4. Denotational Semantics

Before paper and scissors

4. Denotational Semantics

Basic Idea

Defining Denotational Semantics

Domain Constructions

How to Define Semantics

Informal/non-specifications

- ***Reference Implementation:*** *run program through compiler*
- ***Manuals:*** *give descriptions in natural language*

Formal specification

- ***Denotational Semantics:*** *map terms to (mathematical) values*
- ***Operational Semantics:*** *give rules for evaluating terms*
- ***Axiomatic Semantics:*** *capture meaning through logical formulas*

Denotational Semantics

A denotational semantics maps

programs *(= ASTs)*

to

denotations *(= values in some* ***semantic domain****)*

Valuation function

$[\![\cdot]\!] \ : \ \textit{abstract syntax} \ \rightarrow \ \textit{semantic domain}$

Valuation function in Elm

```
eval : AST -> Value
```

Why Abstract Syntax in Semantics?

Principle of Compositionality

- *Syntactic structure determines semantic content*
- *The meaning of an expression is obtained as a function of the meanings of its subexpressions.*

Compositional Semantics

$$[\![f(e_1, \ldots, e_k)]\!] = [\![f]\!]([\![e_1]\!], \ldots, [\![e_k]\!])$$

Semantic Domains

***Semantic domain*:** The set of possible meanings of a program

What is a meaning? — It depends on the language!

Language	Meaning
Boolean expressions	Boolean value
Arithmetic expressions	Integer
Imperative language	State transformation
SQL	Relation
Logo	Picture
Music notation	Sound
Labanotation	Human movement

4. Denotational Semantics

Basic Idea

Defining Denotational Semantics

Domain Constructions

Denotational Semantics in Three Steps

Example in Elm:

1. Define the ***abstract syntax*** T
   Set of abstract syntax trees

```
type AST = ...
```

2. Define the ***semantic domain*** V
   Set of semantic values

```
type (alias) Value = ...
```

3. Define the ***valuation function*** $[\![\cdot]\!] : T \to V$
   Mapping from ASTs to semantic values
   *a.k.a. the "**semantic function**"*

```
sem : AST -> Value
```

Denotational Semantics in Elm

1. Abstract Syntax

```
type Expr = Num Int
          | Neg Expr
          | Plus  Expr Expr
          | Times Expr Expr
```

2. Semantic Domain

```
type alias Value = Int
```

3. Semantic Function

```
sem : Expr -> Value
sem e = case e of
    Num i         -> i
    Neg e1        -> -(sem e1)
    Plus e1 e2    -> sem e1 + sem e2
    Times e1 e2   -> sem e1 * sem e2
```

Exercise …

Question 1

(1) Define the abstract syntax for the following language, (2) identify a semantic domain, and (3) define the denotational semantics.

$$b \in bexpr \ ::= \ \mathsf{T} \mid \mathsf{F} \mid b \lor b \mid b \land b \mid \neg b$$

1. Abstract Syntax

```
type BExpr = T | F
           | Or BExpr BExpr
           | And BExpr BExpr
           | Not BExpr
```

2./3. Semantic Domain & Function

```
sem : BExpr -> Bool
sem e = case e of
          T           ->  True
          F           ->  False
          Or b1 b2    ->  sem b1 || sem b2
          And b1 b2   ->  sem b1 && sem b2
          Not b       ->  not (sem b)
```

4. Denotational Semantics

Basic Idea

Defining Denotational Semantics

Domain Constructions

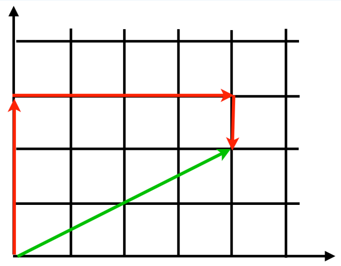
Type Constructors for Domain Building

Domain Construction	Type Representation
Product of domains	Use tuple types
Union of domains	Define data type
Adding errors	Use `Maybe` (or add `Error` constructor)
Adding state	Use function types

Example: Move Language

Language describing movements on a 2D plane

- a ***step*** is a horizontal or vertical vector
- a ***movement*** is a sequence of steps

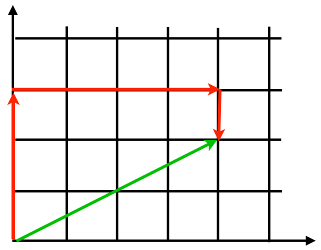


Concrete Syntax

$move ::= \texttt{go up}\ num \mid \texttt{go right}\ num \mid move\ \texttt{;}\ move$

```
go up 3; go right 4; go up −1
```

Abstract Syntax & Semantic Domains



Concrete Syntax

move ::= `go up` *num* | `go right` *num* | *move* `;` *move*

```
go up 3; go right 4; go up -1
```

```
Seq (Seq (GoUp 3) (GoRight 4)) (GoUp -1)
Seq (GoUp 3) (Seq (GoRight 4) (GoUp -1))
```

1. Abstract Syntax

```
type Move = GoUp Int
          | GoRight Int
          | Seq Move Move
```

2. Semantic Domain

```
type alias Pos = (Int,Int)
```

Semantic Function

1. Abstract Syntax

```
type Move = GoUp Int
          | GoRight Int
          | Seq Move Move
```

2. Semantic Domain

```
type alias Pos = (Int,Int)
```

3. Semantic Function

```
addPos : Pos -> Pos -> Pos
addPos (u,v) (x,y) = (u+x,v+y)

sem : Move -> Pos
sem m = case m of
    GoUp n    -> (0,n)
    GoRight n -> (n,0)
    Seq m1 m2 -> addPos (sem m1) (sem m2)
```

Exercise …

Question 2

Define the semantics of the Move language using the following abstract syntax.

Abstract Syntax

```
type Step = Up Int | Rgt Int
type alias Move = List Step
```

```
go up 3; go right 4; go up -1
```

```
[Up 3,Rgt 4,Up -1]
```

Semantic Function

```
sem : Move -> Pos
sem m = case m of
    []         -> (0,0)
    Up n::ms   -> addPos (0,n) (sem ms)
    Rgt n::ms  -> addPos (n,0) (sem ms)
```

Exercise …

Question 3

Define the semantics for the Move language as the total distance traveled, i.e., the semantic domain for moves is `Int`.

1. Abstract Syntax

```
type Dir = Up | Rgt
type alias Move = List (Dir,Int)
```

2. Semantic Domain

```
type alias Dist = Int
```

3. Semantic Function

```
sem : Move -> Dist
sem m = case m of
    []           -> 0
    (_,n)::ms   -> abs n + sem ms
```

Product Domains

Product Domain

A semantic domain for combining semantic features of types `V1` *and* `V2` *can be defined as the tuple type* `(V1,V2)`.

```
type alias V = (V1,V2)
```

Combining Semantic Functions

The semantic functions `sem1 : T -> V1` *and* `sem2 : T -> V2` *are combined as follows.*

```
sem : T -> V
sem p = (sem1 p,sem2 p)
```

Example: Position & Distance of Moves

1. Abstract Syntax

```
type Move = GoUp Int | ...
```

2. Combined Semantic Domains

```
type alias Pos = (Int,Int)
type alias Dist = Int

type alias PosAndDist = (Pos,Dist)
```

3. Semantic Function

```
semP : Move -> Pos
semP m = ...

semD : Move -> Dist
semD m = ...

sem : Move -> PosAndDist
sem m = (semP m,semD m)
```

Error Domains

Lifted Error Domain

To add errors to a type `V` *of "regular" values, use:*

(A) `Maybe V`, *or*

(B) *If* `V` *is a data type, add an* `Error` *or* `Undefined` *constructor to* `V`

(A) Using `Maybe`

```
type Maybe a = Just a | Nothing
type alias V_Error = Maybe V
```

(B) Adding Constructor

```
type V_Error = C1 ... | ... | Ck ... | Error
```

Example: Division by Zero

1. Abstract Syntax

```
type Expr = ...
          | Div Expr Expr
```

2. Semantic Domain

```
type Value = Maybe Int
```

3. Semantic Function

```
sem : Expr -> Maybe Int
...
sem (Plus e1 e2) = case sem e1 of
                     Just i -> case sem e2 of
                                 Just j -> Just (i+j)
                                 _      -> Nothing
                     _      -> Nothing
```

Grouping Case Expressions

1. Abstract Syntax

```
type Expr = ...
          | Div Expr Expr
```

2. Semantic Domain

```
type Value = Maybe Int
```

3. Semantic Function

```
sem : Expr -> Maybe Int
...
sem (Plus e1 e2) = case (sem e1,sem e2) of
                     (Just i,Just j) -> Just (i+j)
                     _               -> Nothing
```

Factoring Error Handling

Import from the Maybe module

```
map2 : (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
map2 f m1 m2 = case (m1,m2) of
    (Just x,Just y) -> Just (f x y)
    _               -> Nothing
```

Customized Control Structures

Possible with Higher-Order Functions!

3. Semantic Function

```
sem : Expr -> Maybe Int
sem e = case e of
    Num i        -> Just i
    Plus e1 e2   -> map2 (+) (sem e1) (sem e2)
    Times e1 e2  -> map2 (*) (sem e1) (sem e2)
    Div e1 e2    -> if sem e2==Just 0 then Nothing
                                      else map2 (//) (sem e1) (sem e2)
```

Union Domains

Union Domain

A semantic domain that comprises k alternative types `V1` *…* `Vk` *can be defined as a data type with k constructors.*

```
type V = C1 V1
       | ...
       | Ck Vk
```

Each `Cj` *injects (or "lifts") values of type* `Vj` *into* `V`

Example: Two-Type Expressions

1. Abstract Syntax

```
type Expr = Num Int | Plus Expr Expr | Equal Expr Expr | Not Expr
```

2. Semantic Domain

```
type Val = I Int
         | B Bool
         | Undefined
```

3. Semantic Function

```
sem : Expr -> Val
sem e = case e of
    Num i        -> I i
    ...
    Plus e1 e2   -> case (sem e1,sem e2) of
                      (I i,I j) -> I (i+j)
                      _         -> Undefined
```

Factoring Dynamic Type Checking

Custom Lifting Function

```
liftIII : (Int -> Int -> Int) -> Val -> Val -> Val
liftIII f v1 v2 = case (v1,v2) of
    (I x,I y) -> I (f x y)
    _         -> Undefined
```

Customized Control Structures

Possible with Higher-Order Functions!

3. Semantic Function

```
sem : Expr -> Val
sem e = case e of
    Num i        -> I i
    ...
    Plus e1 e2  -> liftIII (+) (sem e1) (sem e2)
    Times e1 e2 -> liftIII (*) (sem e1) (sem e2)
```

Domains for Stateful Computation

State Update Domain

If `V` is a type representing a state, then a domain for state updates on `V` can be represented as the function type `V -> V`.

```
type Update = V -> V
```

***Note:** This is a very general and widely applicable schema.*

States can be plain values (e.g., `Int`)
or arbitrary mappings (e.g., `[(Name,Value)]`).

Example: Machine Language

1. Abstract Syntax

```
type Op = LD Int | INC | DBL
type alias Prog = List Op
```

2. Semantic Domains

```
type alias State = Int
type alias Update = State -> State
```

3. Semantic Functions

```
-- exec : Op -> Update
exec : Op -> State -> State
exec op s = case op of
    LD i -> i
    INC  -> s+1
    DBL  -> s*2

sem : Prog -> State -> State
sem p s = case p of
    []      -> s
    op::ops -> sem ops (exec op s)
```

Function Domains

Multi-Parameter View

```
exec : Op -> State -> State
exec op s = case op of
    ...
    INC -> s+1

sem : Prog -> State -> State
sem p s = case p of
    ...
    op::ops -> sem ops (exec op s)
```

Function View

```
exec : Op -> (State -> State)
exec op = \s -> case op of
    ...
    INC -> s+1

sem : Prog -> (State -> State)
sem p = \s -> case p of
    ...
    op::ops -> sem ops (exec op s)
```

Exercise ...

Question 4

Extend the machine language syntax to work with 2 registers A *and* B.

```
type Op = LD Int | INC | DBL
type alias Prog = List Op
```

Hints: *(1) You need a new type* Reg *for registers.*

(2) Operations must be parameterized by Reg.

```
type Reg = A | B
type Op = LD Reg Int | INC Reg | DBL Reg
type alias Prog = List Op
```

Exercise ...

Question 5

Define the semantic domain(s) for the extended machine language.

```
type Reg = A | B
type Op = LD Reg Int | INC Reg | DBL Reg
type alias Prog = List Op
```

```
type alias State = (Int,Int)
type alias Update = State -> State
```

Exercise ...

Question 6

Define the semantic function for the extended machine language.

```
type Reg = A | B
type Op = LD Reg Int | INC Reg | DBL Reg
type alias Prog = List Op

type alias State = (Int,Int)
type alias Update = State -> State
```

```
exec : Op -> Update
exec op (a,b) = case op of
    LD A i -> (i,b)
    LD B i -> (a,i)
    INC A  -> (a+1,b)
    INC B  -> (a,b+1)
    ...
```

Redundant code for each register ...

Refactoring Semantics

```
type Reg = A | B
type Op = LD Reg Int
        | INC Reg | DBL Reg
type Prog = [Op]

type State = (Int,Int)
type Update = State -> State

exec : Op -> Update
exec op (a,b) = case op of
    LD A i -> (i,b)
    LD B i -> (a,i)
    INC A  -> (a+1,b)
    INC B  -> (a,b+1)
    ...
```

```
onReg : Reg -> (Int -> Int) -> Update
onReg r f (a,b) = case r of
    A -> (f a,b)
    B -> (a,f b)

exec : Op -> Update
exec op = case op of
   LD r i -> onReg r (\_->i)
   INC r  -> onReg r ((+) 1)
   DBL r  -> onReg r ((*) 2)
```

Note: onReg *is a control structure for dispatching semantic effects.*

Discussion ...

Consider the following abstract syntax.

```
type Exp = Num Int | Plus Exp Exp | Equal Exp Exp
```

Which type definition for `D` *should be used as the semantic domain?*

```
  (One) type alias D = Int
  (Two) type alias D = Maybe Int
(Three) type       D = I Int | B Bool
 (Four) type       D = I Int | B Bool | Error   ✓
 (Five) type alias D = Maybe (Int,Bool)
```

Discussion ...

Consider a language for computing with integers and fractions.

```
type Exp = Num Int | Frac Int Int | Plus Exp Exp
```

Which type definition for `D` *could be used as the semantic domain?*

```
(One)   type alias D = Maybe (Int,Int)           ✓
(Two)   type       D = I Int | F (Int,Int)
(Three) type       D = I (Maybe Int) | F (Int,Int)   (✓)
(Four)  type       D = I Int | F (Int,Maybe Int)
(Five)  type       D = I Int | F (Maybe (Int,Int))   ✓
```