CS536 Lecture 21

Tuesday 21 April 2015

Last class:

- Code Generation (intro)

Today:

- Code generation (cont'd)

Code Generation (review)

Traverse AST

- Add code generation methods to AST nodes which directly output corresponding code to a file

Simplifying assumptions:

- All function arguments on the stack
- Use the stack for computation

Standard MIPS register usage conventions:

```
$v0, $v1
$a0, $a1, $a2, $a3
$t0 - $t9
$t1
$s0 - $s7
$sp, $fp, $ra
```

Code Generation for Global Variable Declarations

Source:

```
int iname;
struct MyStruct sname;
```

In `varDeclNode`, generate:

```
          .data
          .align 2 # align on word boundaries
_iname:   .space N #(N: size of the variable in bytes)
```

How do we know the size? Can be calculated during name analysis.

- Well-defined for scalars, e.g. int, bool (4 bytes).
- For structs, 4*size of the struct.

Simpler form for primitives:

```
          .data
          _iname: .word <value>
```

Code Generation for Functions

Need to generate:

- Preamble (like function signature)
- Prologue (set up the function)
- Body (Statements of the function)
- Epilogue

```
int f(int a, int b) {
    int c = a + b;
    int d = c - 7;
    return d;
}
```

Preamble:

```
    .text
    f:
    # … function body …
```

This label gives us a location to jump to:

```
    jal f
```

Code Generation for Functions: Prologue

Recall our view of the Activation Record:

- Save the return address
- Save the frame pointer (control link)
- Make space for locals
- Update the frame pointer

Translated as:

```
.text
f:
  sw   $ra 0($sp)    # return address
  subu $sp $sp 4     # (push)
  sw   $fp 0($sp)    # control link
  subu $sp $sp 4     # (push)
  subu $sp $sp 8     # locals
  addu $fp $sp 16    # update frame pointers
```

Code Generation for Functions: Epilogue

Recall how to restore the caller AR:

- Restore the return address
- Restore the frame pointer
- Restore the stack pointer
- Return control

Translated as:

```
.text
f:
  sw   $ra 0($sp)    # call link
  subu $sp $sp 4     # (push)
  sw   $fp 0($sp)    # control link
  subu $sp $sp 4     # (push)
  subu $sp $sp 8     # locals
  addu $fp $sp 16    # update frame pointers
  # … Function body …
  lw   $ra, 0($fp)
  move $t0, $fp
  lw   $fp, -4($fp)
  move $sp, $t0
  jr $ra
```

Code Generation for Functions: Returns

Returns:

```
.text
f:
  # … prologue … #
  # … body … #

  lw $v0, -12($fp)
  j f_exit

f_exit:
  # … epilogue … #
```

Type checkers will not catch situations where there is an execution path in a function that doesn't return the right type …

Code Generation for Function Body: Expressions

Goal:

Insight:

Example: `2*id`

Serialized pseudocode:

L1:
L2:
L3:
L4:
L5:
L6:

Serialized MIPS:

```
L1: li $t0 2
    sw $t0 0($sp)
    subu $sp $sp 4
L2: lw $t0 id
    sw $t0 0($sp)
    subu $sp $sp 4
L3: lw $t1 4($sp)
    addu $sp $sp 4
L4: lw $t0 4($sp)
    addu $sp $sp 4
L5: mult $t0 $t0 $t1
L6: sw $t0 0($sp)
    subu $sp $sp 4
```

Code Generation for Function Body: Statements

When we computed the expression above, the stack changed.

Algorithm:

1: Compute RHS expression on stack
2: Compute LHS *location* on stack
3: Pop LHS into $t1
4: Pop RHS into $t0
5: Store value $t0 at address $t1

Example: Generate MIPS code for `id = 1 + 2`

Code Generation: Dot Access

Struct members can be accessed using an offset (known statically) from the base of the struct.

```
struct1.a.b = struct2.a.b + 1;
```

Example:

```
void v(){
    struct Inner{
        bool hi;
        int there;
        int c;
    };

    struct Demo{
        struct Inner b;
        int val;
    };

    struct Demo inst;
    inst.b.c = inst.b.c + 1;
}
```

LHS:

```
subu $t0 $sp 16
sw $t0 0($sp)
```

RHS:

```
lw $t0 -16($sp)
sw $t0 0($sp)
subu $sp $sp 4
```

Code Generation: Control Flow Constructs

-
-
-

Function Call:

1: Put argument *values* on the stack (pass-by-value semantics)

2: Jump to the *callee* preamble label

May also need to save live registers (not for our stack machine though) and retrieve result value.

Example:

```
int f(int a, int b) {
    return 2;
}

int main() {
    int x;
    x = f(x, 4);
}
```

```
li  $t0 4        # push argument b
sw  $t0 0($sp)
subu$sp $sp 4
lw $t0 -8($fp)       # push argument a
sw $t0 0($sp)
subu $sp $sp 4
jal f                # goto f
addu $sp $sp 8       # tear down parameters
sw $v0 -8($fp)       # retrieve result
```

Code Generation: If-Then-Else statements

Generate label names for three branches of code: true, false, and successor.

Generate code for the condition check and make calls to the true and false labels (which haven't yet been placed).

Generate the true branch:

- Place the true label
- Write the body of the branch
- Jump to the successor label (not placed yet)

Generate the false branch (similar to above).

Place the successor label.

Example:

```
…
if (val == 1) {
    val = 2;
}
else {
    val = 3;
}
…
```

```
                lw   $t0  val      # eval cond LHS
                sw   $t0  0($sp)   # push onto stack
                subu $sp  $sp  4
                li   $t0  1        # eval cond RHS
                sw   $t0  0($sp)   # push onto stack
                subu $sp  $sp  4
                lw   $t1  4($sp)   # pop RHS into t1
                addu $sp  $sp  4
                lw   $t0  4($sp)   # pop LHS into t0
                addu $sp  $sp  4
                bne  $t0  $t1 L1   # branch if false
                li   $t0  2        # True branch
                sw   $t0  val
                j    L0            # end true branch
            L1:                    # False branch
                …
            L0:                    # Branch successor
```

Code Generation: While Loops

Similar to if-then-else:

Generate label names for head, successor.

At the end of the loop, jump back unconditionally to the head.

Example:

```
                      L0:
…                       lw   $t0  val      # eval cond LHS
while (val == 1) {      sw   $t0  0($sp)   # push onto stack
    val = 2;            subu $sp  $sp  4
}                       li   $t0  1        # eval cond RHS
                        sw   $t0  0($sp)   # push onto stack
                        subu $sp  $sp  4
                        lw   $t1  4($sp)   # pop RHS into t1
                        addu $sp  $sp  4
                        lw   $t0  4($sp)   # pop LHS into t0
                        addu $sp  $sp  4
                        bne  $t0  $t1 L1   # branch loop end
                        li   $t0  2        # Loop body
                        sw   $t0  val
                        j    L0            # jump to head
                    L1:                    # Loop successor
```