Operational Semantics 3
Compilation

Jim Royer
CIS 352
February 27, 2018

Interpreters and compilers

<table>
<thead>
<tr>
<th>interpreter</th>
<th>source code via lexer and parser</th>
<th>abstract syntax via evaluator/interpreter</th>
<th>value</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>compiler</th>
<th>source code via lexer and parser</th>
<th>abstract syntax via compiler</th>
<th>object code via linker</th>
<th>executable via hardware</th>
<th>value</th>
</tr>
</thead>
</table>

There are many variations on the above.

Problem 1: Compile Aexp to a stack-based VM

Aexp

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

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

A big-step semantics for Aexp

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

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

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!!!!)
Our target VM, 2

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 (≈ an array)
- \(stk\) = the stack (≈ an array)
- \(pc\) = the program counter (≈ an index into \(obj\))
- \(sp\) = the stack pointer (≈ an index into \(stk\))

Rule format

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

Push:
\[
obj \vdash (pc, sp, stk) \Rightarrow (pc + 2, sp + 1, stk[sp] \rightarrow n) \quad (obj[pc] = \text{push}, \ obj[pc + 1] = n)
\]

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

Add:
\[
obj \vdash (pc, sp, stk) \Rightarrow (pc + 1, sp - 1, stk[sp - 2] \rightarrow n) \quad (*)
\]

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

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

Evaluation/Compilation rules for Aexp

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num:</td>
<td>(v \downarrow v)</td>
</tr>
<tr>
<td>Num_{trans}:</td>
<td>(v \downarrow \text{Push } v)</td>
</tr>
<tr>
<td>Plus:</td>
<td>(a_1 \downarrow v_1, a_2 \downarrow v_2, (v = v_1 + v_2))</td>
</tr>
<tr>
<td>Plus_{trans}:</td>
<td>(a_1 \downarrow I_1, a_2 \downarrow I_2, (a_1 + a_2) \downarrow I_1 + I_2 + [\text{Add}])</td>
</tr>
</tbody>
</table>

Haskell implementation in \texttt{vm0.hs}.

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 \(((\ell_0, sp, stk) \Rightarrow^* (\ell_1 + 1, sp + 1, stk[sp] \rightarrow v)), \text{ where } a \downarrow v,\)

\textit{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
(The Δ’s mark changes.)

Phases

\[ P ::= C \mid E \mid B \]

Commands

\[ C ::= \text{skip} \mid \ell : = E \mid C ; C \mid \text{if } B \text{ then } C \text{ else } C \mid \text{while } B \text{ do } C \]

Integer Expressions

\[ E ::= n \mid !\ell \mid E \oplus E (\oplus \in \{+, -, \times, \ldots \}) \]

Boolean Expressions

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

8-Bit Integers

\[ n \in \mathbb{Z}_{256} = \{0, 1, \ldots, 255\} \quad (\Delta) \]

Booleans

\[ b \in B = \{\text{true, false}\} \]

Locations

\[ \ell \in L = \{\ell_0, \ell_1, \ldots, \ell_{255}\} \quad (\Delta) \]

\[ !\ell \equiv \text{the integer currently stored in } \ell \]

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

instructions

- Halt
- Push n
- Pop
- Fetch n
- Store n
- Iadd
- Isub
- Imult
- Ilt
- Jmp
- Jz
- Jnz

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

\[
\text{name:} \quad \ldots \text{premises} \ldots \quad \frac{\text{obj} \vdash (pc, sp, stk, regs)}{(pc', sp', stk', regs') (\ast)}
\]

\[ (\ast) = \text{side-conditions} \]

The VM in a picture

The VM just before executing a Fetch

PC \[ a \]

Obj \[ b \]

Regs \[ : n : v : \]

Stack \[ : \]

SP \[ a \]

\[ \text{Fetch} \]

\[ a+1 \]

\[ n \]

\[ b \]

\[ ? \]

\[ a+2 \]

\[ b+1 \]

??
Our target VM, 3

**Fetch:**

\[
obj \vdash (pc, sp, stk, regs) \Rightarrow (pc + 2, sp + 1, stk[sp \mapsto v], \text{regs}) \quad (\ast)
\]

**Store:**

\[
obj \vdash (pc, sp, stk, regs) \Rightarrow (pc + 2, sp, stk, \text{regs}[n \mapsto v]) \quad (\ast\ast)
\]

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

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

**N.B.** Store does **not** pop the stack!!!
Ilt: Before and After

Before: Ilt

After: Ilt

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

Jmp: Before and after

Before: Jmp n

After: Jmp n

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

N.B. Jmp is a relative jump.

Our target VM, 5

Jz: Before and After

Before: Jz n

After: Jz n

(* ) \( \text{obj}[pc] = \text{jz}, \text{ obj}[pc + 1] = n, \text{ stk}[sp - 1] = v, \) 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 a relative jumps.
Jnz: Before and After

Before: Jnz n

After: Jnz n

Compiling integer and boolean expressions

- Compiling integer expressions: A repeat of what we had before.
- Compiling boolean expressions: Maintain the convention that a boolean value is represented by either 0 (for False) or 1 (for True).

Compiling statements, 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skip</td>
<td>Skip ⇒ []</td>
</tr>
<tr>
<td>Assign</td>
<td>( \ell : = ae ) ⇒ ( I_0 ++ [Store i, Pop] )</td>
</tr>
<tr>
<td>Seq</td>
<td>( S_1 \Rightarrow I_1 \quad S_2 \Rightarrow I_2 ) \quad ( S_1; S_2 \Rightarrow I_1 + I_2 )</td>
</tr>
</tbody>
</table>

Compiling statements, 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>If</td>
<td>( be \Rightarrow I_0 \quad S_1 \Rightarrow I_1 \quad S_2 \Rightarrow I_2 ) \quad ( \text{if } be \text{ then } S_1 \text{ else } S_2 \Rightarrow I_0 ++ [Jz \ n_0] ++ I_1 ++ [Jmp \ n_1] ++ I_2 ) \quad (*)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>While</td>
<td>( be \Rightarrow I_0 \quad S \Rightarrow I_1 ) \quad ( \text{while } be \text{ do } S \Rightarrow I_0 ++ [Jz \ n_0] ++ I_1 ++ [Jmp \ n_1] ) \quad (*)</td>
</tr>
</tbody>
</table>

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

Implementation in \texttt{LCvm.hs} and \texttt{LCCompiler.hs}.
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.?