Compiler construction 2002 – lecture 8

Koen Langendoen
Delft University of Technology
The Netherlands

Overview
- intermediate code
- interpretation
- code generation
  - code selection
  - register allocation
  - instruction ordering

Intermediate code
- language independent
  - no structured types, only basic types (char, int, float)
  - no structured control flow, only (un)conditional jumps
- linear format
  - Java byte code

Interpretation
- recursive interpretation
  - operates directly on the AST [attribute grammar]
  - simple to write
  - thorough error checks
  - very slow: 1000x speed of compiled code
- iterative interpretation
  - operates on intermediate code
  - good error checking
  - slow: 100x

Recursive interpretation
- function per node type
  - implement semantics
  - visit children
- status indicator
  - normal mode
  - return mode (value)
  - jump mode (label)
  - exception mode (name)

Recursive interpretation
PROCEDURE Elaborate if statement (If node):
  SET Result TO Evaluate condition (If node .condition);
  IF Status .mode /= Normal mode: RETURN;
  IF Result .type /= Boolean:
    ERROR “Condition in if-statement is not of type Boolean”;
    RETURN;
  IF Result .boolean .value = True:
    Elaborate statement (If node .then part);
  ELSE
    // Is there an else-part at all?
    IF If node .else part /= No node:
      Elaborate statement (If node .else part);
**Self-identifying data**

- must handle user-defined data types
- value = pointer to type descriptor + array of subvalues
- example: complex number
  
<table>
<thead>
<tr>
<th>re</th>
<th>im</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

**Complex numbers**

<table>
<thead>
<tr>
<th>name:</th>
<th>value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>complex_number</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name:</th>
<th>value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>re</td>
<td>3.0</td>
</tr>
<tr>
<td>im</td>
<td>4.0</td>
</tr>
</tbody>
</table>

**Iterative interpretation**

- operates on threaded AST
- active node pointer (similar to PC)
- flat loop over a case statement

```plaintext
WHILE Active node .type /= End of program type:
SELECT Active node .type:
  CASE ... IF condition THEN ELSE
  // We arrive here after the condition has been evaluated;
  // the Boolean result is on the working stack.
  SET Value TO Pop working stack ();
  IF Value .boolean .value = True:
    SET Active node TO Active node .true successor;
  ELSE Value .boolean .value = False:
    IF Active node .false successor /= No node:
      SET Active node TO Active node .false successor;
    ELSE Active node .false successor = No node:
      SET Active node TO Active node .successor;
  CASE ...
```

**Iterative interpretation**

- data structures implemented as arrays
- global data of the source program
- stack for storing local variables
- shadow memory to store properties
  - status: (un)initialized data
  - access: read-only / read-write data
  - type: data / code pointer / ...

**Code generation**

- tree rewriting
  - replace nodes and subtrees of the AST by target code segments
  - produce a linear sequence of instructions from the rewritten AST
- example
  - code: \( p := p + 5 \)
  - target: RISC machine
Register machine instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
<th>Tree pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load_Const</td>
<td>Rn := c</td>
<td>Rn := c</td>
</tr>
<tr>
<td>Load_Mem</td>
<td>x, Rn</td>
<td>x := Rn</td>
</tr>
<tr>
<td>Store_Mem</td>
<td>Rn, x</td>
<td>x := Rn</td>
</tr>
<tr>
<td>Add_Reg</td>
<td>Rm, Rn</td>
<td>Rn := Rn + Rm</td>
</tr>
<tr>
<td>Sub_Reg</td>
<td>Rm, Rn</td>
<td>Rn := Rn - Rm</td>
</tr>
<tr>
<td>Mul_Reg</td>
<td>Rm, Rn</td>
<td>Rn := Rn * Rm</td>
</tr>
</tbody>
</table>

Tree rewriting for \( p := p + 5 \)

- linearize instructions:
  - depth-first traversal

Code generation

main issues:
- code selection – which template?
- register allocation – too few!
- instruction ordering

optimal code generation is NP-complete
- consider small parts of the AST
- simplify target machine
- use conventions

Simple code generation

- consider one AST node at a time
- two simplistic target machines
  - pure register machine
  - pure stack machine

Simple code generation for a stack machine

- example: \( b*b - 4*a*c \)
- threaded AST
Simple code generation for a stack machine

- example: \( b^2 - 4ac \)
- rewritten AST

Depth-first code generation for a stack machine

**PROCEDURE** Generate code (Node):

<table>
<thead>
<tr>
<th>SELECT Node.type:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE Constant type:</td>
<td>Emit (“Push_Const” Node.value);</td>
</tr>
<tr>
<td>CASE LocalVar type:</td>
<td>Emit (“Push_Local” Node.number);</td>
</tr>
<tr>
<td>CASE StoreLocal type:</td>
<td>Emit (“Store_Local” Node.number);</td>
</tr>
<tr>
<td>CASE Add type:</td>
<td>Generate code (Node.left); Generate code (Node.right); Emit (“Add_Top2”);</td>
</tr>
<tr>
<td>CASE Subtract type:</td>
<td>Generate code (Node.left); Generate code (Node.right); Emit (“Sub_Top2”);</td>
</tr>
<tr>
<td>CASE Multiply type:</td>
<td></td>
</tr>
</tbody>
</table>
Weighted register allocation

• registers are scarce, depth-first traversal is not optimal

2

2 registers

3

3 registers

• evaluate heaviest subtree first

Load_Mem b, R1
Load_Mem b, R2
Mul_Reg R2, R1
Load_Const 4, R2
Load_Mem a, R3
Load_Mem b, R4
Mul_Reg R4, R3
Mul_Reg R3, R2
Sub_Reg R2, R1

Register requirements of a subtree

FUNCTION Weight of (Node) RETURNING an integer:
SELECT Node .type:
CASE Constant type: RETURN 1;
CASE Variable type: RETURN 1;
CASE ...;
CASE Add type:
SET Required left TO Weight of (Node .left);
SET Required right TO Weight of (Node .right);
IF Required left > Required right: RETURN Required left;
IF Required left < Required right: RETURN Required right;
// Required left = Required right
RETURN Required left + 1;
CASE ...

Exercise (5 min.)

• compute the minimal number of registers needed to evaluate the expression

b*b – 4*a*c

Answers

Register spilling

too few registers?

• spill registers in memory, to be retrieved later
• heuristic: select subtree that uses all registers, and replace it by a temporary

example:

• b*b – 4*a*c
• 2 registers

Register spilling

Load_Mem b, R1
Load_Mem b, R2
Mul_Reg R2, R1
Store_Mem R1, tmp
Load_Mem a, R1
Load_Mem b, R2
Mul_Reg R2, R1
Load_Const 4, R2
Mul_Reg R1, R2
Load_Mem tmp, R1
Sub_Reg R2, R1

Load_Mem b, R1
Load_Mem b, R2
Mul_Reg R2, R1
Store_Mem R1, tmp
Load_Mem a, R1
Load_Mem b, R2
Mul_Reg R2, R1
Load_Const 4, R2
Mul_Reg R1, R2
Load_Mem tmp, R1
Sub_Reg R2, R1
Summary

- interpretation
  - recursive
  - iterative
- simple code generation
  - code per AST node
  - stack and register machines
  - weighted register allocation
  - register spilling

Homework

- study sections:
  - 4.2.1 – 4.2.3  from interpreter to compiler
- assignment 1:
  - replace yacc with LLgen
  - new deadline April 16 08:59
- print handout for next week [blackboard]