optimization: Dispatch Tables

Consider 3 instances of class C:

<table>
<thead>
<tr>
<th>C.f:</th>
<th>C.g:</th>
<th>C.h:</th>
</tr>
</thead>
</table>

Observation

- Every instance of a given class has the same values for all of its method pointers

  - Space optimization: Put all method pointers for a given class into a common table, called the "dispatch table"
    - Each instance has a pointer to the dispatch table

Picture with Dispatch Table

- Consider again 3 instances of C:

  1. tag: C
  2. dispPr
  3. x
  4. y
  5. z
  6. w
  7. f()
  8. g()
  9. h()

  - Objects are smaller
  - Dispatch is slower

Subclassing Again

- Call f:
  - lw $t1 1($s0)
  - lw $t1 0($t1)
  - jalr $t1

- C.f: return self [6]
- C.g: return self [2]
- C.h: return self [4]
Real COOL Object Layout

- Actually, the first 3 words of Cool objects contain header information:

```
<table>
<thead>
<tr>
<th>Offset (in bytes)</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class Tag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dispatch Ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attribute 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Needed for garbage collector

Parameter Passing Mechanisms

Topic III

- There are many semantic issues in programming languages centering on when values are computed, and the scopes of names
  - Evaluation is heart of computation
  - Names are most primitive abstraction mechanism

- We'll focus on parameter passing
  - When are arguments of function calls evaluated?
  - What are formal parameters bound to?

Call-by-value

- C uses call-by-value everywhere (except macros...)

```c
void callByValue(int y)
{
    y = y + 1;
    print(y);
}

void main()
{
    int x = 2;
    print(x);
    callByValue(x);
    print(x);
}
```

Output:

```
x = 2
y = 3
x = 2
```

x's value does not change when y's value is changed

Call-by-reference

- Available in C++ with the '&' type constructor

```c
void callByRef(int &y)
{
    output: x = 2
    y = y + 1;
    print(y);
}

void main()
{
    int x = 2;
    print(x);
    callByRef(x);
    print(x);
}
```

x's value changes when y's value is changed

Call-by-reference can be faked with pointers

- C++:

```c
void callByRef(int &y)
{
    output: x = 2
    y = y + 1;
    print(y);
}

void main()
{
    int x = 2;
    print(x);
    callByRef(x);
    print(x);
}
```

- C:

```c
void fakeCallByRef(int *y)
{
    output: x = 2
    *y = *y + 1;
    print(*y);
}

void main()
{
    int x = 2;
    print(x);
    fakeCallByRef(&x);
    print(x);
}
```

x's value does not change when y's value is changed

Must explicitly take the address of a local variable
Pointers to fake call-by-reference (cont.)

• It's not quite the same
  – A pointer can be reassigned to point at something else; a C++ reference cannot
• The pointer itself was passed by value
• This is how you pass structures in C

What about Java?

• Primitive types (int, boolean, etc.) are always passed by value
• Objects are not quite -by-value nor -by-reference:
  – If you reassign an object reference, the caller's argument does not get reassigned (like -by-value)
  – But if you modify the object referred-to, the caller will see that modification (like -by-reference)
• It's really ordinary call-by-value with pointers, but the pointers are not syntactically obvious. COOL is the same way.

Call-by-name

• Whole different ballgame: it's like passing the text of the argument expression, unevaluated
  – Also passes the environment, so free variables are still bound according to rules of static scoping
• The argument is not evaluated until it is actually used, inside the callee.
  – Might not get evaluated at all
• Used in some functional languages (e.g. Haskell)

Call-by-name example (in “C++-Extra”)

callByName(int closure y)
{
  print(y);
  
  // => print(x = x+1)
}
main()
{
  int x = 2;
  print(x);
  callByName( [x = x+1] );
  print(x);
}