Denotational semantics, Pure functional programming

John Mitchell

Reading: Chapter 4, sections 4.3, 4.4 only
Skim section 4.1 for background on parsing

Syntax and Semantics of Programs

• Syntax
  – The symbols used to write a program
• Semantics
  – The actions that occur when a program is executed
• Programming language implementation
  – Syntax $\rightarrow$ Semantics
  – Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur

Theoretical Foundations

• Many foundational systems
  – Computability Theory
  – Program Logics
  – Lambda Calculus
  – Denotational Semantics
  – Operational Semantics
  – Type Theory
• Consider some of these methods
  – Computability theory (halting problem)
  – Lambda calculus (syntax, operational semantics)
  – Denotational semantics

Denotational Semantics

• Describe meaning of programs by specifying the mathematical
  – Function
  – Function on functions
  – Value, such as natural numbers or strings defined by each construct

Original Motivation for Topic

• Precision
  – Use mathematics instead of English
• Avoid details of specific machines
  – Aim to capture “pure meaning” apart from implementation details
• Basis for program analysis
  – Justify program proof methods
    • Soundness of type system, control flow analysis
  – Proof of compiler correctness
  – Language comparisons

Why study this in CS 242?

• Look at programs in a different way
• Program analysis
  – Initialize before use, ...
• Introduce historical debate: functional versus imperative programming
  – Program expressiveness: what does this mean?
  – Theory versus practice: we don’t have a good theoretical understanding of programming language “usefulness”
Basic Principle of Denotational Sem.

- Compositionality
  - The meaning of a compound program must be defined from the meanings of its parts (not the syntax of its parts).
- Examples
  - $P; Q$
    - composition of two functions, state $\rightarrow$ state
  - letrec $f(x) = e_1$ in $e_2$
    - meaning of $e_2$ where $f$ denotes function ...

Trivial Example: Binary Numbers

- Syntax
  - $b ::= 0 \mid 1$
  - $n ::= b \mid nb$
  - $e ::= n \mid e + e$
- Semantics
  - value function $E : \exp \rightarrow$ numbers
  - $E[[0]] = 0$
  - $E[[1]] = 1$
  - $E[[nb]] = 2E[[n]] + E[[b]]$
  - $E[[e_1 + e_2]] = E[[e_1]] + E[[e_2]]$

Second Example: Expressions w/vars

- Syntax
  - $d ::= 0 \mid 1 \mid 2 \mid \ldots \mid 9$
  - $n ::= d \mid nd$
  - $e ::= x \mid n \mid e + e$
- Semantics
  - value $E : \exp \times \state \rightarrow$ numbers
  - $E[[x]]s = s(x)$
  - $E[[0]]s = 0$
  - $E[[1]]s = 1$
  - $E[[nd]]s = 10 * E[[n]]s + E[[d]]s$
  - $E[[e_1 + e_2]]s = E[[e_1]]s + E[[e_2]]s$

Semantics of Imperative Programs

- Syntax
  - $P ::= x := e \mid \text{if } B \text{ then } P \text{ else } Q \mid P; P \mid \text{while } B \text{ do } P$
- Semantics
  - $C : \text{Programs} \rightarrow (\state \rightarrow \state)$
  - $\text{State} = \text{Variables} \rightarrow \text{Values}$
    - would be locations $\rightarrow$ values if we wanted to model aliasing

Semantics of Assignment

$C[[x := e]]$

is a function states $\rightarrow$ states

$C[[x := e]]s = s'$

where $s'$ : variables $\rightarrow$ values is identical to $s$ except
$s'(x) = E[[e]]s$ gives the value of $e$ in state $s$

Semantics of Conditional

$C[[\text{if } B \text{ then } P \text{ else } Q]]$

is a function states $\rightarrow$ states

$C[[\text{if } B \text{ then } P \text{ else } Q]]s =$

$C[[P]]s$ if $E[[B]]s$ is true
$C[[Q]]s$ if $E[[B]]s$ is false

Simplification: assume $B$ cannot diverge or have side effects
Semantics of Iteration

C[[ while B do P ]] is a function states → states

C[[ while B do P ]] = the function f such that
f(s) s̸∈ E[[ B ]] s is false
f(s) f( C[[ P ]] (s)) s ∈ E[[ B ]] s is true

Mathematics of denotational semantics: prove that there is such a function and that it is uniquely determined. "Beyond scope of this course."

Perspective

- Denotational semantics
  - Assign mathematical meanings to programs in a structured, principled way
  - Imperative programs define mathematical functions
  - Can write semantics using lambda calculus, extended with operators like modify: (state × var × value) → state
- Impact
  - Influential theory
  - Applications via abstract interpretation, type theory, ...

Functional vs Imperative Programs

- Denotational semantics shows
  - Every imperative program can be written as a functional program, using a data structure to represent machine states
- This is a theoretical result
  - I guess "theoretical" means "it's really true" (?)
- What are the practical implications?
  - Can we use functional programming languages for practical applications?
    - Compilers, graphical user interfaces, network routers, ...

What is a functional language?

- “No side effects”
- OK, we have side effects, but we also have higher-order functions...

We will use pure functional language to mean "a language with functions, but without side effects or other imperative features."

No-side-effects language test

Within the scope of specific declarations of x1, x2, ..., xn, all occurrences of an expression e containing only variables x1, x2, ..., xn must have the same value.

- Example
  begin
  integer x=3; integer y=4;
  5*(x+y)-3
  ... // no new declaration of x or y //
  d^[x,y]=1
  end

Example languages

- Pure Lisp
  atom, eq, car, cdr, cons, lambda, define
- Impure Lisp: rplaca, rplacd
  lambda (x) (cons (car x) (... rplaca (... x) ...) (car x) ...
  })
  Cannot just evaluate (car x) once
- Common procedural languages are not functional
  - Pascal, C, Ada, C++, Java, Modula, ...
  Example functional programs in a couple of slides
Backus’ Turing Award

- John Backus was designer of Fortran, BNF, etc.
- Turing Award in 1977
- Turing Award Lecture
  - Functional prog better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
  - Reason about programs
  - Optimizing compilers

Reasoning about programs

- To prove a program correct,
  - must consider everything a program depends on
- In functional programs,
  - dependence on any data structure is explicit
- Therefore,
  - easier to reason about functional programs
- Do you believe this?
  - This thesis must be tested in practice
  - Many who prove properties of programs believe this
  - Not many people really prove their code correct

Haskell Quicksort

- Very succinct program

qsort [] = []
qsort (x:xs) = qsort (y:ys) ++ [x]
  ++ qsort (elts_greq_x)
where elts_greq_x = [y | y < xs, y < x]
  elts_greq_x = [y | y < xs, y >= x]
- This is the whole thing
  - No assignment – just write expression for sorted list
  - No array indices, no pointers, no memory management, ...

Compare: C quicksort

```c
qsort(a, lo, hi) { int il, ih, n, lo, hi;
  if (lo < hi) {
    l = lo; h = hi; p = a[hi];
    do {
        while ((l < h) && (a[l] <= p)) l = l + 1;
        while ((h > l) && (a[h] >= p)) h = h - 1;
        if (l < h) (t = a[l], a[l] = a[h], a[h] = t);
        qsort(a, lo, l-1);
        qsort(a, l+1, hi);
    }
```
Sample Optimization: Update in Place

Function uses updated list

\[
\text{list-update } x 3 y \text{ (cons } 'E \text{ (cdr } x) \text{ ...)}
\]

Can we implement list-update as assignment to cell? May not improve efficiency if there are multiple pointers to list, but should help if there is only one.

Sample Optimization: Update in Place

Initial list x

List x after (list-update x 3 y)

Initial list x

List x after (list-update x 3 y)

This works better for arrays than lists.

Sample Optimization: Update in Place

Array A

Update(A, 3, 5)

• Approximates efficiency of imperative languages
• Preserves functional semantics (old value persists)

Von Neumann bottleneck

• Von Neumann
  – Mathematician responsible for idea of stored program
• Von Neumann Bottleneck
  – Backus’ term for limitation in CPU-memory transfer
• Related to sequentiality of imperative languages
  – Code must be executed in specific order
    function f(x) { if (x < y) then y = x; else x = y; }
    g(f(i), f(j));

Eliminating VN Bottleneck

• No side effects
  – Evaluate subexpressions independently
  – Example
    • function f(x) { return x < y ? 1 : 2; }
    • g(f(0), f(0), f(0), ...);
• Does this work in practice? Good idea but ...
  – Too much parallelism
  – Little help in allocation of processors to processes
  – ... 
    – David Shaw promised to build the non-Von ...
• Effective, easy concurrency is a hard problem