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

$NS(e_1 + e_2) = \quad max(NS(e_1), 1 + NS(e_2))$

$NS(e_1 - e_2) = \quad max(NS(e_1), 1 + NS(e_2))$

$NS(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$

$\quad max(NS(e_1), 1 + NS(e_2), NS(e_3), NS(e_4))$

$NS(f(e_1,\ldots,e_n)) =$

$\quad max(NS(e_1), 1 + NS(e_2), 2 + NS(e_3), \ldots, (n-1) + NS(e_n), n)$

$NS(int) = \quad 0$

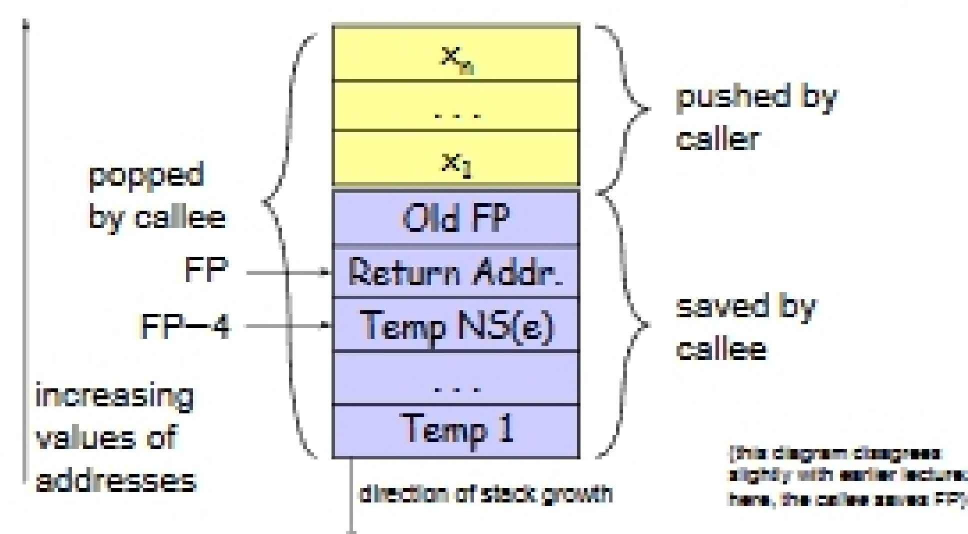
$NS(id) = \quad 0$

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)          <- compile-time prediction
   sw $a0 ns($fp)        static allocation
   cgen(e2, ns + 4)
   lw $t1 ns($fp)
   add $a0 $t1 $a0
```


Code Generation for OO Languages

Topic II

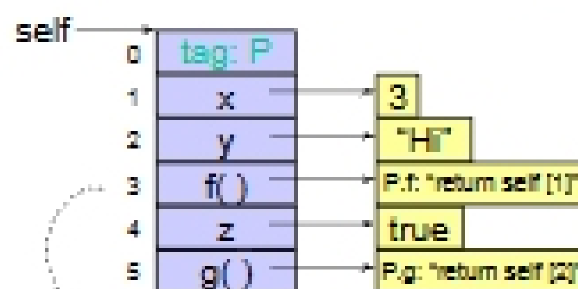
OO code generation and memory layout

- 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 };
};
```



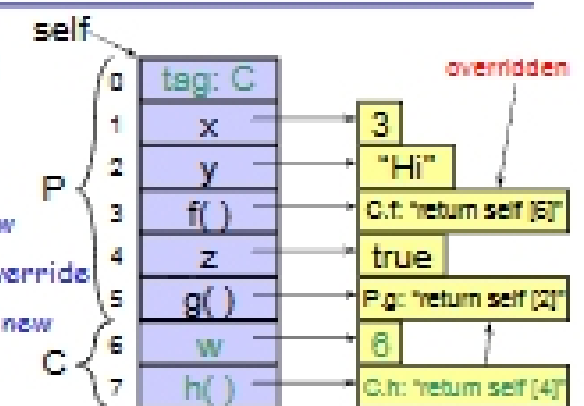
To call f: (self)
lw $t1 12($s0)
jalr $t1
dynamic dispatch

- 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