CS40032: Principles of Programming Languages
Module 07: Denotational Semantics

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

Source: *Denotational Semantics* by David A. Schmidt, 1997

Mar 11, 16, 18: 2019

Table of Contents

Introduction to Denotational Semantics

- **Overview**:
  - Syntax and Semantics
  - Approaches to Specifying Semantics
  - Sets, Semantic Domains, Domain Algebra, and Valuation Functions
  - Semantics of Expressions
  - Semantics of Assignments
  - Other Issues
- **References**:
  - David A. Schmidt, *Denotational Semantics – A Methodology for Language Development*, Allyn and Bacon, 1986
  - David Watt, *Programming Language Concepts and Paradigms*, Prentice Hall, 1990

Defining Programming Languages

Three main characteristics of programming languages:

- **Syntax**: What is the appearance and structure of its programs?
- **Semantics**: What is the meaning of programs?
  The static semantics tells us which (syntactically valid) programs are semantically valid (that is, which are type correct) and the dynamic semantics tells us how to interpret the meaning of valid programs.
- **Pragmatics**: What is the usability of the language? How easy is it to implement? What kinds of applications does it suit?

Uses of Semantic Specifications

Semantic specifications are useful for language designers to communicate to the implementors as well as to programmers.

A semantic specification is:

- A precise standard for a computer implementation:
  How should the language be implemented on different machines?
- User documentation:
  What is the meaning of a program, given a particular combination of language features?
- A tool for design and analysis:
  How can the language definition be tuned so that it can be implemented efficiently?
- An input to a compiler generator:
  How can a reference implementation be obtained from the specification?

Semantic Styles

Methods for Specifying Semantics

- **Operational Semantics**:
  - program = abstract machine program
  - can be simple to implement
  - hard to reason about
- **Axiomatic Semantics**:
  - program = set of properties
  - good for proving theorems about programs
  - somewhat distant from implementation
- **Denotational Semantics**:
  - program = mathematical denotation (typically, a function)
  - facilitates reasoning
  - not always easy to find suitable semantic domains

Programming Language of Binary Numerals with Addition

Examples:

- 110
- 010101
- $101 \oplus 111$

Grammar:

$$B = 0 \mid 1 \mid B0 \mid B1 \mid B \oplus B$$

- The empty string is not in the language
- We do not use parentheses in the abstract syntax although parentheses are needed to distinguish $(x \oplus y) \oplus z$ and $x \oplus (y \oplus z)$

Source: *COMP 745 Semantics of Programming Languages – Course Notes* by Peter Grogono, 2002.

Operational Semantics

An ***operational semantics*** *is a collection of rules that define a possible evaluation or execution of a program*

How programs are executed, or How the computer `operates`

Operational Semantics: Rules

$$\begin{aligned}
\epsilon \oplus x &\rightarrow x && (1)\\
x \oplus \epsilon &\rightarrow x && (2)\\
0x &\rightarrow x \quad (x \neq \epsilon) && (3)\\
x0 \oplus y0 &\rightarrow (x \oplus y)\ 0 && (4)\\
x1 \oplus y0 &\rightarrow (x \oplus y)\ 1 && (5)\\
x0 \oplus y1 &\rightarrow (x \oplus y)\ 1 && (6)\\
x1 \oplus y1 &\rightarrow (x \oplus y \oplus 1)\ 0 && (7)
\end{aligned}$$

PoPL-07

Partha Pratim Das

Styles

Syntax

Domains

Domains

Product

Sum

Rat

Algebras

Nat, Tr

String

Unit

Product Dom

Sum Dom

Lists

Function

Arrays

Lifted Domains

Recursive Fn

Denot. Defn.

Binary

Calculator

Operational Semantics: Example

Show that $101 \oplus 111 = 1100$.

Derivation:

$$
\begin{array}{rcll}
\epsilon \oplus x & \rightarrow & x & (1) \\
x \oplus \epsilon & \rightarrow & x & (2) \\
0x & \rightarrow & x \quad (x \neq \epsilon) & (3) \\
x0 \oplus y0 & \rightarrow & (x \oplus y)\ 0 & (4) \\
x1 \oplus y0 & \rightarrow & (x \oplus y)\ 1 & (5) \\
x0 \oplus y1 & \rightarrow & (x \oplus y)\ 1 & (6) \\
x1 \oplus y1 & \rightarrow & (x \oplus y \oplus 1)\ 0 & (7)
\end{array}
$$

$$
\begin{array}{rcl}
101 \oplus 111 & \Rightarrow & (10 \oplus 11 \oplus 1)\ 0 \\
 & \Rightarrow & ((1 \oplus 1)\ 1 \oplus 1)\ 0 \\
 & \Rightarrow & ((\epsilon \oplus \epsilon \oplus 1)\ 01 \oplus 1)\ 0 \\
 & \Rightarrow & ((\epsilon \oplus 1)\ 01 \oplus 1)\ 0 \\
 & \Rightarrow & (101 \oplus 1)\ 0 \\
 & \Rightarrow & (10 \oplus \epsilon \oplus 1)\ 00 \\
 & \Rightarrow & (10 \oplus 1)\ 00 \\
 & \Rightarrow & (1 \oplus \epsilon)\ 100 \\
 & \Rightarrow & 1100 \quad \square
\end{array}
$$

Operational Semantics: Example

Show that $1100 \oplus 1010 \Rightarrow 10110$ and $1101 \oplus 1001 \Rightarrow 10110$.

Derivation:

$$
\begin{array}{rcll}
\epsilon \oplus x & \rightarrow & x & (1) \\
x \oplus \epsilon & \rightarrow & x & (2) \\
0x & \rightarrow & x \quad (x \neq \epsilon) & (3) \\
x0 \oplus y0 & \rightarrow & (x \oplus y)\ 0 & (4) \\
x1 \oplus y0 & \rightarrow & (x \oplus y)\ 1 & (5) \\
x0 \oplus y1 & \rightarrow & (x \oplus y)\ 1 & (6) \\
x1 \oplus y1 & \rightarrow & (x \oplus y \oplus 1)\ 0 & (7)
\end{array}
$$

$$
\begin{array}{rcl}
1100 \oplus 1010 & \Rightarrow & (110 \oplus 101)\ 0 \\
& \Rightarrow & (11 \oplus 10)\ 10 \\
& \Rightarrow & (1 \oplus 1)\ 110 \\
& \Rightarrow & (\epsilon \oplus \epsilon \oplus 1)\ 0110 \\
& \Rightarrow & (\epsilon \oplus 1)\ 0110 \\
& \Rightarrow & 10110 \quad \square \\
1101 \oplus 1001 & \Rightarrow & (110 \oplus 100 \oplus 1)\ 0 \\
& \Rightarrow & ((11 \oplus 10)\ 0 \oplus 1)\ 0 \\
& \Rightarrow & ((1 \oplus 1)\ 10 \oplus 1)\ 0 \\
& \Rightarrow & ((\epsilon \oplus \epsilon \oplus 1)\ 010 \oplus 1)\ 0 \\
& \Rightarrow & ((\epsilon \oplus 1)\ 010 \oplus 1)\ 0 \\
& \Rightarrow & (1010 \oplus 1)\ 0 \\
& \Rightarrow & (101 \oplus \epsilon)\ 10 \\
& \Rightarrow & 10110 \quad \square
\end{array}
$$

Operational Semantics

- **Operational Semantics**: specifies the behavior of a programming language by defining a simple *abstract machine* for it
  - This machine is *abstract* in the sense that it uses the terms of the language as its machine code, rather than some low-level microprocessor instruction set.
  - A *state* of the machine is just a *term*, and
  - The machine's behavior is defined by a *transition function* that, for each state:
    - either gives the next state by performing a step of simplification on the term or
    - declares that the machine has halted
  - The meaning of a term t can be taken to be the final state that the machine reaches when started with t as its initial state

Axiomatic Semantics

In ***axiomatic semantics*** *we set a meaning of binary numerals through a set of laws, or axioms, that binary numerals must satisfy*

Equality: There are (at least) two possible interpretations of a formula such as $x = y$.

- *syntactic equality*: We might be comparing the appearance of x and y ($101 = 000101$ is false), or
- *semantic equality*: We might be comparing their meanings ($2 + 2 = 4$)

Axiomatic Semantics: Semantic Equality

$$
\begin{aligned}
0 \oplus 0 &= 0 && (1)\\
0 \oplus 1 &= 1 && (2)\\
1 \oplus 1 &= 10 && (3)\\
0x &= x && (4)\\
x \oplus y &= y \oplus x && (5)\\
x \oplus (y \oplus z) &= (x \oplus y) \oplus z && (6)\\
x0 \oplus y0 &= (x \oplus y)\ 0 && (7)\\
x1 \oplus y0 &= (x \oplus y)\ 1 && (8)\\
x1 \oplus y1 &= (x \oplus y \oplus 1)\ 0 && (9)
\end{aligned}
$$

Axiomatic Semantics: Example

$$
\begin{aligned}
11 \oplus 10 &= (1 \oplus 1)1 \\
&= (10)1 \\
&= 101
\end{aligned}
$$

Note: We can interpret this deduction as $3 + 2 = 5$ but – note carefully! – the semantics does not say this: all it says is that the string $11 \oplus 10$ is equivalent to the string 101

Axiomatic Semantics: Example

Show that $101 \oplus 111 = 1100$.

Proof:

$$\begin{array}{rcll}
0 \oplus 0 & = & 0 & (1) \\
0 \oplus 1 & = & 1 & (2) \\
1 \oplus 1 & = & 10 & (3) \\
0x & = & x & (4) \\
x \oplus y & = & y \oplus x & (5) \\
x \oplus (y \oplus z) & = & (x \oplus y) \oplus z & (6) \\
x0 \oplus y0 & = & (x \oplus y)\ 0 & (7) \\
x1 \oplus y0 & = & (x \oplus y)\ 1 & (8) \\
x1 \oplus y1 & = & (x \oplus y \oplus 1)\ 0 & (9)
\end{array}$$

$$\begin{array}{rcl}
101 \oplus 111 & = & (10 \oplus 11 \oplus 1)\ 0 \\
& = & ((1 \oplus 1)\ 1 \oplus 1)\ 0 \\
& = & (101 \oplus 1)\ 0 \\
& = & (101 \oplus 01)\ 0 \\
& = & (10 \oplus 0 \oplus 1)\ 00 \\
& = & (10 \oplus 1)\ 00 \\
& = & (10 \oplus 01)\ 00 \\
& = & (1 \oplus 0)\ 100 \\
& = & 1100 \quad \square
\end{array}$$

Axiomatic Semantics: Example

Show that $1100 \oplus 1010 \Rightarrow 10110$ and $1101 \oplus 1001 \Rightarrow 10110$.

Proof:

$$
\begin{aligned}
0 \oplus 0 &= 0 && (1)\\
0 \oplus 1 &= 1 && (2)\\
1 \oplus 1 &= 10 && (3)\\
0x &= x && (4)\\
x \oplus y &= y \oplus x && (5)\\
x \oplus (y \oplus z) &= (x \oplus y) \oplus z && (6)\\
x0 \oplus y0 &= (x \oplus y)\ 0 && (7)\\
x1 \oplus y0 &= (x \oplus y)\ 1 && (8)\\
x1 \oplus y1 &= (x \oplus y \oplus 1)\ 0 && (9)
\end{aligned}
$$

$$
\begin{aligned}
1100 \oplus 1010 &= (110 \oplus 101)\ 0\\
&= (11 \oplus 10)\ 10\\
&= (1 \oplus 1)\ 110\\
&= 10110 \quad \square\\
1101 \oplus 1001 &= (110 \oplus 100 \oplus 1)\ 0\\
&= ((11 \oplus 10)\ 0 \oplus 1)\ 0\\
&= ((1 \oplus 1)\ 10 \oplus 1)\ 0\\
&= (1010 \oplus 1)\ 0\\
&= (1010 \oplus 01)\ 0\\
&= (101 \oplus 0)\ 10\\
&= (101 \oplus 00)\ 10\\
&= (10 \oplus 0)\ 110\\
&= (10 \oplus 00)\ 110\\
&= (1 \oplus 0)\ 0110\\
&= 10110 \quad \square
\end{aligned}
$$

Axiomatic Semantics: Facts

Exercise: Why is the empty string used in the operational semantics but not in the axiomatic semantics?

Exercise: Why do we not obtain the operational semantics simply by changing $=$ to $\rightarrow$ in the axiomatic semantics?

Axiomatic Semantics

- **Axiomatic Semantics**: takes a more direct approach to these laws: instead of
  - first defining the behaviors of programs (by giving some operational or denotational semantics like 101 means number 5) and then
  - deriving laws from this definition (like $3 + 2 = 5$),

  axiomatic methods take the laws themselves as the definition of the language
- The meaning of a term is just what can be proved about it
- The beauty of axiomatic methods is that they focus attention on the process of reasoning about programs
- Leads to the powerful ideas such as *invariants* – *Design by Contract*

Axiomatic Semantics: Data Structures

- **Axiomatic Semantics**: Domains, Functions and Axioms
  - Domains:

Nat	the natural numbers
Stack	of natural numbers
Bool	boolean values

  - Functions:

$newStack$:	$() \rightarrow Stack$
$push$:	$(Nat, Stack) \rightarrow Stack$
pop :	$Stack \rightarrow Stack$
top :	$Stack \rightarrow Nat$
$empty$:	$Stack \rightarrow Bool$

Axiomatic Semantics: Data Structures

- **Axiomatic Semantics**: Domains, Functions and Axioms
  - Axioms:

$$
\begin{array}{lcl}
push(N, S) & \neq & S \\
pop(S) & \neq & S, \text{ if } empty(S) = false \\
pop(S) & = & error, \text{ if } empty(S) = true \\
pop(newStack()) & = & error \\
pop(push(N, S)) & = & S \\
top(push(N, S)) & = & N \\
top(S) & = & error, \text{ if } empty(S) = true \\
top(newStack()) & = & error \\
empty(push(N, S)) & = & false \\
empty(newStack()) & = & true
\end{array}
$$

where $N \in Nat$ and $S \in Stack$

Axiomatic Semantics: Data Structures

Write the axiomatic semantics for:

- Array
- Priority Queue
- Queue
- Singly Linked List
- Binary Search Tree

Denotational Semantics

A ***denotational semantics*** *is a system that provides a denotation in a mathematical domain for each string of a language*

- The numeral 101 represents the natural number 5
- Formally – *the denotation of 101 is 5*

In denotational semantics:

- **Semantic Function**: $\mathcal{M} : \mathbf{B} \rightarrow \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers
- Enclose **syntactic objects** (in this example, members of **B**) in $[[.]]$
- The formal way of writing *the denotation of 101 is 5* is:

$$\mathcal{M}[[101]] = 5$$

Denotational Semantics: Semantic Function

$$
\begin{aligned}
\mathcal{M}[[0]] &= 0 && (1)\\
\mathcal{M}[[1]] &= 1 && (2)\\
\mathcal{M}[[x0]] &= 2 * \mathcal{M}[[x]] && (3)\\
\mathcal{M}[[x1]] &= 2 * \mathcal{M}[[x]] + 1 && (4)\\
\mathcal{M}[[x \oplus y]] &= \mathcal{M}[[x]] + M[[y]] && (5)
\end{aligned}
$$

Note: The 0 or 1 on the left is a binary numeral (member of **B**); the 0 or 1 on the right is a natural number (member of $\mathbb{N}$)

Denotational Semantics: Example

Show that $\mathcal{M}[[101 \oplus 111]] = 12 = \mathcal{M}[[1100]]$.

Proof:

$$
\begin{aligned}
\mathcal{M}[[0]] &= 0 && (1)\\
\mathcal{M}[[1]] &= 1 && (2)\\
\mathcal{M}[[x0]] &= 2 * \mathcal{M}[[x]] && (3)\\
\mathcal{M}[[x1]] &= 2 * \mathcal{M}[[x]] + 1 && (4)\\
\mathcal{M}[[x \oplus y]] &= \mathcal{M}[[x]] + M[[y]] && (5)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{M}[[101]] &= 2 * \mathcal{M}[[10]] + 1\\
&= 2 * (2 * \mathcal{M}[[1]]) + 1\\
&= 2 * (2 * 1) + 1 = 5\\
\mathcal{M}[[111]] &= 2 * \mathcal{M}[[11]] + 1\\
&= 2 * (2 * \mathcal{M}[[1]] + 1) + 1\\
&= 2 * (2 * 1 + 1) + 1 = 7\\
\mathcal{M}[[1100]] &= 2 * \mathcal{M}[[110]]\\
&= 2 * 2 * \mathcal{M}[[11]]\\
&= 2 * 2 * (2 * \mathcal{M}[[1]] + 1)\\
&= 2 * 2 * (2 * 1 + 1) = 12\\
\mathcal{M}[[101 \oplus 111]] &= \mathcal{M}[[101]] + \mathcal{M}[[111]]\\
&= 5 + 7 = 12\\
&= \mathcal{M}[[1100]] \quad \square
\end{aligned}
$$

Denotational Semantics: Example

Show that $\mathcal{M}[[1100 \oplus 1010]] = 22 = \mathcal{M}[[10110]]$.

Proof:

$$
\begin{array}{rcll}
\mathcal{M}[[0]] & = & 0 & (1)\\
\mathcal{M}[[1]] & = & 1 & (2)\\
\mathcal{M}[[x0]] & = & 2 * \mathcal{M}[[x]] & (3)\\
\mathcal{M}[[x1]] & = & 2 * \mathcal{M}[[x]] + 1 & (4)\\
\mathcal{M}[[x \oplus y]] & = & \mathcal{M}[[x]] + M[[y]] & (5)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M}[[1100]] & = & 2 * \mathcal{M}[[110]]\\
& = & 2 * 2 * \mathcal{M}[[11]]\\
& = & 2 * 2 * (2 * \mathcal{M}[[1]] + 1)\\
& = & 2 * 2 * (2 * 1 + 1) = 12\\
\mathcal{M}[[1010]] & = & 2 * \mathcal{M}[[101]]\\
& = & 2 * (2 * \mathcal{M}[[10]] + 1)\\
& = & 2 * (2 * 2 * \mathcal{M}[[1]] + 1)\\
& = & 2 * (2 * 2 * 1 + 1) = 10\\
\mathcal{M}[[10110]] & = & 2 * \mathcal{M}[[1011]]\\
& = & 2 * (2 * \mathcal{M}[[101]] + 1)\\
& = & 2 * (2 * (2 * \mathcal{M}[[10]] + 1) + 1)\\
& = & 2 * (2 * (2 * 2 * \mathcal{M}[[1]] + 1) + 1)\\
& = & 2 * (2 * (2 * 2 * 1 + 1) + 1) = 22\\
\mathcal{M}[[1100 \oplus 1010]] & = & \mathcal{M}[[1100]] + \mathcal{M}[[1010]]\\
& = & 12 + 10 = 22\\
& = & \mathcal{M}[[10110]] \quad \square
\end{array}
$$

Denotational Semantics: Example

Show that $\mathcal{M}[[1101 \oplus 1001]] = 22 = \mathcal{M}[[10110]]$.

Proof:

$$\begin{array}{rcll}
\mathcal{M}[[0]] & = & 0 & (1) \\
\mathcal{M}[[1]] & = & 1 & (2) \\
\mathcal{M}[[x0]] & = & 2 * \mathcal{M}[[x]] & (3) \\
\mathcal{M}[[x1]] & = & 2 * \mathcal{M}[[x]] + 1 & (4) \\
\mathcal{M}[[x \oplus y]] & = & \mathcal{M}[[x]] + M[[y]] & (5)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{M}[[1101]] & = & 2 * \mathcal{M}[[110]] + 1 \\
 & = & 2 * 2 * \mathcal{M}[[11]] + 1 \\
 & = & 2 * 2 * (2 * \mathcal{M}[[1]] + 1) + 1 \\
 & = & 2 * 2 * (2 * 1 + 1) + 1 = 13 \\
\mathcal{M}[[1001]] & = & 2 * \mathcal{M}[[100]] + 1 \\
 & = & 2 * 2 * \mathcal{M}[[10]] + 1 \\
 & = & 2 * 2 * 2 * \mathcal{M}[[1]] + 1 \\
 & = & 2 * 2 * 2 * 1 + 1 = 9 \\
\mathcal{M}[[10110]] & = & 2 * \mathcal{M}[[1011]] \\
 & = & 2 * (2 * \mathcal{M}[[101]] + 1) \\
 & = & 2 * (2 * (2 * \mathcal{M}[[10]] + 1) + 1) \\
 & = & 2 * (2 * (2 * 2 * \mathcal{M}[[1]] + 1) + 1) \\
 & = & 2 * (2 * (2 * 2 * 1 + 1) + 1) = 22 \\
\mathcal{M}[[1101 \oplus 1001]] & = & \mathcal{M}[[1101]] + \mathcal{M}[[1001]] \\
 & = & 13 + 9 = 22 \\
 & = & \mathcal{M}[[10110]] \quad \square
\end{array}$$

Denotational Semantics: Example

Exercise: Leading zeroes do not affect the value of a binary numeral. For example, 00101 denotes the same natural number (5) as 101.
Prove that, for any binary numeral x, $\mathcal{M}[[0x]] = \mathcal{M}[[x]]$ $\square$

Hint: Use induction on the length of x

Denotational Semantics: Example

Exercise: Show that the operational semantics is correct with respect to the denotational semantics

Exercise: Show that the axioms of the Axiomatic Semantics are logical consequences of the Denotational Semantics.
Hint: Show that the denotation of lhs and rhs of every axiom match each other.

Can you do the reverse?

Denotational Semantics

- **Denotational Semantics**: takes a more abstract view of meaning: instead of just a sequence of machine states, the meaning of a term is taken to be some mathematical object, such as a number or a function
- Giving denotational semantics for a language consists of:
  - finding a collection of *semantic domains* and then
  - defining an *interpretation function* mapping terms into elements of these domains
- The search for appropriate semantic domains for modeling various language features has given rise to *domain theory*
- Significantly relies on λ-Calculus

Denotational Semantics: Data Structures

Write the denotational semantics for:

- Array
- Stack
- Queue
- Priority Queue
- Singly Linked List
- Binary Search Tree

Semantic Styles: Comparison

- **Operational Semantics**: tells us how to execute a program, but does not tell us either the meaning of the program or any properties that it may possess
- **Axiomatic Semantics**: describes properties that programs must have, but does not say what the program means or how to execute it
- **Denotational Semantics**: tells us what program means, but does not (necessarily) tell us how to execute it

	Meaning	Properties	Execution
Operational Semantics	No	No	Yes
Axiomatic Semantics	No	Yes	No
Denotational Semantics	Yes	No	No

Syntax

Concrete and Abstract Syntax

- How to parse "4 * 2 + 1"?
- Abstract syntax is compact but ambiguous

Expr	::=	Num
	|	Expr Op Expr
Op	::=	'+' | '-' | '*' | '/'

- Concrete syntax is unambiguous, but verbose

Expr	::=	Expr LowOp Expr
	|	Term
Term	::=	Term HighOp Factor
	|	Factor
Factor	::=	Num
	|	'(' Expr ')'
LowOp	::=	'+' | '-'
HighOp	::=	'*' | '/'

Semantic Domains

Set, Functions, and Domains

- A set is a collection: it can contain numbers, persons, other sets, or (almost) anything one wishes:
  - { 1, {1, 2, 4}, 4}
  - { red, yellow, gray }
  - {}
- A function is like *black box* that accepts an object as its input and then transforms it in some way to produce another object as output. We must use an *external approach* to characterize functions. Sets are ideal for formalizing the method. (*Extensional and Intentional Views*)
- The sets that are used as value spaces in programming language semantics are called *semantic domains*. Semantic domains may have a different structure than a set, and in practice not all of the sets and set building operations are needed for building domains.

Common Sets

1. Natural numbers: $\mathcal{N} = \{0, 1, 2, \ldots\}$
2. Integers: $\mathcal{Z} = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$
3. Rational numbers: $\mathcal{Q} = \{\, x : \text{ for } p \in \mathcal{Z} \text{ and } q \in \mathcal{Z}, q > 0, gcd(p, q) = 1, x = p/q\}$
4. Real numbers: $\mathcal{R} = \{x : x \text{ is a point on the line } \cdots\ -2\ -1\ 0\ 1\ 2 \cdots\}$
5. Characters: $\mathcal{C} = \{x : x \text{ is a character}\}$
6. Truth values (Booleans): $\mathcal{B} = \{\text{ true, false }\}$

Basic Domains

- Primitive domains:
  - Natural numbers $\mathcal{N}$
  - Boolean values $\mathcal{B}$
  - Floating point numbers $\mathcal{F}$
- Compound domains:
  - Product domains $\mathcal{A} \times \mathcal{B}$
  - Sum domains $\mathcal{A} + \mathcal{B}$
  - Function domains $\mathcal{A} \rightarrow \mathcal{B}$
- Lifted domains:
  - Lifted domains add a special value $\perp$ (*bottom*) that denotes non-termination or *no value at all*. Including as a value is an alternative to using a theory of partial functions.
  - Lifted domains are written $A_\perp$, where $A_\perp = A \cup \{\perp\}$

Product domains

- The product construction takes two component domains and builds a domain of tuples from the components
- The product domain builder $\times$ builds the domain $A \times B$, a collection whose members are ordered pairs of the form (a, b), for $a \in A$ and $b \in B$.
- The operation builders for the product domain include the two disassembly operations:
  $fst : A \times B \to A$ which takes an argument $(a, b) \in A \times B$ and produces its first component $a \in A$, that is,
  $fst(a, b) = a$
  $snd : A \times B \to B$ which takes an argument $(a, b) \in A \times B$ and produces its second component $b \in B$, that is,
  $snd(a, b) = b$
- The assembly operation is the ordered pair builder: if a is an element of A, and b is an element of B, then (a, b) is an element of $A \times B$

Product domains

- The product construction can be generalized to work with any collection of domains $A_1, A_2, \cdots, A_n$, for any $n > 0$
- We write $(x_1, x_2, ..., x_n)$ to represent an element of $A_1 \times A_2 \times \cdots \times A_n$
- The subscripting operations *fst* and *snd* generalize to a family of n operations: for each i from 1 to n, $\downarrow i$ denotes the operation such that $(a_1, a_2, \cdots, a_n) \downarrow i = a_i$

Sum domains

- For domains A and B, the disjoint union builder $+$ builds the domain $A + B$, a collection whose members are the elements of A and the elements of B, **labeled to mark their origins**
- The classic representation of this labeling is the ordered pair $(zero, a)$ for an $a \in A$ and (one, b) for a $b \in B$.
- The associated operation builders include two assembly operations:
  $inA : A \to A + B$ which takes an $a \in A$ and labels it as originating from A; that is, $inA(a) = (zero, a)$, using the pair representation described above.
  $inB : B \to A + B$ which takes a $b \in B$ and labels it as originating from B, that is, $inB(b) = (one, b)$.

Sum domains

- The *type tags* that the assembly operations place onto their arguments are put to good use by the disassembly operation, the cases operation, which combines an operation on A with one on B to produce a disassembly operation on the sum domain.
- If d is a value from $A + B$ and $f(x) = e_1$ and $g(y) = e_2$ are the definitions of $f : A \to C$ and $g : B \to C$, then:

$$(cases\ d\ of\ isA(x) \to e_1\ []\ isB(y) \to e_2\ end)$$

represents a value in C.

Sum domains

- The following properties hold:

$$(cases\ inA(a)\ of\ \ isA(x) \rightarrow e_1\ []\ isB(y) \rightarrow e_2\ \ end) =$$

$$[a/x]e_1 = f(a)$$

and

$$(cases\ inB(b)\ of\ \ isA(x) \rightarrow e_1\ []\ isB(y) \rightarrow e_2\ \ end) =$$

$$[b/y]e_2 = g(b)$$

- The cases operation checks the tag of its argument, removes it, and gives the argument to the proper operation.
- Sums of an arbitrary number of domains can be built. We write $A_1 + A_2 + \ldots + A_n$ to stand for the disjoint union of domains $A_1, A_2, \ldots, A_n$. The operation builders generalize in the obvious way.

Semantic Algebras

- The format for representing semantic domains is called *semantic algebra* and defines a grouping of a set with the fundamental operations on the set.
- This format is used because it:
  - Clearly states the structure of a domain and how its elements are used by the functions,
  - Encourages the development of *standard* algebra *modules* or *kits* that can be used in a variety of semantics definitions,
  - Makes it easier to analyze a semantic definition concept by concept,
  - Makes it straightforward to alter a semantic definition by replacing one semantic algebra with another.
- The expression $e1 \rightarrow e2[]e3$ is the *choice function*, which has as its value $e2$ if $e1 = \mathit{true}$ and $e3$ if $e1 = \mathit{false}$.

Domain Rat

- Domain $\underline{\text{Rat}} = (\mathcal{Z} \times \mathcal{Z})_\perp$
- Operations

  makeRat :: $\mathcal{Z} \to \mathcal{Z} \to \underline{\text{Rat}}$

  makeRat $= \lambda p.\lambda q.(q = 0) \to \perp [](p, q)$

  addRat :: $\underline{\text{Rat}} \to \underline{\text{Rat}} \to \underline{\text{Rat}}$

  addRat $= \underline{\lambda}(p_1, q_1).\underline{\lambda}(p_2, q_2).((p_1 * q_2) + (p_2 * q_1), q_1 * q_2)$

  Since the possibility of an undefined rational exists, the addrat operation checks both of its arguments for definedness before performing the addition of the two fractions.

  mulRat :: $\underline{\text{Rat}} \to \underline{\text{Rat}} \to \underline{\text{Rat}}$

  mulRat $= \underline{\lambda}(p_1, q_1).\underline{\lambda}(p_2, q_2).(p_1 * p_2, q_1 * q_2)$

Haskell Implementation

module Rational (Rational, makerat, addrat, mulrat)

where

$data\ Rational = Rat\ Int\ Int$

makerat :: $Int- > Int- > Rational$
makerat $p\ q$
$|q == 0 =$ error "$Rational:\ \ division\ by\ zero$"
$|otherwise = Rat\ p\ q$

addrat :: $Rational- > Rational- > Rational$
addrat =
$\backslash(Rat\ p1\ q1)- > \backslash(Rat\ p2\ q2)- > Rat\ ((p1\ *q2) + (p2\ *q1))\ (q1\ *q2)$

mulrat :: $Rational- > Rational- > Rational$
mulrat $= \backslash(Rat\ p1\ q1)- > \backslash(Rat\ p2\ q2)- > Rat\ (p1\ *p2)\ (q1\ *q2)$

instance Show Rational where – tell Haskell how to print rationals

$show\ (Rat\ p\ q) = "(" + +show\ p + +"," + +show\ q + +")"$

Semantic Algebras

Primitive Domain – Natural Numbers

- Domain
  $\text{Nat} = \mathcal{N}$
- Operations
  $zero : Nat$
  $one : Nat$
  $two : Nat$
  $\ldots$
  $plus : Nat \times Nat \to Nat$
  $minus : Nat \times Nat \to Nat$
  $times : Nat \times Nat \to Nat$
  $div : Nat \times Nat \to Nat$

Primitive Domain – Natural Numbers

- *Note*:
  - $x\ minus\ y = zero$, if $x < y$
  - $six\ div\ two = three$
  - $seven\ div\ two = three$
  - $seven\ div\ zero = error$
  - $two\ plus\ error = error$
  - We need to handle *no value* or *error*. We may include this in $\mathcal{N}$ and extend all operations to handle it.
- *Note*: The error element is not always included in a primitive domain, and we will always make it clear when it is.

Primitive Domain – Truth Values

- Domain Tr $= \mathcal{B}$
- Operations
  $true : Tr$
  $false : Tr$
  $not : Tr \to Tr$
  $or : Tr \times Tr \to Tr$
  $(\_ \to \_[]\_) : Tr \times D \times D \to D$,
  for a previously defined domain D

The truth values algebra has two constants – *true* and *false*. Operation *not* is logical negation, and *or* is logical disjunction. The last operation is the choice function. It uses elements from another domain in its definition.For values $m, n \in D$, it is defined as:

$(true \to m \; [] \; n) = m$

$(false \to m \; [] \; n) = n$

Primitive Domain – Truth Values

- $((not(false))\ or\ false$
- $(true\ or\ false) \rightarrow (seven\ div\ three)\ []\ zero$
- $not(not\ true) \rightarrow false\ []\ false\ or\ true$

Primitive Domain – Natural Numbers

- Domain Nat $= \mathcal{N}$
- Operations

$zero : Nat$

$one : Nat$

$two : Nat$

$\ldots$

$plus : Nat \times Nat \rightarrow Nat$

$minus : Nat \times Nat \rightarrow Nat$

$times : Nat \times Nat \rightarrow Nat$

$div : Nat \times Nat \rightarrow Nat$

$equals : Nat \times Nat \rightarrow Tr$

$lessthan : Nat \times Nat \rightarrow Tr$

$greaterthan : Nat \times Nat \rightarrow Tr$

Primitive Domain – Natural Numbers

- Example:
  $\mathit{not}(\mathit{four}\ \mathit{equals}(\mathit{one}\ \mathit{plus}\ \mathit{three})) \rightarrow$
  $(\mathit{one}\ \mathit{greaterthan}\ \mathit{zero})\ []\ ((\mathit{five}\ \mathit{times}\ \mathit{two})\ \mathit{lessthan}\ \mathit{zero})$

Primitive Domain – String

- Domain

  $String$ = the strings formed from the elements of $\mathcal{C}$ (including an *error* string)

- Operations

  $A, B, C, ..., Z : String$

  $empty : String$

  $error : String$

  $concat : String \times String \to String$

  $length : String \to Nat$

  $substr : String \times Nat \times Nat \to String$

- Note:

  $substr(''ABC'', one, two) = ''AB''$

  $substr(''ABC'', one, four) = error$

  $substr(''ABC'', six, two) = error$

  $concat(error, ''ABC'') = error$

  $length(error) = zero$

Primitive Domain – One element domain

- Domain *Unit*, the domain containing only one element
- Operations
  () : *Unit*

This degenerate algebra is useful for theoretical reasons; we will also make use of it as an alternative form of error value. The domain contains exactly one element, (). *Unit* is used whenever an operation needs a dummy argument.

Primitive Domain – Computer store locations

- Domain *Location*, the address space in a computer store
- Operations

  $\mathit{first\_locn} : \mathit{Location}$

  $\mathit{next\_locn} : \mathit{Location} \rightarrow \mathit{Location}$

  $\mathit{equal\_locn} : \mathit{Location} \times \mathit{Location} \rightarrow \mathit{Tr}$

  $\mathit{lessthan\_locn} : \mathit{Location} \times \mathit{Location} \rightarrow \mathit{Tr}$

Compound Domain – Payroll information

A person's name, payrate, and hours worked

- Domain
  $Payroll\_record = String \times Rat \times Rat$
- Operations
  $new\_employee : String \rightarrow Payroll\_record$
  $update\_payrate : Rat \times Payroll\_record \rightarrow Payroll\_record$
  $update\_hours : Rat \times Payroll\_record \rightarrow Payroll\_record$
  $compute\_pay : Payroll\_record \rightarrow Rat$

Compound Domain – Payroll information

A person's name, payrate, and hours worked

- Domain $Payroll\_record = String \times Rat \times Rat$
- Operations
  $new\_employee : String \to Payroll\_record$
  $new\_employee(name) = (name, minimum\_wage, \mathbf{0})$
  where $minimum\_wage \in Rat$ is some fixed value from Rat and $\mathbf{0}$ is the Rat value $(makerat(0)(1))$

  $update\_payrate : Rat \times Payroll\_record \to Payroll\_record$
  $update\_payrate(pay, employee) = (employee \downarrow 1, pay, employee \downarrow 3)$

  $update\_hours : Rat \times Payroll\_record \to Payroll\_record$
  $update\_hours(hours, employee) =$
  $(employee \downarrow 1, employee \downarrow 2, hours\ addrat\ employee \downarrow 3)$

Compound Domain – Payroll information

Example:

$compute\_pay(update\_hours(makerat(35, 1), new\_employee(''J.Doe'')))$

Compound Domain – Payroll information

Example:

$compute\_pay(update\_hours(makerat(35, 1), new\_employee("J.Doe")))$
$= compute\_pay(update\_hours(makerat(35, 1), ("J.Doe", minimum\_wage, 0)))$
$= compute\_pay(("J.Doe", minimum\_wage, 0) \downarrow 1, ("J.Doe", minimum\_wage, 0) \downarrow 2, makerat(35, 1)\ addrat\ ("J.Doe", minimum\_wage, 0) \downarrow 3)$
$= compute\_pay("J.Doe", minimum\_wage, makerat(35, 1)\ addrat\ 0)$
$= minimum\_wage\ multrat\ makerat(35, 1)$

Compound Domain – Revised Payroll information

A person's name, payrate, and hours worked

- Domain
  $Payroll\_rec = String \times (Day + Night) \times Rat$
  where $Day = Rat$ and $Night = Rat$
  (The names Day and $Night$ are aliases for two occurrences of Rat. We use $dwage \in Day$ and $nwage \in Night$ in the operations that follow.)
- Operations
  $new\_employee : String \to Payroll\_rec$
  $update\_payrate : Rat \times Payroll\_rec \to Payroll\_rec$
  $move\_to\_dayshift : Payroll\_rec \to Payroll\_rec$
  $move\_to\_nightshift : Payroll\_rec \to Payroll\_rec$
  $update\_hours : Rat \times Payroll\_rec \to Payroll\_rec$
  $compute\_pay : Payroll\_rec \to Rat$

Disjoint Union

Revised payroll information

- Domain $Payroll\_rec = String \times (Day + Night) \times Rat$
  where $Day = Rat \; and \; Night = Rat$
  (The names Day and $Night$ are aliases for two occurrences of Rat. We use $dwage \in Day$ and $nwage \in Night$ in the operations that follow.)
- Operations
  $newemp : String \rightarrow Payroll\_rec$
  $newemp(name) = (name, inDay(minimum\_wage), 0)$

  $move\_to\_dayshift : Payroll\_rec \rightarrow Payroll\_rec$
  $move\_to\_dayshift(employee) = (employee \downarrow 1,$
  $(cases \; (employee \downarrow 2) \; of \; isDay(dwage) \rightarrow inDay(dwage)$
  $[] \; isNight(nwage) \rightarrow inDay(nwage) \; end),$
  $employee \downarrow 3)$

  $move\_to\_nightshift : Payroll\_rec \rightarrow Payroll\_rec$
  $move\_to\_nightshift(employee) = (employee \downarrow 1,$
  $(cases \; (employee \downarrow 2) \; of \; isDay(dwage) \rightarrow inNight(dwage)$
  $[] \; isNight(nwage) \rightarrow inNight(nwage) \; end),$
  $employee \downarrow 3)$
  $update\_hours : Rat \times Payroll\_record \rightarrow Payroll\_record$
  ...

Disjoint Union

Revised payroll information

- Operations
  $compute\_pay : Payroll\_record \rightarrow Rat$
  $compute\_pay(employee) = (cases\ (employee \downarrow 2)\ of$
  $isDay(dwage) \rightarrow dwage\ multrat\ (employee \downarrow 3)$
  $[]\ isNight(nwage) \rightarrow (nwage\ multrat\ makerat(3, 2))\ multrat\ (employee \downarrow 3)$

- Example:
  If $jdoe = newemp(''J.Doe'') = (''J.Doe'', inDay(minimum\_wage), 0)$ and $jdoe\_thirty = update\_hours(makerat(30, 1), jdoe)$, then

  $compute\_pay(jdoe\_thirty)$

Disjoint Union

Example:
If $jdoe = newemp(''J.Doe'') = (''J.Doe'', inDay(minimum\_wage), 0)$ and $jdoe\_thirty = update\_hours(makerat(30, 1), jdoe)$, then

$compute\_pay(jdoe\_thirty)$
$= (cases\ jdoe\_thirty \downarrow 2\ of$
$isDay(wage) \rightarrow wage\ multrat\ (jdoe\_thirty \downarrow 3)$
$[]\ isNight(wage) \rightarrow (wage\ multrat\ makerat(3, 2))multrat\ (jdoe\_thirty \downarrow 3)\ end)$
$= (cases\ inDay(minimum\_wage)\ of$
$isDay(wage) \rightarrow wage\ multrat\ makerat(30, 1)$
$[]\ isNight(wage) \rightarrow wage\ multrat\ makerat(3, 2)\ multrat\ makerat(30, 1)\ end)$
$= minimum\_wage\ multrat\ makerat(30, 1)$

Disjoint Union: Representing Truth Values

- Domain
  $Tr = TT + FF$
  $where TT = Unit \ and \ FF = Unit$
- Operations
  $true : Tr$
  $\quad true = inTT()$
  $false : Tr$
  $\quad false = inFF()$
  $not : Tr \rightarrow Tr$
  $\quad not(t) = cases \ t \ of \ isTT() \rightarrow inFF() \ [] \ isFF() \rightarrow inTT() \ end$
  $or : Tr \times Tr \rightarrow Tr$
  $\quad or(t, u) = cases \ t \ of$
  $\qquad isTT() \rightarrow inTT()$
  $\qquad [] \ isFF() \rightarrow (cases \ u \ of \ isTT() \rightarrow inTT() \ [] \ isFF() \rightarrow inFF() \ end)$
  $\quad end$
- Choice Function
  $$(t \rightarrow e1 \ [] \ e2) = (cases \ t \ of \ isTT() \rightarrow e1 \ [] \ isFF() \rightarrow e2 \ end)$$

Finite Lists

For a domain D with an error element, the collection of finite lists of elements from D can be defined as a disjoint union.

$$D^* = Unit + D + (D \times D) + (D \times (D \times D)) + \ldots$$

$Unit$ represents those lists of length zero (namely the empty list), D contains those lists containing one element, $D \times D$ contains those lists of two elements, and so on.

- Domain
  D^*
- Operations
  $nil : D^*$
  $\quad nil = inUnit()$
  $cons : D \times D^* \to D^*$
  $\quad cons(d, l) = cases\ l\ of$
  $\qquad isUnit() \to inD(d)$
  $\qquad [\!]\ isD(y) \to inDXD(d, y)$
  $\qquad [\!]\ isDXD(y) \to inDX(DXD)(d, y)$
  $\qquad [\!]\ \cdots\ end$

Finite Lists

- $hd : D^* \to D$
  $hd(l) = cases\ l\ of$
  $isUnit() \to error$
  $[]\ isD(y) \to y$
  $[]\ isDXD(y) \to fst(y)$
  $[]\ isDX(DXD)(y) \to fst(y)$
  $[]\ \cdots\ end$

  $tl : D^* \to D^*$
  $tl(l) = cases\ l\ of$
  $isUnit() \to inUnit()$
  $[]\ isD(y) \to inUnit()$
  $[]\ isDXD(y) \to inD(snd(y))$
  $[]\ isDX(DXD)(y) \to inDXD(snd(y))$
  $[]\ \cdots\ end$

  $null : D^* \to Tr$
  $null(l) = cases\ l\ of$
  $isUnit() \to true$
  $[]\ isD(y) \to false$
  $[]\ isDXD(y) \to false$
  $[]\ \cdots\ end$

Finite Lists – Tuple Representation

- The domain has an infinite number of components and the cases expressions have an infinite number of choices; yet the domain and codomain operations are still mathematically well defined.
- To implement the algebra on a machine, representations for the domain elements and operations must be found.
- Since each domain element is a tagged tuple of finite length, a list can be represented as a tuple.
- The tuple representations lead to simple implementations of the operations.

Function Space

- **Assembly Operation**: *Function Space Builder* collects the functions from a domain A to a codomain B
  - If e is an expression containing occurrences of an identifier x, such that whenever a value $a \in A$ replaces the occurrences of x in e, the value $[a/x]e \in B$ results, then $(\lambda x.e)$ is an element in $A \to B$.
  - The form $(\lambda x.e)$ is called an *Abstraction*. We often give names to abstractions, say $f = (\lambda x.e)$, or $f(x) = e$, where f is some name not used in e.
  - For example, the function $plus\ two(n) = n\ plus\ two$ is a member of $Nat \to Nat$ because $n\ plus\ two$ is an expression that has a unique value in Nat when n is replaced by an element of Nat.
  - We will usually abbreviate a nested abstraction $(\lambda x.(\lambda y.e))$ to $(\lambda x.\lambda y.e)$
  - The binding of argument to binding identifier works the expected way with abstractions:
    $(\lambda n.n\ plus\ two)one = [one/n]n\ plus\ two\ = one\ plus\ two$

Function Space

- **Disassembly Operation**: *Function Application*

$$\_(\_) : (A \rightarrow B) \times A \rightarrow B$$

which takes an $f \in A \rightarrow B$ and an $a \in A$ and produces $f(a) \in B$

Function Space

Examples:

1. $(\lambda m.(\lambda n.n\ times\ n)(m\ plus\ two))(one)$
2. $(\lambda m.\lambda n.(m\ plus\ m)\ times\ n)(one)(three)$
3. $(\lambda m.(\lambda n.n\ plus\ n)(m)) = (\lambda m.m\ plus\ m)$
4. $(\lambda p.\lambda q.p\ plus\ q)(r\ plus\ one) = (\lambda q.(r\ plus\ one)\ plus\ q)$

Function Space

Examples:

1. $(\lambda m.(\lambda n.n\ times\ n)(m\ plus\ two))(one)$
   $= (\lambda n.n\ times\ n)(one\ plus\ two)$
   $= (one\ plus\ two)\ times\ (one\ plus\ two)$
   $= three\ times\ (one\ plus\ two) = three\ times\ three\ = nine$

2. $(\lambda m.\lambda n.(m\ plus\ m)\ times\ n)(one)(three)$
   $= (\lambda n.(one\ plus\ one)\ times\ n)(three)$
   $= (\lambda n.two\ times\ n)(three)$
   $= two\ times\ three = six$

3. $(\lambda m.(\lambda n.n\ plus\ n)(m)) = (\lambda m.m\ plus\ m)$

4. $(\lambda p.\lambda q.p\ plus\ q)(r\ plus\ one) = (\lambda q.(r\ plus\ one)\ plus\ q)$

Dynamic Arrays

- Domain:
  $Array = Nat \rightarrow A$, where A is a domain with an error element
- Operations:
  $newarray : Array$
  $\quad newarray = \lambda n.error$
  An empty array is represented by the constant $newarray$. It is a function and it maps all of its index arguments to error
  $access : Nat \times Array \rightarrow A$
  $\quad access(n, r) = r(n)$
  $update : Nat \times A \times Array \rightarrow Array$
  $\quad update(n, v, r) = [n \mapsto v]r$

where the update expression $[n \mapsto v]r$ is a function that abbreviates for $(\lambda m.m \; equals \; n \rightarrow v \; [] \; r(m))$. That is, $([n \mapsto v]r)(n) = v$, and $([n \mapsto v]r)(m) = r(m)$ when $m \neq n$.

Dynamic Arrays

Prove:

- *for any* $m_0, n_0 \in Nat$ *such that* $m_0 \neq n_0$,
  $access(m_0, update(n_0, v, r))$
  $= r(m_0)$
- $access(n_0,\ update(n_0, v, r))$
  $= v$

Dynamic Arrays

- *for any* $m_0, n_0 \in Nat$ *such that* $m_0 \neq n_0$,

$$
\begin{aligned}
& access(m_0, update(n_0, v, r)) \\
& = (update(n_0, v, r))(m_0) \\
& \qquad \text{(by definition of access)} \\
& = ([n_0 \mapsto v]r)(m_0) \\
& \qquad \text{(by definition of update)} \\
& = (\lambda m.m \ equals \ n_0 \rightarrow v \ [] \ r(m))(m_0) \\
& \qquad \text{(by definition of function updating)} \\
& = m_0 \ equals \ n_0 \rightarrow v \ [] \ r(m_0) \\
& \qquad \text{(by function application)} \\
& = false \rightarrow v \ [] \ r(m_0) \\
& = r(m_0)
\end{aligned}
$$

Dynamic Arrays

- $access(n_0,\ update(n_0, v, r))$

$$\begin{aligned}
&(update(n_0, v, r))(n_0) \\
&= ([n0 \mapsto v]r)(n_0) \\
&= (\lambda m.m\ equals\ n_0 \rightarrow v\ []\ r(m))(n_0) \\
&= n_0\ equals\ n_0 \rightarrow v\ []\ r(n_0) \\
&= true \rightarrow v\ []\ r(n_0) \\
&= v
\end{aligned}$$

Dynamic Arrays

Dynamic array with curried operations

- Domain:
  $Array = Nat \rightarrow A$
- Operations:
  $newarray : Array$
  $\quad newarray = \lambda n.error$
  $access : Nat \rightarrow Array \rightarrow A$
  $\quad access = \lambda n.\lambda r.r(n)$
  $update : Nat \rightarrow A \rightarrow Array \rightarrow Array$
  $\quad update = \lambda n.\lambda v.\lambda r.[n \mapsto v]r$

Lifted Domains and Strictness

- **Assembly Operation**: For domain A, the *Lifting domain builder* $()_\perp$ creates the domain $A_\perp$, a collection of the members of A plus an additional distinguished element $\perp$

  The elements of A in $A_\perp$ are called *proper elements*; $\perp$ is the *improper element*

Lifted Domains and Strictness

- **Disassembly Operation**: The disassembly operation builder converts an operation on A to one on $A_\perp$:
  - For $(\lambda x.e) : A \rightarrow B_\perp$, $(\underline{\lambda} x.e) : A_\perp \rightarrow B_\perp$ is defined as
    $(\underline{\lambda} x.e)\perp = \perp$
    $(\underline{\lambda} x.e)a = [a/x]e \text{ for } a \neq \perp$

    Note that $\underline{\lambda}$ with underline – for lifted operation
  - An operation that maps a $\perp$ argument to a $\perp$ answer is called *strict*. Operations that map $\perp$ to a proper element are called *non-strict*
  - Hence,
    $(\underline{\lambda} m.zero)((\underline{\lambda} n.one)\perp)$
    $= (\underline{\lambda} m.zero)\perp$, (by strictness)
    $= \perp$

    On the other hand, $(\lambda p.zero) : Nat_\perp \rightarrow Nat_\perp$ is *non-strict*, and: $(\lambda p.zero)((\underline{\lambda} n.one)\perp)$
    $= [(\underline{\lambda} n.one)\perp/p]zero$, (by the definition of application)
    $= zero$

Lifted Domains and Strictness

Let us use the following abbreviation:

$$(\mathit{let}\ x = e_1\ \mathit{in}\ e_2)\ \mathit{for}\ (\underline{\lambda} x.e_2)e_1$$

- $\mathit{let}\ m\ = (\lambda x.\mathit{zero})\bot\ \mathit{in}\ m\ \mathit{plus}\ \mathit{one}$
  $= \mathit{let}\ m = \mathit{zero}\ \mathit{in}\ m\ \mathit{plus}\ \mathit{one}$
  $= \mathit{zero}\ \mathit{plus}\ \mathit{one}\ = \mathit{one}$
- $\mathit{let}\ m = \mathit{one}\ \mathit{plus}\ \mathit{two}\ \mathit{in}\ \mathit{let}\ n = (\underline{\lambda} p.m)\bot\ \mathit{in}\ m\ \mathit{plus}\ n$
  $= \mathit{let}\ m = \mathit{three}\ \mathit{in}\ \mathit{let}\ n = (\underline{\lambda} p.m)\bot\ \mathit{in}\ m\ \mathit{plus}\ n$
  $= \mathit{let}\ n = (\underline{\lambda} p.\mathit{three})\bot\ \mathit{in}\ \mathit{three}\ \mathit{plus}\ n$
  $= \mathit{let}\ n = \bot\ \mathit{in}\ \mathit{three}\ \mathit{plus}\ n$ (by call-by-value)
  $= \bot$

Lifted Domains and Strictness

Unsafe Access of Unsafe Values

- Domain:
  $Unsafe = Array_{\perp}$
  where $Array = Nat \rightarrow Tr'$ and $Tr' = (B \cup \{error\})_{\perp}$
- Operations:
  $new\_unsafe : Unsafe$
  $\quad new\_unsafe = newarray = \lambda n.error$
  $access\_unsafe : Nat_{\perp} \rightarrow Unsafe \rightarrow Tr'$
  $\quad access\_unsafe = \underline{\lambda} n.\underline{\lambda} r.(access\ n\ r)$
  Operation access_unsafe must check the definedness of its arguments n and r before it passes them on to *access*
  $update\_unsafe : Nat_{\perp} \rightarrow Tr' \rightarrow Unsafe \rightarrow Unsafe$
  $\quad update\_unsafe = \underline{\lambda} n.\lambda t.\underline{\lambda} r.(update\ n\ t\ r)$
  The operation *update_unsafe* is similarly paranoid, but an improper truth value may be stored into an array

Lifted Domains and Strictness

Example: Evaluation of an expression where
$let\ not' = \underline{\lambda} t.not(t))$:

$$
\begin{aligned}
& let\ start\_array = new\_unsafe \\
& in\ update\_unsafe(one\ plus\ two)(not'(\bot))(start\_array) \\
& = let\ start\_array = newarray \\
& \quad in\ update\_unsafe(one\ plus\ two)(not'(\bot))(start\_array) \\
& = let\ start\_array = (\lambda n.error) \\
& \quad in\ update\_unsafe(one\ plus\ two)(not'(\bot))(start\_array) \\
& = update\_unsafe(one\ plus\ two)(not'(\bot))(\lambda n.error) \\
& = update\_unsafe(three)(not'(\bot))(\lambda n.error) \\
& = update(three)(not'(\bot))(\lambda n.error) \\
& = [three \mapsto not'(\bot)](\lambda n.error) \\
& = [three \mapsto \bot](\lambda n.error)
\end{aligned}
$$

Recursive Functions Definitions

A recursive definition may not uniquely define a function. Consider

$$q(x) = x \; equals \; zero \rightarrow one \; [] \; q(x \; plus \; one)$$

which apparently is: $\mathcal{N} \rightarrow \mathcal{N}_\perp$. The following functions all satisfy q's definition in the sense that they have exactly the behavior required by the equation:

- $f_1(x) = one$, if $x = zero$
  $= \perp$, otherwise. OR
  $f_1(x) = \lambda x.(x \; equals \; zero \rightarrow one \; [] \; \perp)$
- $f_2(x) = one$, if $x = zero$
  $= two$, otherwise. OR
  $f_2(x) = \lambda x.(x \; equals \; zero \rightarrow one \; [] \; two)$
- $f_3(x) = \lambda x.(one)$

and there are infinitely many others.

Recursive Functions Definitions

Given

$$q(x) = x \ equals \ zero \rightarrow one \ [] \ q(x \ plus \ one)$$

Prove that $\forall n \in Nat$

1. $n \ equals \ zero \rightarrow one \ [] \ f_1(n \ plus \ one) = f_1(n) = q(n)$
   where $f_1(x) = \lambda x.(x \ equals \ zero \rightarrow one \ [] \ \perp)$
2. $n \ equals \ zero \rightarrow one \ [] \ f_2(n \ plus \ one) = f_2(n) = q(n)$
   where $f_2(x) = \lambda x.(x \ equals \ zero \rightarrow one \ [] \ two)$
3. $n \ equals \ zero \rightarrow one \ [] \ f_3(n \ plus \ one) = f_3(n) = q(n)$
   where $f_3(x) = \lambda x.(one)$

Recursive Functions Definitions

1. $$\begin{aligned} & n \textit{ equals zero} \rightarrow \textit{one} \; [] \; f_1(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \\ &\quad (\lambda x.(x \textit{ equals zero} \rightarrow \textit{one} \; [] \; \bot))(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \\ &\quad ((n \textit{ plus one}) \textit{ equals zero} \rightarrow \textit{one} \; [] \; \bot) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \; \bot \\ &= f_1(n) = \lambda x.(x \textit{ equals zero} \rightarrow \textit{one} \; [] \; \bot) \end{aligned}$$
2. $$\begin{aligned} & n \textit{ equals zero} \rightarrow \textit{one} \; [] \; f_2(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \\ &\quad (\lambda x.(x \textit{ equals zero} \rightarrow \textit{one} \; [] \; \textit{two}))(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \\ &\quad ((n \textit{ plus one}) \textit{ equals zero} \rightarrow \textit{one} \; [] \; \textit{two}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \; \textit{two} \\ &= f_2(n) = \lambda x.(x \textit{ equals zero} \rightarrow \textit{one} \; [] \; \textit{two}) \end{aligned}$$
3. $$\begin{aligned} & n \textit{ equals zero} \rightarrow \textit{one} \; [] \; f_3(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \\ &\quad (\lambda x.(\textit{one}))(n \textit{ plus one}) \\ &= n \textit{ equals zero} \rightarrow \textit{one} \; [] \; \textit{one} \\ &= \textit{one} \\ &= f_3(n) = \lambda x.(\textit{one}) \end{aligned}$$

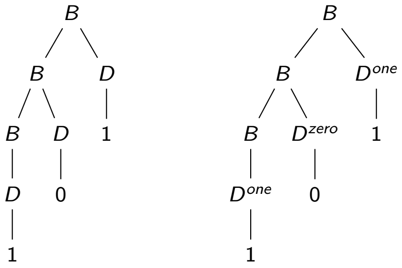
Structure of Denotational Definitions

Basic Structure of Denotational Definitions

- **Format for Denotational Definitions**
  - *Abstract Syntax*: Appearance of a language
  - *Semantic Algebra*: Meaning of a language
  - *Valuation Function*: Connects *Abstract Syntax* with *Semantic Algebra*
- The denotational semantics of two simple languages presented

Valuation Function

- The valuation function maps a language's abstract syntax structures to meanings drawn from semantic domains
- The domain of a valuation function is the set of derivation trees of a language
- The valuation function is defined structurally
- It determines the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree



$B \in Binary\_numeral$
$D \in Binary\_digit$
$B ::= BD \mid D$
$D ::= 0 \mid 1$
$D[[0]] = zero$
$D[[1]] = one$

Valuation Function

- The valuation function assigns a meaning to the tree by assigning meanings to its subtrees
- Use two valuation functions: $\mathbf{D} : \mathit{Binary\_digit} \rightarrow \mathit{Nat}$, which maps binary digits to their meanings, and $\mathbf{B} : \mathit{Binary\_numeral} \rightarrow \mathit{Nat}$, which maps binary numerals to their meanings
- Distinct valuation functions make the semantic definition easier to formulate and read

$$\begin{array}{c} D \\ | \\ 0 \end{array} \Rightarrow \begin{array}{c} \mathbf{D}(D^{zero}) \\ | \\ 0 \end{array} \Rightarrow \mathbf{D}[[0]] = \mathit{zero}$$

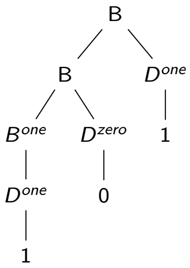
$$\begin{array}{c} D \\ | \\ 1 \end{array} \Rightarrow \begin{array}{c} \mathbf{D}(D^{one}) \\ | \\ 1 \end{array} \Rightarrow \mathbf{D}[[1]] = \mathit{one}$$

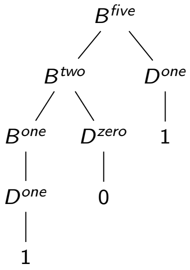
Valuation Function

Similarly,
$\mathbf{B}[[D]] = \mathbf{D}[[D]]$ for $B := D$

Next for $B := BD$, we get

$\mathbf{B}[[BD]] = (\mathbf{B}[[B]]\ \mathit{times}\ \mathit{two})\ \mathit{plus}\ \mathbf{D}[[D]]$





Valuation Function – Example

$\mathbf{B}[[101]]$
$= (\mathbf{B}[[10]]\ \textit{times two})\ \textit{plus}\ \mathbf{D}[[1]]$
$= (((\mathbf{B}[[1]]\ \textit{times two})\ \textit{plus}\ \mathbf{D}[[0]])\ \textit{times two})\ \textit{plus}\ \mathbf{D}[[1]]$
$= (((\mathbf{D}[[1]]\ \textit{times two})\ \textit{plus}\ \mathbf{D}[[0]])\ \textit{times two})\ \textit{plus}\ \mathbf{D}[[1]]$
$= (((\textit{one times two})\ \textit{plus zero})\ \textit{times two})\ \textit{plus one}$
$= \textit{five}$

Format of Denotational Definition

- *Abstract Syntax* :
  $B \in Binary\_numeral$
  $D \in Binary\_digit$
  $B ::= BD \mid D$
  $D ::= 0 \mid 1$
- *Semantic Algebras* :
  I. Natural numbers
  Domain
  $Nat = \mathcal{N}$
  Operations
  $zero,\ one,\ two,\ \cdots : Nat$
  $plus,\ times:\ Nat \times Nat \rightarrow Nat$
- *Valuation Functions* :
  $\mathbf{B} : Binary\_numeral \rightarrow Nat$
  $\mathbf{B}[[BD]] = (\mathbf{B}[[B]]\ times\ two)\ plus\ \mathbf{D}[[D]]$
  $\mathbf{B}[[D]] = \mathbf{D}[[D]]$
  $\mathbf{D} : Binary\_digit \rightarrow Nat$
  $\mathbf{D}[[0]] = zero$
  $\mathbf{D}[[1]] = one$

Ternary Numerals

Write the denotational semantics for ternary numerals:

$T \in Ternary\_numeral$
$D \in Ternary\_digit$
$T ::= TD \mid D$
$D ::= 0 \mid 1 \mid 2$
$D[[0]] = zero$
$D[[1]] = one$
$D[[2]] = two$

Evaluate:

$T[[201]]$

Decimal Numerals

Write the denotational semantics for decimal numerals:

$N \in Decimal\_numeral$
$W \in Whole\_Decimal$
$F \in Fractional\_Decimal$
$D \in Decimal\_digit$
$N ::= W.F$
$W ::= WD \mid D$
$F ::= FD \mid D$
$D ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$D[[0]] = zero$
$D[[1]] = one$
$D[[2]] = two$
$D[[3]] = three$
$D[[4]] = four$
$D[[5]] = five$
$D[[6]] = six$
$D[[7]] = seven$
$D[[8]] = eight$
$D[[9]] = nine$
$N[[.]] = point$

Evaluate: $N[[237.92]]$

A Calculator Language

- A calculator is a good example of a processor that accepts programs in a simple language as input and produces simple, tangible output
- The programs are entered by pressing buttons on the device, and the output appears on a display screen
- It has an inexpensive model with a single *memory cell* for retaining a numeric value
- There is also a conditional evaluation feature, which allows the user to enter a form of if-then-else expression

Simple Calculator

display				
ON	**OFF**	**LASTANSWER**		
1	**2**	**3**	**(**	**+**
4	**5**	**6**	**)**	*****
7	**8**	**9**	**IF**	**,**
	0			**TOTAL**

A Calculator Language

- *Simple Calculator*

display				
ON	**OFF**	**LASTANSWER**		
1	**2**	**3**	**(**	**+**
4	**5**	**6**	**)**	*****
7	**8**	**9**	**IF**	**,**
	0			**TOTAL**

- Sample Session:
  press *ON*
  press $(4 + 1\ 2) * 2$
  press *TOTAL* (*the calculator prints* 32)
  press $1 + LASTANSWER$
  press *TOTAL* (*the calculator prints* 33)
  press $IF\ LASTANSWER\ +\ 1, 0, 2 +\ 4$
  press *TOTAL* (*the calculator prints* 6)
  press *OFF*
- The calculator's memory cell automatically remembers the value of the previous expression calculated so the value can be used in a later expression
- The IF and , keys are used to build a conditional expression that chooses its second or third argument to evaluate based upon whether the value of the first is zero or nonzero

A Calculator Language

- *Abstract Syntax* :

  $P \in Program$
  $S \in Expr\_sequence$
  $E \in Expression$
  $N \in Numeral$

  $P ::= ON\ S$
  $S ::= E\ TOTAL\ S \mid E\ TOTAL\ OFF$
  $E ::= E_1 + E_2 \mid E_1 * E_2 \mid IF\ E_1, E_2, E_3 \mid LASTANSWER \mid (E) \mid N$

- *Semantic Algebras* :

  I. Truth values
  Domain
  $t \in Tr = B$
  Operations
  true, false: Tr
  II. Natural numbers
  Domain
  $n \in Nat$
  Operations
  zero, one, two, ... : Nat
  $plus, times : Nat \times Nat \to Nat$
  $equals : Nat \times Nat \to Tr$

A Calculator Language

- Valuation Functions:

  $P : Program \rightarrow Nat^*$ (sequence of outputs / display)
  $P ::= ON\ S$

  $S : Expr\_sequence \rightarrow Memory\_cell \rightarrow Nat^*$, where $Memory\_cell = Nat$
  $S ::= E\ TOTAL\ S \mid E\ TOTAL\ OFF$

  - Every expression is evaluated in the context of the value in the memory cell.
  - The value in the memory cell is updated as a side-effect and is not directly modeled in terms of the valuation functions.
  - An expression sequence is one or more expressions, separated by occurrences of TOTAL, terminated by the OFF key.

  $E : Expression \rightarrow Nat \rightarrow Nat$
  $E ::= E_1 + E_2 \mid E_1 * E_2 \mid IF\ E_1, E_2, E_3 \mid LASTANSWER \mid (E) \mid N$

  $N : Numeral \rightarrow Nat$

A Calculator Language

- Valuation functions:

  $\mathbf{P} : \mathit{Program} \rightarrow \mathit{Nat}^*$

  $\mathbf{P}[[ON\ S]] = \mathbf{S}[[S]](\mathit{zero})$ (memory cell is initialized to *zero*)

  $\mathbf{S} : \mathit{Expr\_sequence} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}^*$

  $\mathbf{S}[[E\ \ TOTAL\ \ S]](n) = \mathit{let}\ n' = \mathbf{E}[[E]](n)\ \mathit{in}\ n'\ \mathit{cons}\ \mathbf{S}[[S]](n')$

  $\mathbf{S}[[E\ \ TOTAL\ \ OFF]](n) = \mathbf{E}[[E]](n)\ \mathit{cons}\ \mathit{nil}$

  $\mathbf{E} : \mathit{Expression} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}$

  $\mathbf{E}[[E_1 + E_2]](n) = \mathbf{E}[[E_1]](n)\ \mathit{plus}\ \mathbf{E}[[E_2]](n)$

  $\mathbf{E}[[E_1 * E_2]](n) = \mathbf{E}[[E_1]](n)\ \mathit{times}\ \mathbf{E}[[E_2]](n)$

  $\mathbf{E}[[IF\ \ E_1, E_2, E_3]](n) = \mathbf{E}[[E_1]](n)\ \mathit{equals}\ \mathit{zero} \rightarrow$
  $\quad \mathbf{E}[[E_2]](n)\ []\ \mathbf{E}[[E_3]](n)$

  $\mathbf{E}[[LASTANSWER]](n) = n$

  $\mathbf{E}[[(E)]](n) = \mathbf{E}[[E]](n)$

  $\mathbf{E}[[N]](n) = \mathbf{N}[[N]]$

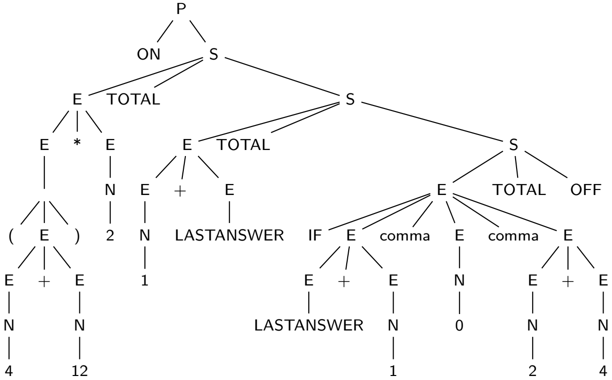
  $\mathbf{N} : \mathit{Numeral} \rightarrow \mathit{Nat}$ (maps numeral $\mathcal{N}$ to corresponding $n \in \mathit{Nat}$)

Note: $(\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$ *for* $(\underline{\lambda} x.e_2)e_1$

A Calculator Language

Sample Session:

press ON
press $(4 + 1\ 2) * 2$
press $TOTAL$ (*the calculator prints* 32)
press $1 + LASTANSWER$
press $TOTAL$ (*the calculator prints* 33)
press $IF\ LASTANSWER\ +\ 1, 0, 2 +\ 4$
press $TOTAL$ (*the calculator prints* 6)
press OFF



A Calculator Language

- We can list the corresponding actions that the calculator would take for $\mathbf{S}[[E\ TOTAL\ S]]$:
  1. Evaluate $[[E]]$ using cell n, producing value n'
  2. Print n' out on the display.
  3. Place n' into the memory cell
  4. Evaluate the rest of the sequence $[[S]]$ using the cell
- Note how each of these four steps are represented in the semantic equation:
  1. is handled by the expression $\mathbf{E}[[E]](n)$, binding it to the variable n'
  2. is handled by the expression $n'\,cons \cdots$ (out on the display)
  3. and 4. are handled by the expression $\mathbf{S}[[S]](n')$

A Calculator Language

- Simplify the calculator program:
  $\mathbf{P}[[\mathit{ON}\ 2+1\ \mathit{TOTAL}\ \mathit{IF}\ \mathit{LASTANSWER}, 2, 0\ \mathit{TOTAL}\ \mathit{OFF}]]$

A Calculator Language

PoPL-07

Partha Pratim Das

- Simplification of a sample calculator program:

$$\begin{aligned}
&\mathbf{P}[[ON\ 2+1\ TOTAL\ IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]] \\
&= \mathbf{S}[[2+1\ TOTAL\ IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]](zero) \\
&= let\ n' = \mathbf{E}[[2+1]](zero) \\
&\qquad\qquad in\ n'\ cons\ \mathbf{S}[[IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]](n') \\
&= three\ in\ n'\ cons\ \mathbf{S}[[IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]](n') \\
&= three\ cons\ \mathbf{S}[[IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]](three) \\
&= three\ cons\ (\mathbf{E}[[IF\ LASTANSWER, 2, 0]](three)\ cons\ nil)
\end{aligned}$$

$$\begin{aligned}
&\mathbf{E}[[IF\ LASTANSWER, 2, 0]](three) \\
&= \mathbf{E}[[LASTANSWER]](three)\ equals\ zero \rightarrow \mathbf{E}[[2]](three)\ []\ \mathbf{E}[[0]](three) \\
&= three\ equals\ zero \rightarrow two\ []\ zero \\
&= false \rightarrow two\ []\ zero \\
&= zero
\end{aligned}$$

$$\begin{aligned}
&\mathbf{P}[[ON\ 2+1\ TOTAL\ IF\ LASTANSWER, 2, 0\ TOTAL\ OFF]] \\
&= three\ cons\ (zero\ cons\ nil)
\end{aligned}$$