Interpreters and compilers

**Operational Semantics 3**  
**Compilation**

Jim Royer  
CIS 352  
February 26, 2015

There are many variations on the above.

**Problem 1: Compile Aexp to a stack-based VM**

Aexp

\[ v \in \text{Num} \ (\text{Numeric Values}) \quad a \in \text{Aexp} \ (\text{Arithmetic expressions}) \]

\[ a ::= \ v \mid (a_1 + a_2) \mid (a_1 - a_2) \mid (a_1 * a_2) \]

A big-step semantics for Aexp

\[ \text{Num: } v \Downarrow v \]

\[ \text{Eval-\(\oplus\): } a_1 \Downarrow v_1 \quad a_2 \Downarrow v_2 \quad (v = v_1 \oplus v_2) \]

where \( \oplus = +, -, \) and \(*\).

What is our target VM?

**Our target VM, 1**

Memory banks

- 256 many 8 bit words
- so 8-bit addresses and 8-bit contents

used to store the stack, object code, and (later) registers.

Registers (internal)

- PC = program counter (points to the current instruction)
- SP = stack pointer (points to the top of the stack + 1)

Arithmetic

- mod 256 many 8 bit words
- So 255+1 = 0. (IMPORTANT!!!!)

**Interpreters and compilers**

**compiler**

source code \(\xrightarrow{\text{via lexer}}\) abstract syntax \(\xrightarrow{\text{via parser}}\) object code

\(\xrightarrow{\text{via compiler}}\) object code

\(\xrightarrow{\text{via linker}}\) executable

\(\xrightarrow{\text{via hardware}}\) value

**interpreter**

source code \(\xrightarrow{\text{via lexer}}\) abstract syntax \(\xrightarrow{\text{via parser}}\) value

\(\xrightarrow{\text{via evaluator/interpreter}}\) value
What do the instructions do?

To precisely nail this down, we define a transition system given by a small-step operational semantics:

\[(pc, sp, stk) \Rightarrow (pc', sp', stk')\]

where:
- \(obj\) = the object code (\(\approx\) an array)
- \(stk\) = the stack (\(\approx\) an array)
- \(pc\) = the program counter (\(\approx\) an index into \(obj\))
- \(sp\) = the stack pointer (\(\approx\) an index into \(stk\))

Rule format

\[
\text{name: ... premises ...} \quad \text{obj} \vdash (pc, sp, stk) \Rightarrow (pc', sp', stk') \quad \text{(side conditions)}
\]

Push:

\[
\frac{\text{obj} \vdash (pc, sp, stk)}{(pc + 2, sp + 1, stk[sp] \mapsto n)} \quad \text{(obj}[pc] = \text{push}, \text{obj}[pc + 1] = n)
\]

Pop:

\[
\frac{\text{obj} \vdash (pc, sp, stk)}{(pc + 1, sp - 1, stk)} \quad \text{(obj}[pc] = \text{pop})
\]

Add:

\[
\frac{\text{obj} \vdash (pc, sp, stk)}{(pc + 1, sp - 2 \mapsto n)} \quad (*)
\]

\[
(*) \quad \text{obj}[pc] = \text{add} \quad \text{and} \quad n = stk[sp - 2] + stk[sp - 1]
\]

N.B. Since pointer arithmetic is mod 256, underflow and overflow are wrap-arounds.

Questions

- Is the translation well-behaved?
  (In what condition does each expression leave the stack?)
- Is the translation correct?
  (No, we could easily overflow the stack.)
  (Yes, if we stay within size bounds. How to prove this?)

Proposition

Suppose
- \(a\) is an \(Aexp\) expression
- \(I_a\) is the sequence of instructions the compiler generates for \(a\)
- \(I_a\) is loaded into the code bank from address \(\ell_0\) to address \(\ell_1\).

Then \((\ell_0, sp, stk) \Rightarrow^* (\ell_1 + 1, sp + 1, stk[sp] \mapsto v)\), where \(a \downarrow v\), provided there is no stack overflow or underflow.

Proof: By an easy structural induction on \(a\).
Problem 2: Compile LC to a stack-based VM

LC Syntax and Base Types

\( P \ ::= \ C \mid E \mid B \)

\( C \ ::= \) skip \( \mid \ell : = E \mid C \mid \text{if} \ B \ \text{then} \ C \ \text{else} \ C \mid \text{while} \ B \ \text{do} \ C \)

\( E \ ::= \ n \mid \ell \mid E \ast E \ (\ast \in \{+,-,\times,\ldots\}) \)

\( B \ ::= \ b \mid E \ast E \ (\ast \in \{=,<,\geq,\ldots\}) \)

What is our target VM?

Our target VM, 1

- memory banks
  - 256 many 8 bit words
  - so 8-bit addresses and 8-bit contents
- used to store the stack, object code, and user registers.

Registers (internal)

- \( pc \) = program counter (points to the current instruction)
- \( sp \) = stack pointer (points to the top of the stack + 1)

Registers (user)

- 256-many 8-bit registers
- Named \( \ell_0 \) through \( \ell_{255} \) (or alternatively, 0 through 255)

Our target VM, 2

What do the instructions do?

Transition system on VM configs: \((pc, sp, stk, regs)\).

- \( obj \) = the object code
- \( stk \) = the stack
- \( regs \) = the user registers
- \( pc \) = the program counter
- \( sp \) = the stack pointer

Rule format

\( \text{name: } obj \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp', stk', regs') \)

\( * \) = side-conditions

Our target VM, 3

- Fetch:
  \[ obj \vdash (pc, sp, stk, regs) \Rightarrow (pc + 2, sp + 1, stk[sp \mapsto v], regs) \]

- Store:
  \[ obj \vdash (pc, sp, stk, regs) \Rightarrow (pc + 2, sp, stk[regs[n] \mapsto v]) \]

\( * \) = \( obj[pc] = \text{fetch} \), \( obj[pc + 1] = n \), \( \text{regs}[n] = v \)

\( ** \) = \( obj[pc] = \text{store} \), \( obj[pc + 1] = n \), \( stk[sp - 1] = v \)

N.B. Store does not pop the stack!!!
Our target VM, 3

Before: Fetch n

After: Fetch n

Our target VM, 4

Ilit: \[ \text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc + 1, sp - 1, stk[(sp - 2) \mapsto v], regs) \] \(^(*)\)

Jmp: \[ \text{obj} \vdash (pc, sp, stk, regs) \Rightarrow (pc', sp, stk, regs) \] \(^(**)\)

\(^(*)\) \text{obj}[pc] = \text{ilt}, \text{stk}[sp - 2] = v_1, \text{stk}[sp - 1] = v_2, \text{and if} v_1 < v_2 \text{then} v = 1 \text{else} v = 0

\(^(**)\) \text{obj}[pc] = \text{jmp}, \text{obj}[pc + 1] = n, pc' = pc + n + 1

N.B. \text{jmp} is a relative jump.

[Draw before/after pictures]
Our target VM, 5

\[
\text{Jz: } \quad \overline{obj, (pc, sp, stk, regs)} \rightarrow (pc', sp - 1, stk, regs) \quad (*)
\]

\[
\text{Jnz: } \quad \overline{obj, (pc, sp, stk, regs)} \rightarrow (pc', sp - 1, stk, regs) \quad (**) \quad \overline{obj, pc} = jz, \quad \overline{obj, pc + 1} = n, \quad \overline{stk, sp - 1} = v, \quad \text{and} \quad \overline{stk, sp} = v, \quad \text{and}
\]

if \(v = 0\) then \(pc' = pc + n + 1\)
else \(pc' = pc + 2\).

N.B. Both Jz and Jnz pop the stack!!!
N.B. Both Jz and Jnz are relative jumps.

[Draw before/after pictures]
Compiling integer and boolean expressions

- The compiler for integer expressions is a repeat of what we had before.
- For boolean expressions we maintain the convention that a boolean value is represented by either 0 (for False) or 1 (for True).

<table>
<thead>
<tr>
<th>Bvaltrans:</th>
<th>$b \Rightarrow \text{Push } v$ (v = 1 when $b = \text{True}$, v = 0 when $b = \text{False}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lttrans:</td>
<td>$ae_1 \Rightarrow I_1$  $ae_2 \Rightarrow I_2$  $(ae_1 &lt; ae_2) \Rightarrow I_1 + I_2 + [\text{Ilt}]$</td>
</tr>
<tr>
<td>Eqtrans:</td>
<td>$a_1 \Rightarrow I_1$  $a_2 \Rightarrow I_2$  $(ae_1 == ae_2) \Rightarrow I_1 + I_2 + [\text{Isub}, \text{Push } 1, \text{Ilt}]$</td>
</tr>
</tbody>
</table>

Compiling statements, 1

- **Skiptrans**:  \( \overline{\text{Skip}} \Rightarrow \emptyset \)
- **Assigntrans**:  \( ae \Rightarrow I_0 \)  \( \ell_i := ae \Rightarrow I_0 + \text{[Store } i, \text{Pop]} \)
- **Seqtrans**:  \( S_1 \Rightarrow I_1 \)  \( S_2 \Rightarrow I_2 \)  \( S_1; S_2 \Rightarrow I_1 + I_2 \)

Compiling statements, 2

- **Iftrans**:  \( be \Rightarrow I_0 \)  \( S_1 \Rightarrow I_1 \)  \( S_2 \Rightarrow I_2 \)
  \( \text{if } be \text{ then } S_1 \text{ else } S_2 \Rightarrow I_0 + I_0 + [\text{Jz } n_0] + + I_1 + + I_2 + [\text{Jmp } n_1] + + I_2 \)  \( \text{(*)} \)

  \( (*) \)  \( n_0 = 3 + \text{codeLen}(I_1) \)  and  \( n_1 = 1 + \text{codeLen}(I_2) \)

- **Whiletrans**:  \( be \Rightarrow I_0 \)  \( S \Rightarrow I_1 \)
  \( \text{while } be \text{ do } S \Rightarrow I_0 + + [\text{Jz } n_0] + + I_1 + + [\text{Jmp } n_1] \)  \( \text{(*)} \)

  \( (*) \)  \( n_0 = 3 + \text{codeLen}(I_1) \)  and  \( n_1 = -(3 + \text{codeLen}(I_0) + \text{codeLen}(I_1)) \)

Questions

- What does it mean for the compiler to be correct?
  Any run of a compiled program ends up with the state (register contents) dictated by the operational semantics of LC.
- How does one prove that?
  Another structural induction on LC code.
- Does compiled code behave well (e.g., always leaves the stack in some sensible condition)?
- What about variables, blocks, procedures, etc.?

Implementation in LCvm.hs and LCCompiler.hs.