

Cobol
Scheme
Tcl/Tk
Lisp
Rexx
Simula
Java
Logo
SML
Basic
PostScript
C++
APL
Fortran
Modula
SmallTalk

The Structure of Typed Programming Languages
(Chapter 1 - continued)

CS386L- Programming Languages

Dr. Greg Lavender
Department of Computer Sciences
The University of Texas at Austin

CS386L: Lecture-06 - The Structure of Typed Programming Languages: Chapter 1 - cont.

Semantics of the Core Language

- The purpose of a well-designed syntax is to guide the programmer's understanding of the semantics, leading to correct programs
 - but we need more than just intuitive understanding
- Core imperative programming language
 - locations and numerals represent themselves
 - expressions represent integers and booleans
 - commands represent state transformations
- Denotational semantics is one way to formalize the semantics of a language
 - what is the meaning of each legal syntactic phrase?
 - a denotational semantics maps well-typed derivation trees to their mathematical meanings
 - $[[E1 := E2:comm]] = ???$

3/1/04 2

Semantic domains & operations

Bool = { false, true }
 not: Bool \rightarrow Bool
 not(false) = true
 not(true) = false
 equalbool: Bool * Bool \rightarrow Bool
 equalbool (m,n) = (m = n)

Int = $\{-\infty, \dots, -1, 0, 1, \dots, \infty\}$
 plus: Int * Int \rightarrow Int
 plus(m,n) = m + n
 equalint: Int * Int \rightarrow Bool
 equalint (m,n) = (m=n)

Location { loc_i | i > 0 }

Store = { $\langle n_1, n_2, \dots, n_m \rangle$ | n_i in Int, $1 \leq i \leq m$, $m \geq 0$ }
 lookup: Location * Store \rightarrow Int
 update: Location * Int * Store \rightarrow Store
 if: Bool * Store * Store \rightarrow Store
 if(true,s1,s2)=s1
 if(false,s1,s2)=s2

3/1/04

3

Semantics of the Core Language

- A denotational semantics is a recursive definition that maps well-typed derivation trees to their mathematical (functional) meanings
 - Bool is a semantic domain with two values
 - true and false
 - two semantic functions (in a sugared λ -calculus)
 - not - logical negation
 - equalbool - boolean equality
 - Int is the set of integers
 - addition and equality functions
 - $[[\underline{2}:\text{int}]] = 2$, where $\underline{2}$ is a numeral and 2 is an integer
 - Location is a set of labeled locations
 - $[[\text{loc}_i:\text{intloc}]] = \text{loc}_i$

3/1/04

4

The Store

- An abstraction over locations and values
 - semantics for l-values and r-values
 - store is a linear storage vector
 - lookup function
 - $\text{lookup}(\text{loc}_j, \langle n_1, n_2, \dots, n_j, \dots, n_m \rangle) = n_j$
 - if $j > m$, $\text{lookup}(\text{loc}_j, \langle n_1, \dots, n_m \rangle) = 0$
 - update function
 - $\text{update}(\text{loc}_j, n, \langle n_1, n_2, \dots, n_j, \dots, n_m \rangle) = \langle n_1, n_2, \dots, n, \dots, n_m \rangle$
 - if $j > m$, $\text{update}(\text{loc}_j, \langle n_1, \dots, n_m \rangle) = \langle n_1, \dots, n_m \rangle$
- The if operation selects a store based on a boolean expression evaluation

3/1/04

5

Compositional Semantics

- The meaning of a well-typed program is a recursive equation using $[[\cdot]]$
 - for $L := E$ we compute the meaning of this command using a semantic equation along with a typing rule
 - the typing rule indicates the compositional elements of the rhs of the equation

$$\begin{array}{l} L:\text{intloc} \quad E:\text{intexp} \\ \hline L := E : \text{comm} \end{array}$$

$$[[L := E : \text{comm}]] = \dots [[L:\text{intloc}]] \dots [[E:\text{intexp}]]$$

3/1/04

6

Referential Transparency

- semantic definitions are referentially transparent if two phrases have the same meaning they can be substituted for one another
 - $\text{plus}(\text{plus}(2,3),4) = \text{plus}(5,4)$

3/1/04

7

Semantic equations

Command:

```

[[L:=E:comm]](s) = update([[L:intloc]], [[E:intexp]](s),s)
[[C1;C2:comm]](s) = [[C2:comm]] ([[C1:comm]](s))
[[if E then C1 else C2 fi:comm]](s) = if ([[E:boolexp]](s)), [[C1:comm]](s), [[C2:comm]](s)
[[while E do C od:comm]](s) = w(s)
    where w(s) = if ([[E:boolexp]](s), w[[C:comm]](s)), s
[[skip:comm]](s) = s
  
```

Expression:

```

[[N:intexp]](s) = [[N:int]]
[[@L:intexp]](s) = lookup([[L:intloc]],s)
[[E1+E2:intexp]](s) = plus([[E1:intexp]](s), [[E2:intexp]](s))
[[~E1:boolexp]](s) = not ([[E1:boolexp]](s))
[[E1=E2:boolexp]](s) = equalbool([[E1:boolexp]](s), [[E2:boolexp]](s))
[[E1=E2:boolexp]](s) = equalint([[E1:intexp]](s), [[E2:intexp]](s))
  
```

Location: $[[loc_i:intloc]] = loc_i$

Numeral: $[[n:int]] = n$

3/1/04

8

Semantics of Expressions

- Semantics of an expression is dependent on the store
 - given the initial storage vector $\langle 3, 4, 5 \rangle$
 - example 1:
 - $[[@loc_1: intexp]]\langle 3, 4, 5 \rangle = \text{lookup}([[loc_1: intloc]], \langle 3, 4, 5 \rangle) = 3$
 - example 2:
 - $[[@loc_1+1: intexp]]\langle 3, 4, 5 \rangle = \text{plus}([[@loc_1: intloc]]\langle 3, 4, 5 \rangle, [[1: intexp]]\langle 3, 4, 5 \rangle)$
 $= \text{plus}(3, [[1: int]])$
 $= \text{plus}(3, 1) = 4$

Semantics of Commands

- Semantics of a command is dependent on the store
 - a command changes the store into a new one
 - example:
 - $[[loc_3 := @loc_1 + 1: comm]]\langle 3, 4, 5 \rangle$
 $= \text{update}([[loc_3: intloc]], [[@loc_1+1: intexp]]\langle 3, 4, 5 \rangle, \langle 3, 4, 5 \rangle)$
 $= \text{update}(loc_3, 4, \langle 3, 4, 5 \rangle) = \langle 3, 4, 4 \rangle$

Command Composition

- meaning of command sequencing is function composition of the corresponding denotations
- $[[C1;C2:comm]](s) = [[C2:comm]]([[C1:comm]](s))$
- example:
 - $[[loc_3 := @loc_1+1;$
 - $if @loc_3=0 then loc_2:=1 else skip:comm]]\langle 3,4,5 \rangle$
 - $= [[if @loc_3 = 0 then loc_2 := 1 else skip:comm]]$
 - $([[loc_3 := @loc_1+1:comm]]\langle 3,4,5 \rangle)$
 - $= [[if @loc_3 = 0 then loc_2 := 1 else skip:comm]]\langle 3,4,4 \rangle$
 - $= if([[@loc_3=0:boolexp]]\langle 3,4,4 \rangle, [[loc_2:=1:comm]]\langle 3,4,4 \rangle,$
 - $[[skip:comm]]\langle 3,4,4 \rangle)$
 - $= if (false, [[loc_2 := 1:comm]]\langle 3,4,4 \rangle, [[skip:comm]]\langle 3,4,4 \rangle)$
 - $= [[skip:comm]]\langle 3,4,4 \rangle = \langle 3,4,4 \rangle$

3/1/04

11

The While loop

- iteration and recursion present a challenge in semantics
 - what is required is a way to represent a self-referential object in the semantic domain
 - one way to do this is to consider a special semantic function that incrementally approximates or converges to the final value of the iteration/recursion
 - e.g., we can think of a recursive factorial program as denoting a factorial function that on each recursion approaches the limit of the function in a mathematical sense by evaluating itself repeatedly until some terminating condition

3/1/04

12

The w function

- This function is actually a family of functions that approximate an answer by unfolding, like unrolling a loop
 - $w_0(s) = \text{"bottom"}$
 - $w_1(s) = \text{if} ([[E:\text{boolexp}]](s), w_0([[C:\text{comm}]](s)), s)$
 - $w_2(s) = \text{if} ([[E:\text{boolexp}]](s), w_1([[C:\text{comm}]](s)), s)$
 - ...
 - $w_{i+1}(s) = \text{if} ([[E:\text{boolexp}]](s), w_i([[C:\text{comm}]](s)), s)$
- for all store vectors s and s' , $w(s) = s'$ iff there is some $k \geq 0$ such that $w_k(s) = s'$
 - $w_k(s) = s'$ implies $w_j(s) = s'$ for all $j \geq k$
 - i.e., a termination condition

3/1/04

13

While semantics

- Example
 - $[[\text{while } @loc_1 = 0 \text{ do } loc_1 := @loc_1+1:\text{comm}]]\langle 0, 0 \rangle$
 $= w(\langle 0, 0 \rangle)$
 $= \text{if} ([[@loc_1=0:\text{boolexp}]]\langle 0, 0 \rangle, w([[loc_1:=@loc_1+1:\text{comm}]]\langle 0, 0 \rangle), \langle 0, 0 \rangle)$
 $= \text{if} (\text{true}, w([[loc_1:=@loc_1+1:\text{comm}]]\langle 0, 0 \rangle), \langle 0, 0 \rangle)$
 $= w([[loc_1:=@loc_1+1:\text{comm}]]\langle 0, 0 \rangle)$
 $= w(\langle 1, 0 \rangle)$
 $= \text{if} ([[@loc_1=0:\text{boolexp}]]\langle 1, 0 \rangle, w([[loc_1:=@loc_1+1:\text{comm}]]\langle 1, 0 \rangle), \langle 1, 0 \rangle)$
 $= \langle 1, 0 \rangle$
 - an output store is produced by w in a finite number of k unfoldings until w_k produces the same output store for an input store, and then the loop terminates
 - i.e., w reaches a fixed point where $w(\langle 1, 0 \rangle) = \langle 1, 0 \rangle$
 - refer back to the fixed point version of Kleene's recursion theorem

3/1/04

14

Operational semantics

- semantics of operational steps required to evaluate a program
 - operational semantics are helpful in constructing an interpreter for a language
 - for example, we could implement all of the semantic functions as procedures in an interpreter, augmented with an "environment" for looking up and updating variables (locations) with storage values in RAM
 - The goal then is to reduce a program to some value (its normal form) by a step-by-step evaluation of each syntactic phrase according to the type rules and the semantic equations

3/1/04

15

An interpreter

- First read a program and do lexical analysis and parsing into an internal form (e.g., an abstract syntax tree)
- Have an "eval" procedure that can reduce any valid expression in the language to its normal form
 - for each expression, the eval procedure applies the semantic function for that expression using the current state of the interpreter as the store
- Print out the result of the expression
- Repeat for the next expression
- This is called a Read-Eval-Print-Loop interpreter

3/1/04

16

Computation steps

- A computation is a sequence of steps
 - $p_0 \Rightarrow p_1 \Rightarrow \dots \Rightarrow p_n, n \geq 0$
 - we say $p_0 \Rightarrow^* p_n$ in zero or more steps
 - if p_n is a value, terminate the computation
 - for Bool, true and false are values
 - for Int, numerals are values
 - for Location, every loc_i is a value
 - for Store, every n_i in a vector is a value, so a vector is a value
 - recall that expressions and commands produce new stores
 - for the core imperative language, a program is some phrase $[[C:comm]]s_0$ starting with the initial store s_0

Desired Operational Properties

- subject reduction: if p has an underlying type, τ , and $p \Rightarrow^* p'$, then p' has type τ
- soundness: if p has the underlying mathematical meaning, m , and $p \Rightarrow^* p'$, then p' has the same meaning m
- strong typing: if p is well-typed and $p \Rightarrow^* p'$, then p' has no operator-operand type errors
- computational adequacy: the meaning m of a program p is a proper meaning iff there is a value v such that $p \Rightarrow^* v$ and v has meaning m .

Designing a Language Core

- possible objectives
 - domain specific
 - general purpose
 - user friendly
 - efficient in implementation
 - extensible
 - secure
 - orthogonal
 - simple syntax and semantics
 - good tools (editors, debuggers, refactoring, etc)
 - facilitates automatic verification
 - portable
 - exploits system features (concurrent, parallel, distributed)

Reading Assignment

- *The Next 700 Programming Languages*, by Peter Landin
- *Definitional Interpreters for Higher-Order Programming Languages*, by John Reynolds
- *Definitional Interpreters Revisited*, by John Reynolds