CS 536 Announcements for Wednesday, April 10, 2024

Last Time
- variable access at runtime
  - local vs global variables
  - static vs dynamic scopes

Today
- wrap up variable access at runtime
- start looking at details of MIPS
- code generation

Next Time
- continue code generation

Dynamic non-local scope

Example
function main() {
    int a = 0;
    fun1();
    fun2();
}
function fun2() {
    int a = 27;
    fun1();
}
function fun1() { a = a + 1; }

Key point – we don't know which non-local variable we are referring to (at compile-time)

Two ways to set up dynamic access
- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

Deep access
- if the variable isn't local
  - follow control link to caller's AR
  - check to see if it defines the variable
  - if not, follow the next control link down the stack
- note that we need to know if a variable is defined with that name in an AR
- usually means we'll have to associate a name with a stack slot

Shallow access
- keep a table with an entry for each variable declaration
- compile a direct reference to that entry
- at function call on entry to function F
  - F saves (in its AR) the current values of all variables that F declares itself
  - F restores these values when it finishes
Compiler Big Picture

Scanner → tokens

Parser → parse the AST

Static-Semantic Analysis → annotated AST, symbol table

IR Codegen

Optimizer

MC Codegen

Front End

(Unlike the front end) - can skip phases without sacrificing correctness

What phases do we do? How do we order the phases?
Compiler Back End: Design Decisions

When do we generate?
- directly from AST
- during SDT (e.g., during parsing)
  - still need to do name analysis & type checking
    - → extra pass(es)?

How many passes?
- fewer passes
  - faster compilation
  - less storage required
  - increases burden on programmer
- more passes
  - heavy weight
  - can lead to better modularity

What do we generate?
- machine code
  - much faster to generate
  - less engineering in the compiler
- intermediate representation (IR)
  - more amenable to optimization
  - more flexible output option
  - can reduce complexity of code generation

Possible IRs
- CFG (control-flow graph)
- 3AC (three-address code)
  - instruction set for a fictional machine
  - every operator has at most 3 operands
  - provides illusion of infinitely many registers
  - "flatten out" expressions

\[\text{Front end} \quad \downarrow \quad \text{IR codegen} \quad \uparrow \quad \text{optimize} \quad \downarrow \quad \text{mc codegen} \quad \uparrow \quad \text{optimize (?)}\]
3AC Example

3AC instruction set

Assignment
- \( x = y \) op \( z \)
- \( x = \) op \( y \)
- \( x = y \)

Indirection
- \( x = y[z] \)
- \( y[z] = x \)
- \( x = &y \)
- \( x = *y \)
- \( *y = x \)

Call/Return
- \( \text{param} \ x, k \)
- \( \text{retval} \ x \)
- \( \text{call} \ p \)
- \( \text{enter} \ p \)
- \( \text{return} \)
- \( \text{retrieve} \ x \)

Type Conversion
- \( x = \text{AtoB} \ y \)

Jumps
- \( \text{if} \ ( x \) op \( y \) \) goto \( L \)

Labeling
- \( \text{label} \ L \)

Basic Math
- \( \text{times, plus, etc.} \)

Example

source code

```c
if \( x + y \times z \) > \( x \times y + z \) \\
\[ a = 0. \] \\
\] \\
b = 2.
```

3AC code

```c
\text{tmp1} = y \times z \\
\text{tmp2} = x + \text{tmp1} \\
\text{tmp3} = x \times y \\
\text{tmp4} = \text{tmp3} + z \\
\text{if} ( \text{tmp2} \leq \text{tmp4} ) \text{ goto } L \\
\quad a = 0 \\
L: \ b = 2
```

3AC representation

- each instruction represented using a structure called a “quad”
  - space for the operator
  - space for each operand
  - pointer to auxiliary info (label, succesor quad, etc.)
- chain of quads sent to an architecture-specific machine-code-generation phase

Code Generation

For base

- skip building a separate IR
- generate code by traversing the AST
  - add codeGen methods to AST nodes
  - directly emit corresponding code into file

Two high-level goals

- generate correct code (hard to achieve both at same time)
- generate efficient code
Code Generation (cont.)

Simplified strategy

Make sure we don't have to worry about running out of registers

- for each operation, put all arguments on the stack
- make use of the stack for computation
- only use two registers for computation \( \text{ST0, ST1} \)

Different AST nodes have different responsibilities

Many nodes simply "direct traffic"

- `ProgramNode.codeGen` - call codeGen on its child
- List-node types - call codeGen on each element in the list
- `DeclNode` - what code to generate depends on context - global or local
  - Program
  - DeclList
  - \( \rightarrow \) VarDecl \( \rightarrow \) TupleDecl \( \rightarrow \) FctnDecl
  - implement codeGen for global vars

Code Generation for Global Variable Declarations

Source code:

```
integer name.
tuple MyTuple instance.
```

In AST: `VarDeclNode`

Generate:

```
.data
.align 2   # align on word boundaries
\_\_name: .space \text{N}   # \text{N} is the size of variable (in bytes)
```

Size of variable

- we can calculate this during name analysis
- for scalars, well-defined: integer, boolean are 4 bytes
- for tuples: \(4 \times \text{size of tuples}\)

For string literals: \(\text{\_str1: .asciiz "this is a string"}\)
Code Generation for Function Declarations

Need to generate

- preamble - like a function signature
- prologue - sets up function's AR
- body - generate code for function's statements
- epilogue - tear down AR

MIPS Crash Course

Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
</tr>
<tr>
<td>$v0</td>
<td>used for system calls and to return int values from function calls, including the syscall that reads an int</td>
</tr>
<tr>
<td>$f0</td>
<td>used to return double values from function calls, including the syscall that reads a double</td>
</tr>
<tr>
<td>$a0</td>
<td>used for output of int and string values</td>
</tr>
<tr>
<td>$f12</td>
<td>used for output of double values</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>temporaries for ints</td>
</tr>
<tr>
<td>$f0 - $f30</td>
<td>registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)</td>
</tr>
</tbody>
</table>

Also has $s0 and $s1, special purpose registers for multiplication & division
MIPS Crash Course (cont.)

Program structure

Data
- label: `.data`
- variable names & size; heap storage

Code
- label: `.text`
- program instructions
- starting location: `main`

Data

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1:</td>
<td>.word</td>
<td>10</td>
</tr>
<tr>
<td>a1:</td>
<td>.byte</td>
<td>'a', 'b'</td>
</tr>
<tr>
<td>a2:</td>
<td>.space</td>
<td>40</td>
</tr>
</tbody>
</table>

Memory instructions

- `lw register_destination, RAM_source`
  - copy word (4 bytes) at source RAM location to destination register.

- `lb register_destination, RAM_source`
  - copy byte at source RAM location to low-order byte of destination register.

- `li register_destination, value`
  - load immediate value into destination register

- `sw register_source, RAM_dest`
  - store word in source register into RAM destination

- `sb register_source, RAM_dest`
  - store byte in source register into RAM destination
MIPS Crash Course (cont.)

Arithmetic instructions

- **add** $t0,$t1,$t2  \( \rightarrow \) add/sub of signed (2's complement) integers
- **sub** $t2,$t3,$t4  \( \leftarrow \) add immediate
- **addi** $t2,$t3, 5  \( \leftarrow \) add immediate
- **addu** $t1,$t6,$t7  \( \rightarrow \) add/sub of unsigned integers
- **subu** $t1,$t6,$t7
- **mult** $t3,$t4  \( \leftarrow \) result is in $t0
- **div** $t5,$t6  \( \leftarrow \) result is in $t0 and remainder is in $t1
- **mfhi** $t0  \( \rightarrow \) move from $t1 to $t0
- **mflo** $t1  \( \rightarrow \) move from $t0 to $t1

Control instructions

- **b** target  \( \rightarrow \) unconditional branch to target
- **beq** $t0,$t1,target
- **blt** $t0,$t1,target
- **ble** $t0,$t1,target
- **bgt** $t0,$t1,target
- **bge** $t0,$t1,target
- **bne** $t0,$t1,target
- **j** target  \( \rightarrow \) unconditional jump to target
- **jr** $t3  \( \rightarrow \) indirect jump - jump to address in $t3
- **jal** sub_label  \# "jump and link"
  \( \rightarrow \) jump to sub-label & store return location in $ra

Check out: MIPS tutorial

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html