Programming Languages
COS 441

Intro

Denotational Semantics I

Course Staff

David Walker
Professor

Research: Programming Languages
Email: dpw@cs

Chris Monsanto
Grad Student TA

Research: Programming Languages
Email: cmmonsan@cs

This Week (Sept 16, 19, 21)

Professor Walker is in Tokyo, at the International Conference on Functional Programming

Andrew Appel
Professor & Department Chair

Research: Programming Languages

What is this course about?

- What do programs do?
  - We are going to use *mathematics* as opposed to *English* or *examples* to describe what programs do
  - Our descriptions are going to be *complete* and *exact*
    - For any language we study, they will cover all programs and all corner cases

- How do we answer questions about programs and programming languages?
  - Since we have complete and exact mathematical descriptions of programs, we can *prove* strong properties about them
    - eg: Will *this* program crash? Will *any* program crash?

- Experience new and powerful programming languages

How is this course different from COS 441 last year?

- Last year, the new and powerful PL was called Coq
  - Coq is interactive theorem prover, not a programming language you'd use to build ordinary, day-to-day applications

- This year, the new and powerful PL is Haskell
  - Haskell is a programming language you'd use to build day-to-day applications
  - It's a functional language with an amazing type system and a terse syntax that puts Java to shame
    - one line of Haskell is often worth 2 or more lines of Java
  - It's got a bunch of cool features like *higher-order functions*, *infinite data structures*, *algebraic data types*, *data-parallel programming* and *concurrent transactions*
  - It's also got strong support for defining your own *domain-specific languages* (DSLs); we'll explore that support by developing DSLs for creating simple animations, modeling financial contracts and programming networks

Logistics & Homework

- Sign up for the course email list:
  - https://lists.cs.princeton.edu/mailman/listinfo/cos441
- Read the course web site:
  - http://www.cs.princeton.edu/~dpw/cos441-11
- Start assignment 1
  - you can do parts I and II when this lecture is complete
  - you can do other parts after the second or third lectures
  - due 1pm Tuesday Sept 27
  - see course web site for late policy
- Get ready for assignment 2 by downloading Haskell:
  - http://hackage.haskell.org/platform//
  - see course web site for learning materials

THIS WEEK:
DENOTATIONAL SEMANTICS

Semantics of Programs

- Many ways to use mathematics to give meaning to programs
  - Operational semantics: a step by step account of how to execute a program. For each instruction, explain what program variables or data structures get updated. Useful for building an interpreter that executes a program and computes its results. Easy to scale to very complex languages. Easy to prove some simple properties of programs. Harder to prove deeper properties without additional work.
  - Axiomatic semantics: describes what a program does in terms of logical preconditions and postconditions. Useful for building program analyzers that examine programs before they are run to detect bugs.
  - Denotational semantics: describes the meaning of a program by transforming the syntax of the program into a well-known mathematical object like a set or a mathematical function. Easy to describe and prove deep properties about simple languages. Harder to scale in some cases.
    - We will start with simple denotational semantics

Denotational Modus Operandi

- When employing denotational semantics we are going to proceed as follows:
  1. Define the syntax of the language
     - How do you write the programs down?
     - Use BNF notation (BNF = Backus Naur Form)
  2. Define the denotation (aka meaning) of the language
     - Use a function from syntax to mathematical objects
     - Make sure the function is inductive and (usually) total
  3. Prove something about the language
     - Most of our proofs about denotational definitions will be by induction on the structure of the syntax of the language
       - We will explain what that means and how to do it in a later lecture.

DEFINING SYNTAX

Binary Numbers: Informal Definitions

- Examples of the syntax of binary numbers:
  - #1
  - #0
  - #110
  - #1101010
  - #00101
  - # equivalent to zero

- English description of the syntax binary numbers:
  - A binary number is a hash sign followed by a (possibly empty) sequence of zeros

Binary Numbers: Formal Syntax

“:=“ can be read “is defined to be”

b ::= # | b0 | b1

metavariable b
stands for any item
being defined

vertical bar separates
alternatives in the definition

Examples:

- #01
- #
- #1
- #0001

- How to read the definition in English:
  - a b can either be:
    - a #, or
    - any b followed by a 0, or
    - any b followed by a 1

Binary Numbers: Formal Syntax

":=" can be read "is defined to be"

Examples:

- #01
- #
- #1
- #0001

b ::= # | b0 | b1

metavariable b stands for any item being defined

vertical bar separates alternatives in the definition

- Question: is #01 a binary number? Yes. Justification:
  - #01 has the form b1 where b = #0 and:
  - #0 has the form b'0 where b' = # and:
  - # is unconditionally a binary number
- Comment: if we need to refer to lots of different binary numbers, we will use the same basic letter but add primes and subscripts: b', b'', b''', b_1, b_2, ... to distinguish them

Binary Numbers: Formal Syntax

“:=“ can be read “is defined to be”

b ::= # | b0 | b1

metavariable b
stands for any item
being defined

vertical bar separates
alternatives in the definition

Examples:
- #01
- #
- #1
- #0001

- Question: is #071 a binary number? No! Justification:
  - #071 can only be a binary number if it matches one of the three patterns given above. #071 matches the second pattern if #07 is a binary number, but:
  - #07 is not a binary number because it is not # and it does not have the form b0 and it does not have the form b1 for any b

Binary Numbers: Formal Syntax

b ::= # | b0 | b1

- What we've got so far:
  - some notation defined for binary numbers: #01, #0010, ...
  - a mechanical procedure for checking whether or not some bit of syntax is a binary number. Procedure:
    - is the syntax # ? If so, succeed. It is a binary number.
    - does the syntax end with "0"? If so, recursively check that the prefix is a binary number. If not, fail.
    - does the syntax end with "1"? If so, recursively check that the prefix is a binary number. If not, fail.
    - if the syntax is anything else, fail.
- Terminology:
  - we call # a base case because it contains no references to b, the thing being defined.
  - we call 0b and 1b inductive cases because they do contain references to b, the thing being defined.

Other Examples: Hex Numbers

h ::= # | h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7 | h8 | h9 | hA | hB | hC | hD | hE | hF

- Examples:
  - #001AAF
  - #FFB345
  - #
  - #1001

- Question: How can we tell the difference between constants like A, B, C, D and metavariables like h?
- Answer: h appears to the left of ::=
  - If a character or string does not appear to left of ::=, assume it is a constant

Other Examples: Mixed Numbers

h ::= # | h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7 | h8 | h9 | hA | hB | hC | hD | hE | hF
b ::= # | b0 | b1
n ::= hex h | bin b

- Examples of n:
  - hex #7352AAA, bin #00110, hex #00110
- Non-examples of n:
  - bin #7352AAA, bin (hex #888)

- Comment:
  - programming languages have lots of different kinds of syntax in them so we typically have to define many different metavariables
  - eg: java has numbers, strings, statements, expressions, types, class definitions, ...

Other Examples: Arithmetic Expressions

h ::= # | h0 | h1 | h2 | h3 | h4 | h5 | h6 | h7 | h8 | h9 | hA | hB | hC | hD | hE | hF
b ::= # | b0 | b1
n ::= hex h | bin b
e ::= num n | add(e,e) | mult(e, e)

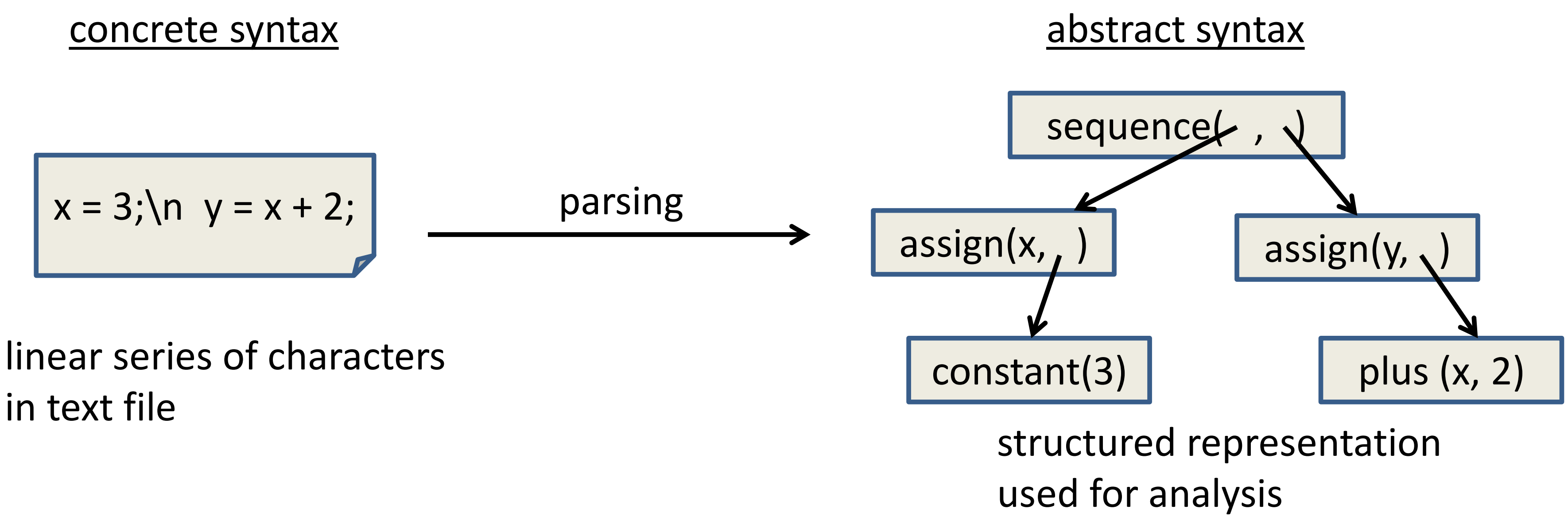
- Examples of e:
  - num (hex #7352AAA)
  - add (num (hex #00110), mult(num (bin #0), num (bin #10)))

- Non-examples of e:
  - num (hex (#FF + #AA))
  - bin #011
  - num #FF

- Comment:
  - we added some extra parentheses in the expressions above; these extra parens aren't part of the "official" syntax.
  - we use them to make the structure of an expression clear.

An Aside: Abstract vs. Concrete Syntax

- First phase of a typical compiler:



- Concrete syntax: a sequence of characters in a text file
- Abstract syntax: structured data that represents the key information needed for semantic analysis
  - discards whitespace, comments, tokens used to make programs easy to read
- COS 441 deals with analysis of abstract syntax
  - we don't worry about extra whitespace, parens, etc.; we care about structure
- COS 320 deals with concrete syntax and parsing

One more example: Unary Numbers

i ::= # | iS

zero

the number after i
(the Successor)

- Examples:
  - #S (one)
  - #SSSS (four)
  - #SS (two)

DENOTATIONAL SEMANTICS!

Denotational Semantics

- Given a binary number #10 you and I have a good idea of what it *means*. But how can we be sure we agree on the details?
- One way is translate it into a common language – the language of mathematics. That's what a denotational semantics does.

Denotational Semantics: Binary Numbers

- The denotation (ie: meaning) of an element of binary number syntax is a natural number
- We'll be precise by defining a mathematical function:

```
binsem ( # ) = 0

binsem (b0) = 2*(binsem(b))

binsem (b1) = 2*(binsem(b)) + 1
```

Denotational Semantics: Binary Numbers

- The denotation (ie: meaning) of an element of binary number syntax is a natural number
- We'll be precise by defining a mathematical function:

each result is
a natural number

binsem (#) = 0

binsem (b0) = 2*(binsem(b))

binsem (b1) = 2*(binsem(b)) + 1

each argument is a pattern drawn from the syntax definition:

b ::= # | b0 | b1

metavariables appearing in the argument position (like b)
are used in the right-hand side

Denotational Semantics: Hex Numbers

- The denotation (ie: meaning) of hex number syntax is also a natural number:

hexsem (#) = 0

hexsem (h0) = 16*(hexsem(h))

hexsem (h1) = 16*(hexsem(h)) + 1

hexsem (h2) = 16*(hexsem(h)) + 2

...

hexsem (hF) = 16*(hexsem(h)) + 15

each argument is
hex syntax

results are
natural numbers

Denotational Semantics: Mixed Numbers

- The denotation (ie: meaning) of mixed number syntax is also a natural number:

```
mixsem ( hex (h) ) = hexsem (h)

mixsem ( bin (b) ) = binsem (b)
```

Note: You may be seeing a bit of a trend here in that the results are always natural numbers but that is an artifact of the arithmetic examples I have chosen for this lecture.

In later lectures, we will see other kinds of results (sets, functions, heaps, etc.) in denotation functions

Denotational Semantics: Arithmetic Expressions

- The denotation (ie: meaning) of an element of arithmetic expression syntax is a natural number:

$$e \;::=\; \text{num } n \mid \text{add}(e,e) \mid \text{mult}(e, e)$$

$$\boxed{\begin{array}{l} \text{expsem (num (n)) = mixsem (n)} \\ \\ \text{expsem (add } (e_1,e_2) \text{) = expsem } (e_1) + \text{expsem } (e_2) \\ \\ \text{expsem (mult } (e_1,e_2) \text{) = expsem } (e_1) * \text{expsem } (e_2) \end{array}}$$

Denotational Semantics: Unary Numbers

- The denotation (ie: meaning) of an element of unary number syntax is a natural number:

i ::= # | iS

usem (#) = 0
usem (iS) = expsem (i) + 1

GOOD DEFINITIONS VS. BAD ONES (TOTALITY)

Good Definitions

- Can I write down just any equation I want to define the semantics of some piece of syntax?
- What are the criteria?

Good Definitions: Totality

- Can I write down just any equation I want to define the semantics of some piece of syntax?
- What are the criteria?
- Here's our semantics of binary numbers:

```
binsem ( # ) = 0

binsem (b0) = 2*(binsem(b))

binsem (b1) = 2*(binsem(b)) + 1
```

Is the definition total?
Are there any binary numbers whose semantics are left undefined?

Good Definitions: Totality

binsem (#) = 0

binsem (b0) = 2*(binsem(b))

binsem (b1) = 2*(binsem(b)) + 1

b ::= # | b0 | b1 <----- Recall the syntax

Good Definitions: Totality

binsem (#) = 0

binsem (b0) = 2*(binsem(b))

binsem (b1) = 2*(binsem(b)) + 1

A mathematical function defined on syntax is total when it produces a result for every element of the function domain.

b ::= # | b0 | b1 <----- Recall the syntax

Good Definitions: Totality

```
binsem ( # ) = 0

binsem (b0) = 2*(binsem(b))
```

Not Total (missing case for b1)

```
binsem ( # ) = 0

binsem ( #0 ) = 0

binsem ( b1 ) = 2*(binsem(b)) + 1

binsem ( b00 ) = 4*(binsem(b))

binsem ( b10 ) = 4*(binsem(b)) + 2
```

Total but a lot harder to check that we haven't missed any cases!

Sticking with cases that exactly match the syntax definition is typically a better bet but not always the most concise.

Good Definitions: Totality

convert less obvious total functions into obvious ones by introducing auxiliary functions:

binsem (#) = 0

binsem (#0) = 0

binsem (b1) = 2*(binsem(b)) + 1

binsem (b00) = 4*(binsem(b))

binsem (b10) = 4*(binsem(b)) + 2

binsem (#) = 0
binsem (b1) = 2*(binsem(b)) + 1
binsem (b0) = auxsem (b)

auxsem (#) = 0
auxsem (b0) = 4*binsem(b)
auxsem (b1) = 4*binsem(b) + 2

every function definition has exactly one case per syntactic alternative:

b ::= # | b0 | b1

GOOD DEFINITIONS VS. BAD ONES (INDUCTION)

Denotational Semantics: Binary Numbers

- What about this function:

```
binsem ( # ) = 0

binsem (b0) = binsem (b0)

binsem (b1) = binsem (b1)
```

- Is it total? What's wrong?

Denotational Semantics: Binary Numbers

- What about this function:

```
binsem ( # ) = 0

binsem (b0) = binsem (b0)

binsem (b1) = binsem (b1)
```

- Is it total? What's wrong?
  - binsem does not terminate on all inputs
    - it is not total
  - in addition, binsem is not an inductive function
    - inductive functions are functions that are guaranteed to terminate because recursive calls are made on smaller arguments and ...
    - the argument type is such that it contains no infinitely shrinking series of values
      - BNF syntax definitions never "shrink infinitely" --- valid syntax is built from base cases using a finite number of BNF rules

Inductive Functions

- What counts as "smaller"?
  - Functions with calls to proper syntactic subexpressions
    - aka: structural induction or induction on syntax

no calls always ok

multiple calls to subexpressions ok

identical calls bad

identical calls to larger expressions bad

b ::= # | b0 | b1

f(#) = ... (no calls) ...
f(#0) = ... (no calls) ...
f(b0) = ... f(b) ...
f(b1) = ... f(b) ... f (b) ...

inductive

f(b0) = ... f(b0) ...
f(b1) = ... f(b11) ...

not inductive

e ::= num (bin b) | add(e,e) | mult(e, e)

calls to other inductive functions ok

g(num (bin b)) = ... f (b) ...
g(add (e_1, e_2)) = ... g (e_1) ... g (e_2) ...
g(mult (e_1, e_2)) = ... g (e_1) ... g (e_2) ...

inductive

Inductive Functions

- What counts as "smaller"?
  - Functions are allowed to be mutually inductive:

```
binsem ( # ) = 0
binsem (b1) = 2*(binsem(b)) + 1
binsem (b0) = auxsem (b)

auxsem ( # ) = 0
auxsem ( b0 ) = 4*binsem(b)
auxsem ( b1 ) = 4*binsem(b) + 2
```

all calls in any of the right-hand sides are calls with smaller arguments than appear on the left-hand side of the corresponding equation.

Inductive Functions

- If you have taken COS 340 (or other math courses) you know that functions on the natural numbers can also be inductive
  - the right-hand side makes calls on smaller natural numbers
  - here is a mutually inductive definition of even and odd as functions from the natural numbers to booleans:

natural numbers: j ::= 0 | 1 | 2 | ...

even (0) = true
even (j+1) = not (odd (j))

odd (0) = false
odd (j+1) = not (even(j))

smaller number: $j < j + 1$

Inductive Functions

- Actually, inductive functions on natural numbers and inductive functions on syntax are the same thing:

i ::= # | iS

j ::= 0 | 1 | 2 | ...

Inductive Functions

- Actually, inductive functions on natural numbers and inductive functions on syntax are the same thing:

usem (#) = 0
usem (iS) = usem (i) + 1

i ::= # | iS

j ::= 0 | 1 | 2 | …

Inductive Functions

- Actually, inductive functions on natural numbers and inductive functions on syntax are the same thing:

usem (#) = 0
usem (iS) = usem (i) + 1

i ::= # | iS

j ::= 0 | 1 | 2 | ...

natrep (0) = #
natrep (j + 1) = (natrep (j))S

Inductive Functions

- Actually, inductive functions on natural numbers and inductive functions on syntax are the same thing:

usem (#) = 0
usem (iS) = usem (i) + 1

i ::= # | iS

even (#) = true
even (iS) = odd i

odd (#) = false
odd (iS) = even i

j ::= 0 | 1 | 2 | ...

even (0) = true
even (j+1) = odd j

odd (0) = false
odd (j+1) = even j

natrep (0) = #
natrep (j + 1) = (natrep (j))S

Summary

- Define syntax using BNF notation:

  b ::= # | b0 | b1

- Define denotation semantics using functions from syntax to mathematical objects like natural numbers, booleans, sets, or functions:

  binsem (#) = 0
  binsem (b0) = binsem(b)
  binsem (b1) = binsem(b) + 1

- Denotational functions are
  - total
    - f is total when for any x with an appropriate type, f(x) produces a result
    - note: sometimes denotational functions will not be total; in such cases we are intentionally saying that some bit of syntax is meaningless
  - inductive
    - functions are only called recursively on smaller arguments
    - a smaller argument is a proper subexpression of the original argument. This is called structural induction or induction on syntax

Reminders

- Sign up for the course email list:
  - https://lists.cs.princeton.edu/mailman/listinfo/cos441
- Read the course web site:
  - http://www.cs.princeton.edu/~dpw/cos441-11
- Start the first assignment (parts I and II)
  - due 1pm Tuesday Sept 27
  - see course web site for late policy
- Get ready for assignment 2 by downloading Haskell:
  - http://hackage.haskell.org/platform//
  - see course web site for learning materials