Operational Semantics of Cool

Lecture 23

Dr. Sean Peisert – ECS 142 – Spring 2009

Wednesday, May 20, 2009
Status

• Project 3 due on Friday
• Project 4 assigned on Friday. Due June 5, 11:55pm
• Office Hours this week Fri at 11am
• No Class (Memorial Day) on Monday, May 25
• Read Sec. 8.1 through 8.8 by Wednesday, May 27
• Final Exam Review 9a-11a on June 5 in Olson 105
Assembly Language Description of Semantics

- Assembly-language descriptions of language implementation have too many irrelevant details
- Whether to use a stack machine or not
- Which way the stack grows
- How integers are represented on a particular machine
- We need a complete specification that is architecture and OS independent and doesn’t overly restrict implementation choices.
Intro to Operational Semantics

• Once again we introduce a formal notation
  • Using logical rules of inference, just like for typing
• Recall the typing judgment
  Context ⊢ e : C
  (in the given context, expression e has type C)
• We try something similar for evaluation
  Context ⊢ e : v
  (in the given context, expression e evaluates to value v)
What Contexts Are Needed?

- Obs.: Contexts are needed to handle variables
- Consider the evaluation of $y \leftarrow x + 1$
  - We need to keep track of values of variables
  - We need to allow variables to change their values during the evaluation
- We track variables and their values with:
  - An *environment* tells us at what address in memory is the value of a variable stored
  - A *store* tells us what is the contents of a memory location
Stores

- A store maps memory locations to values
- Example:
  \[ S = [l_1 \rightarrow 5, l_2 \rightarrow 7] \]
- To *lookup* the contents of a location \( l_1 \) in store \( S \) we write \( S(l_1) \)
- To perform an *assignment* of 12 to location \( l_1 \) we write \( S[12/l_1] \)
- This denotes a new store \( S' \) such that for some location \( l \)
  \[ S'(l_1) = 12 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1 \]
Operational Rules of Cool

• The evaluation judgment is
  \[ \text{so, } E, S \vdash e : v, S' \]

• read:
  • Given the object “so” with the current value of self
  • And \( E \) the current variable environment
  • And \( S \) the current store
  • If the evaluation of \( e \) terminates then
  • The returned value is \( v \)
  • And the new store is \( S' \)
Operational Semantics of new

- Consider the expression $new\ T$
- Informal semantics (we’ll do formal ones in a moment)
  - Essentially, allocate a new object
  - Allocate new locations to hold the values for all attributes of an object of class $T$
  - Initialize those locations with the default values of attributes
  - Evaluate the initializers and set the resulting attribute values
  - Return the newly allocated object
Default Values

- For each class A there is a default value denoted by $D_A$
  - $D_{\text{int}} = \text{Int}(0)$
  - $D_{\text{bool}} = \text{Bool}(\text{false})$
  - $D_{\text{string}} = \text{String}(0, "")$
  - $D_A = \text{void}$ (for another class A)
For a class \( A \) we write the following mapping:

\[
\text{class}(A) = (a_1:T_1 \leftarrow e_1, \ldots, a_n:T_n \leftarrow e_n)
\]

where

- \( a_i \) are the attributes (including the inherited ones)
- \( T_i \) are their declared types
- \( e_i \) are the initializers
Operational Semantics

of new

Obs. : new SELF_TYPE allocates an object with the same dynamic type as self

if $T ==$ SELF_TYPE and $so = X(\ldots)$ then $T_0 = X$ else $T_0 = T$

$\text{class}(T_0) = (a_1:T_1 \leftarrow e_1, \ldots, a_n:T_n \leftarrow e_n)$

$i_i = \text{newloc}(S) \text{ foreach } i = 1, \ldots, n$

$v = T_0(a_1 = i_1, \ldots, a_n = i_n)$

$E' = [a_1: i_1, \ldots, a_n: i_n]$

$S_1 = S[D_{T_1}/i_1, \ldots, D_{T_n}/i_n]$

$v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \ldots; a_n \leftarrow e_n; \} : v_n, S_2$

so, $E, S \vdash \text{new } T : v, S_2$

- The first three lines allocate the object
- The rest of the lines initialize it
- By evaluating a sequence of assignments
- State in which the initializers are evaluated
- Self is the current object
- Only the attributes are in scope (same as in typing)
- Starting value of attributes are the default ones
- The side-effect of initialization is preserved
Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1,\ldots,e_n)$
- Informal semantics:
  - Evaluate the arguments in order $e_1,\ldots,e_n$
  - Evaluate $e_0$ to the target object
  - Let $X$ be the dynamic type of the target object
  - Fetch from $X$ the definition of $f$ (with $n$ args.)
  - Create $n$ new locations and an environment that maps $f$’s formal arguments to those locations
  - Initialize the locations with the actual arguments
  - Set self to the target object and evaluate $f$’s body
More Notation

• For a class $A$ and a method $f$ of $A$ (possibly inherited) we write the following mapping:

$$\text{implementation}(A, f) = (x_1, \ldots, x_n, e_{\text{body}})$$

where

• $x_i$ are the names of the formal arguments

• $e_{\text{body}}$ is the body of the method
Operational Semantics of Dispatch

so, E, S ⊢ e₁ : v₁, S₁
so, E, S₁ ⊢ e₂ : v₂, S₂
...
so, E, Sₙ₋₁ ⊢ eₙ : vₙ, Sₙ
so, E, Sₙ ⊢ e₀ : v₀, Sₙ₊₁

v₀ = X(a₁ = l₁, ..., aₘ = lₘ)
implementation(X, f) = (x₁, ..., xₙ, e_body)
lₓᵢ = newloc(Sₙ₊₁) for i = 1, ..., n
E’ = [x₁: lₓ₁, ..., xₙ: lₓₙ, a₁: l₁, ..., aₘ: lₘ]
Sₙ₊₂ = Sₙ₊₁[v₁/lₓ₁, ..., vₙ/lₓₙ]

v₀, E’, Sₙ₊₂ ⊢ e_body : v, Sₙ₊₃

so, E, S ⊢ e₀.f(e₁, ..., eₙ) : v, Sₙ₊₃
Operational Semantics of Dispatch. Notes.

- The body of the method is invoked with
- $E$ mapping formal arguments and self’s attributes
- $S$ like the caller’s except with actual arguments bound to the locations allocated for formals
- The notion of the activation frame is implicit
- New locations are allocated for actual arguments
- The semantics of static dispatch is similar except the implementation of $f$ is taken from the specified class
Runtime Errors

• Operational rules do not cover all cases. Consider for example the rule for dispatch:

\[
\text{... so, E, } S_n \vdash e_0 : v_0, S_{n+1}
\]
\[
v_0 = X(a_1 = l_1, \ldots, a_m = l_m)
\]
\[
\text{implementation}(X, f) = (x_1, \ldots, x_n, e_{\text{body}})
\]
\[
\text{... so, E, } S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}
\]

• What happens if the mapping function implementation\((X, f)\) is not defined?

• Cannot happen in a well-typed program (type safety theorem)
• There are some runtime errors that the type checker does not try to prevent
  • A dispatch on void
  • Division by zero
  • Substring out of range
  • Heap overflow
• In such case the execution should abort gracefully
  • With an error message not with segfault
Conclusions

- Operational rules are very precise
- Nothing (that can be specified) is left unspecified
- Operational rules contain a lot of details
- Read them carefully
- Most languages do not have a well specified operational semantics (but Cool does)
- When portability is important an operational semantics becomes essential (hence portability problems)
- But not always using the notation we used for Cool
Local Optimizations
Code Generation Summary

- We have discussed
  - Runtime organization
  - Simple stack machine code generation
  - Improvements to stack machine code generation
- Our compiler goes directly from AST to assembly language
  - And does not perform optimizations
- Most real compilers use intermediate languages
Why Intermediate Languages?

- When to perform optimizations
  - On AST
    - Pro: Machine independent
    - Con: Too high level
  - On assembly language
    - Pro: Exposes optimization opportunities
    - Con: Machine dependent
    - Con: Must re-implement optimizations when re-targeting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities
    - Con: One more language to worry about
Intermediate Languages

- Each compiler uses its own intermediate language
- IL design is still an active area of research
- Intermediate language = high-level assembly language
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
Three-Address Intermediate Code

• Each instruction is of the form
  \[ x := y \text{ op } z \]
• \( y \) and \( z \) can be only registers or constants
• Just like assembly
• Common form of intermediate code
• The AST expression \( x + y \ast z \) is translated as
  \[
  t_1 := y \ast z \\
  t_2 := x + t_1
  \]
• Each sub expression has a “name”
Generating Intermediate Code

- Similar to assembly code generation
- Major difference
  - Use any number of IL registers to hold intermediate results
Generating Intermediate Code

- igen(e, t) function generates code to compute the value of e in register t
- Example:
  - igen(e₁ + e₂, t) =
    - igen(e₁, t₁)  (t₁ is a fresh register)
    - igen(e₂, t₂)  (t₂ is a fresh register)
    - t := t₁ + t₂
- Unlimited number of registers
  ⇒ simple code generation
Intermediate Code Notes

• You should be able to use intermediate code
  • At the level discussed in lecture
• You are not expected to know how to generate intermediate code
  • Because we won’t discuss it
• But really just a variation on code generation …
Notes

• Dragon book has very good coverage on
  • Local and global optimizations (Ch. 10) *optional*
  • Also good coverage on
    • Intermediate code (Ch. 8) *read this*
    • Code generation (Ch. 9) *optional*
An Intermediate Language

\[ P \rightarrow S \ P \mid \varepsilon \]
\[ S \rightarrow \text{id := id op id} \]
\[ \mid \text{id := op id} \]
\[ \mid \text{id := id} \]
\[ \mid \text{push id} \]
\[ \mid \text{id := pop} \]
\[ \mid \text{if id relop id goto L} \]
\[ \mid \text{L:} \]
\[ \mid \text{jump L} \]

- id’s are register names
- Constants can replace id’s
- Typical operators: +, -, *

Wednesday, May 20, 2009
Two Useful Concepts

- Basic blocks (BB)
  - Split code into basic atomic units
- Control-flow graphs (CFG)
  - Connect the BBs together as a directed graph
  - Useful for representing intermediate code
  - Use a graphical representation
  - Make control-flow explicit
Definition. Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)
- Idea:
  - Cannot jump in a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - Each instruction in a basic block is executed after all the preceding instructions have been executed
Basic Block Example

- Consider the basic block
  1. L:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L’

- No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \( w := 3 \times x \)
  - Can we eliminate (2) as well?
Definition. Control-Flow Graphs

- A control-flow graph is a directed graph with
- Basic blocks as nodes
- An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B
- E.g., the last instruction in A is jump $L_B$
- E.g., the execution can fall-through from block A to block B
- Frequently abbreviated as CFG
Control-Flow Graphs. Example.

- The body of a method (or procedure) can be represented as a control-flow graph.
- There is one initial node.
- All “return” nodes are terminal.
Optimization Overview

- Optimization seeks to improve a program’s utilization of some resource
- Execution time (most often)
- Code size
- Network messages sent
- Battery power used, etc.
- Optimization should not alter what the program computes – *correctness*
- The answer must still be the same
A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
  1. Local optimizations
     - Apply to a basic block in isolation
  2. Global optimizations (a.k.a. intra-procedural)
     - Apply to a control-flow graph (method body) in isolation
  3. Inter-procedural optimizations
     - Apply across method boundaries
     - Most compilers do (1), many do (2) and very few do (3)
Cost of Optimizations

• In practice, a conscious decision is made not to implement the fanciest optimization known

• Why?
  • Some optimizations are hard to implement
  • Some optimizations are costly in terms of compilation time
  • The fancy optimizations are both hard and costly

• The goal
  • Maximum improvement with minimum cost
Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
  - Just the basic block in question
- Example: algebraic simplification
Algebraic Simplification

- Some statements can be deleted
  - $x := x + 0$
  - $x := x \times 1$

- Some statements can be simplified
  - $x := x \times 0 \Rightarrow x := 0$
  - $y := y^{**} 2 \Rightarrow y := y \times y$
  - $x := x \times 8 \Rightarrow x := x <\times 3$
  - $x := x \times 15 \Rightarrow t := x <\times 4; x := t - x$
  - (on some machines $<\times$ is faster than $\times$; but not on all!)
Constant Folding

• Operations on constants can be computed at compile time
• In general, if there is a statement
  • \( x := y \text{ op } z \)
  • And \( y \) and \( z \) are constants
  • Then \( y \text{ op } z \) can be computed at compile time
• Example: \( x := 2 + 2 \Rightarrow x := 4 \)
• Example: if \( 2 < 0 \) jump L can be deleted
• When might constant folding be dangerous?
Flow of Control Optimizations

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph
  - Basic blocks that are not the target of any jump or “fall through” from a conditional
  - Such basic blocks can be eliminated
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
  - And sometimes also faster
  - Due to memory cache effects (increased spatial locality)
Single Assignment Form

• Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment
• Intermediate code can be rewritten to be in single assignment form
  \[
  \begin{align*}
  x &:= z + y & b &:= z + y \\
a &:= x & a &:= b \\
x &:= 2 * x & x &:= 2 * b \\
\end{align*}
\]
  (b is a fresh register)
• More complicated in general, due to loops
Common Subexpression Elimination

- **Assume**
  - Basic block is in single assignment form
  - A definition $x :=$ is the first use of $x$ in a block
- All assignments with same rhs compute the same value
- **Example:**
  
  $x := y + z$
  $\cdots$
  $\Rightarrow$
  $\cdots$

  
  $w := y + z$
  $w := x$

- (the values of $x$, $y$, and $z$ do not change in the...code)
Copy Propagation

• If \( w := x \) appears in a block, all subsequent uses of \( w \) can be replaced with uses of \( x \)

• Example:

\[
\begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 * a
\end{align*}
\]

\[
\begin{align*}
  b & := z + y \\
  a & := b \\
  x & := 2 * b
\end{align*}
\]

• This does not make the program smaller or faster but might enable other optimizations

• – Constant folding

• – Dead code elimination
Copy Propagation and Constant Folding

- Example:
  
  \[
  a := 5 \\
  x := 2 \times a \\
  y := x + 6 \\
  t := x \times y
  \]

  \[
  \Rightarrow \\
  x := 10 \\
  y := 16 \\
  t := x \ll 4
  \]
Finish Project 3!