Code Generation (II)

Adapted from Lectures by
Profs. Alex Aiken and George Necula (UCB)

Lecture Outline

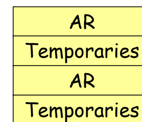
- Allocating temporaries in the Activation Record
  - Let's optimize cgen a little
- Code generation for OO languages
  - Object memory layout
  - Dynamic dispatch
- Parameter passing mechanisms
  - call-by-value, call-by-reference, call-by-name

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



These get put here when we evaluate compound exprs like $e_1 + e_2$ (when we need to store value of e_1 while evaluating e_2)

  - Advantage: Simple code generation
  - Disadvantage: Slow code
    - Storing/loading temporaries requires a store/load and **$sp** adjustment

cgen(e_1 + e_2) =

```
cgen(e1)                 ; eval e1
sw  $a0  0($sp)          ; save its value
addiu  $sp  $sp -  4     ; adjust $sp (!)
cgen(e2)                 ; eval e2
lw  $t1  4($sp)          ; get e1
add  $a0  $t1 $a0        ; $a0 = e1 + e2
addiu  $sp  $sp  4       ; adjust $sp (!)
```


An Optimization

- Idea: Predict how **$sp** will move at run time
  - Do this prediction at compile time
  - Move **$sp** to its limit, at the beginning

- The code generator must *statically* assign a location in the AR for each temporary

Improved Code

Old method

```
cgen(e1 + e2) =
   cgen(e1)
   sw  $a0  0($sp)
   addiu $sp $sp -  4
   cgen(e2)
   lw  $t1  4($sp)
   add $a0 $t1 $a0
   addiu $sp $sp  4
```

New idea

```
cgen(e1 + e2) =
   cgen(e1)
   sw $a0 ?($fp)

   cgen(e2)
   lw $t1 ?($fp)
   add $a0 $t1 $a0
```

statically allocate

Example

```
def   add(w,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: $NS(e_1 + e_2)$
  - Needs at least as many slots as $NS(e_1)$
  - Needs at least one slot to hold value of e_1, plus as many slots as $NS(e_2)$, i.e., $1 + NS(e_2)$
- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations

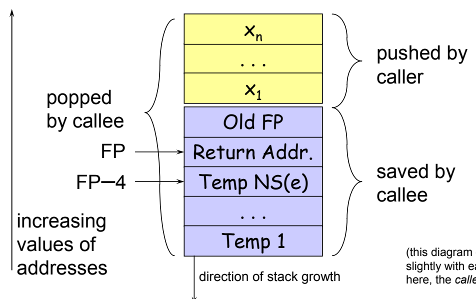
$NS(e_1 + e_2) = \max(NS(e_1), 1 + NS(e_2))$
$NS(e_1 - e_2) = \max(NS(e_1), 1 + NS(e_2))$
$NS(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$
$\max(NS(e_1), 1 + NS(e_2), NS(e_3), NS(e_4))$
$NS(f(e_1,\ldots,e_n)) =$
$\max(NS(e_1), 1 + NS(e_2), 2 + NS(e_3), \ldots, (n-1) + NS(e_n), n)$
$NS(int) = 0$
$NS(id) = 0$

Rule for $f(e_1, \ldots, e_n)$: Each time we evaluate an argument, we put it on the stack.

The Revised Activation Record

- For a function definition $f(x_1,\ldots,x_n) = e$ 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



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

Old method

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp - 4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

New method

```
cgen(e1 + e2, ns) =
    cgen(e1, ns)
    sw $a0 ns($fp)
    cgen(e2, ns + 4)
    lw $t1 ns($fp)
    add $a0 $t1 $a0
```

compile-time prediction

static allocation

Code Generation for OO Languages

Topic II

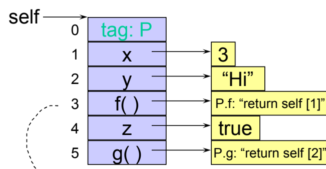
OO code generation and memory layout

- 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

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



dynamic dispatch

To call f:
lw $t1 12($s0)
jalr $t1

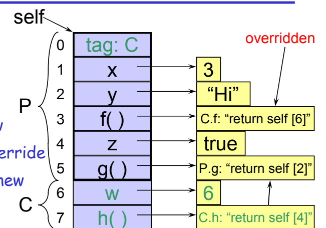
self

- Why method pointers?
- Why the tag? "case"

Subclass Representation

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

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

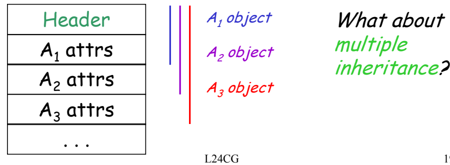


- Idea: Append new fields

To call f:
lw $t1 12($s0)
jalr $t1

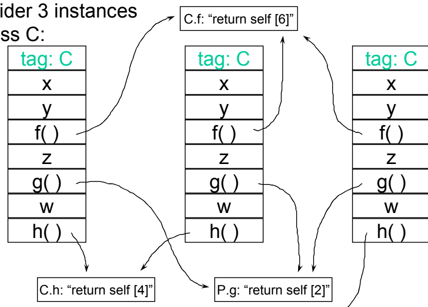
Subclasses (Cont.)

- The offset for an attribute is the same in an instance of a class and all of its subclasses
  - Any method for an A_1 can be used on a subclass A_2
- Consider layout for $A_n < \dots < A_3 < A_2 < A_1$



- 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

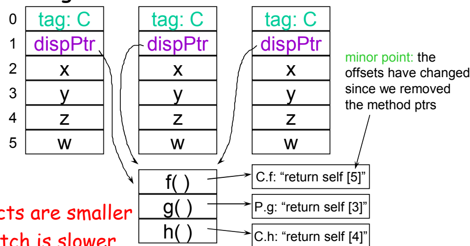


Observation

- 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

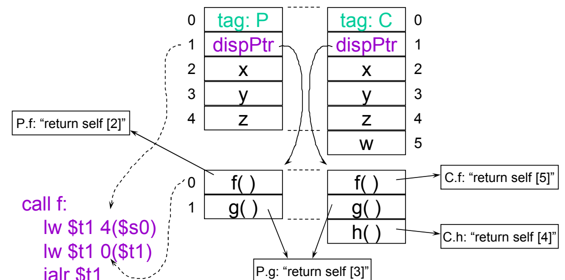
Picture with Dispatch Table

- Consider again 3 instances of C:



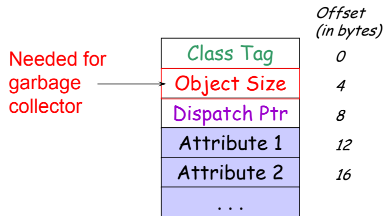
- Objects are smaller
- Dispatch is slower

Subclassing Again



Real COOL Object Layout

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



Parameter Passing Mechanisms

Topic III

Parameter Passing Mechanisms

- 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...)

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

```
callByRef(int &y)
{
   y = y + 1;
   print(y);
}
main()
{
   int x = 2;
   print(x);
   callByRef(x);
   print(x);
}
```

output:
x = 2
y = 3
x = 3

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

Call-by-reference can be faked with pointers

- C++:

```
callByRef(int &y)
{
   y = y + 1;
   print(y);
}
main()
{
   int x = 2;
   print(x);
   callByRef(x);
   print(x);
}
```

C:

```
fakeCallByRef(int *y)
{
   *y = *y + 1;
   print(*y);
}
main()
{
   int x = 2;
   print(x);
   fakeCallByRef(&x);
   print(x);
}
```

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")