Denotational Semantics
and
Domain Theory
Outline

Denotational Semantics

Basic Domain Theory
  Introduction and history
  Primitive and lifted domains
  Sum and product domains
  Function domains

Meaning of Recursive Definitions
  Compositionality and well-definedness
  Least fixed-point construction
  Internal structure of domains
A denotational semantics relates each **term** to a **denotation**

- an abstract syntax tree
- a value in some **semantic domain**

### Valuation function

\[
\lbrack \cdot \rbrack : \text{abstract syntax} \rightarrow \text{semantic domain}
\]

### Valuation function in Haskell

\[
\text{eval} :: \text{Term} \rightarrow \text{Value}
\]
**Semantic domain:** captures the set of possible meanings of a program/term

*what is a meaning? — it depends on the language!*

<table>
<thead>
<tr>
<th>Language</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean expressions</td>
<td>Boolean value</td>
</tr>
<tr>
<td>Arithmetic expressions</td>
<td>Integer</td>
</tr>
<tr>
<td>Imperative language</td>
<td>State transformation</td>
</tr>
<tr>
<td>SQL query</td>
<td>Set of relations</td>
</tr>
<tr>
<td>ActionScript</td>
<td>Animation</td>
</tr>
<tr>
<td>MIDI</td>
<td>Sound waves</td>
</tr>
</tbody>
</table>
Defining a language with denotational semantics

Example encoding in Haskell:

1. Define the **abstract syntax**, $T$
   the set of abstract syntax trees

2. Identify or define the **semantic domain**, $V$
   the representation of semantic values

3. Define the **valuation function**, $\sem : T \rightarrow V$
   the mapping from ASTs to semantic values
   a.k.a. the “semantic function”

```haskell
data Term = ...  

type Value = ... 

sem :: Term -> Value
```
Example: simple arithmetic expressions

1. Define abstract syntax

\[ n \in \text{Nat} \quad ::= \quad 0 \mid 1 \mid 2 \mid \ldots \]
\[ e \in \text{Exp} \quad ::= \quad \text{add} \quad e \quad e \]
\[ \quad \mid \quad \text{mul} \quad e \quad e \]
\[ \quad \mid \quad \text{neg} \quad e \]
\[ \quad \mid \quad n \]

2. Define semantic domain

Use the set of all integers, \( \text{Int} \)

Comes with some operations:
\[ +, \times, -, \text{toInt} : \text{Nat} \rightarrow \text{Int}, \ldots \]

3. Define the valuation function

\[ [\text{Exp}] : \text{Int} \]
\[ [\text{add} \quad e_1 \quad e_2] = [e_1] + [e_2] \]
\[ [\text{mul} \quad e_1 \quad e_2] = [e_1] \times [e_2] \]
\[ [\text{neg} \quad e] = -[e] \]
\[ [n] = \text{toInt}(n) \]

Denotational Semantics
1. **abstract syntax**: define a new **data type**, as usual
2. **semantic domain**: identify and/or define a new **type**, as needed
3. **valuation function**: define a **function** from ASTs to semantic domain

### Valuation function in Haskell

```haskell
sem :: Exp -> Int
sem (Add l r) = sem l + sem r
sem (Mul l r) = sem l * sem r
sem (Neg e) = negate (sem e)
sem (Lit n) = n
```
Desirable properties of a denotational semantics

**Compositionality**: a program’s denotation is built from the denotations of its parts
- supports modular reasoning, extensibility
- supports proof by structural induction

**Completeness**: every value in the semantic domain is denoted by some program
- ensures that semantic domain and language align
- if not, language has expressiveness gaps, or semantic domain is too general

**Soundness**: two programs are “equivalent” iff they have the same denotation
- equivalence: same w.r.t. to some other definition
- ensures that the denotational semantics is correct
More on compositionality

**Compositionality**: a program’s denotation is built from the denotations of its parts

- an AST
- sub-ASTs

Example: What is the meaning of $\text{op } e_1 e_2 e_3$?

1. Determine the meaning of $e_1, e_2, e_3$
2. Combine these submeanings in some way specific to $\text{op}$

Implications:

- The valuation function is probably **recursive**
- Often need different valuation functions for **each syntactic category**
Example: move language

A language describing movements on a 2D plane

- a **step** is an $n$-unit horizontal or vertical movement
- a **move** is described by a sequence of steps

Abstract syntax

\[

definitions:
\begin{align*}
n & \in \text{Nat} &::= & \emptyset | 1 | 2 | \ldots \\
d & \in \text{Dir} &::= & N | S | E | W \\
s & \in \text{Step} &::= & \text{go} \ d \ n \\
m & \in \text{Move} &::= & \epsilon | s ; m
\end{align*}
\]
Semantics of move language

1. Abstract syntax

\[
\begin{align*}
n & \in \text{Nat} \quad ::= \quad 0 \mid 1 \mid 2 \mid \ldots \\
d & \in \text{Dir} \quad ::= \quad N \mid S \mid E \mid W \\
s & \in \text{Step} \quad ::= \quad \text{go } d \ n \\
m & \in \text{Move} \quad ::= \quad \epsilon \mid s \ ; \ m
\end{align*}
\]

2. Semantic domain

\[
\text{Pos} = \text{Int} \times \text{Int}
\]

Domain: \( \text{Pos} \rightarrow \text{Pos} \)

3. Valuation function (Step)

\[
\begin{align*}
S[\text{Step}] & : \text{Pos} \rightarrow \text{Pos} \\
S[\text{go } N \ k] & = \lambda (x, y). (x, y + k) \\
S[\text{go } S \ k] & = \lambda (x, y). (x, y - k) \\
S[\text{go } E \ k] & = \lambda (x, y). (x + k, y) \\
S[\text{go } W \ k] & = \lambda (x, y). (x - k, y)
\end{align*}
\]

3. Valuation function (Move)

\[
\begin{align*}
M[\text{Move}] & : \text{Pos} \rightarrow \text{Pos} \\
M[\epsilon] & = \lambda p. p \\
M[s \ ; \ m] & = M[m] \circ S[s]
\end{align*}
\]
Alternative semantics

Often multiple interpretations (semantics) of the same language

Example: Database schema
One declarative spec, used to:
• initialize the database
• generate APIs
• validate queries
• normalize layout
• ...

Distance traveled

\[ S_D[\text{Step}] : \text{Int} \]
\[ S_D[\text{go } d \ k] = k \]
\[ M_D[\text{Move}] : \text{Int} \]
\[ M_D[\epsilon] = 0 \]
\[ M_D[s ; m] = S_D[s] + M_D[m] \]

Combined trip information

\[ M_C[\text{Move}] : \text{Int} \times (\text{Pos} \rightarrow \text{Pos}) \]
\[ M_C[m] = (M_D[m], M[m]) \]
Simple semantic domains can be combined in two ways:

- **product**: contains a value from both domains
  - e.g. combined trip information for move language
  - use Haskell \((a, b)\) or define a new data type

- **sum**: contains a value from one domain or the other
  - e.g. IntBool language can evaluate to \(\text{Int}\) or \(\text{Bool}\)
  - use Haskell \(\text{Either } a \ b\) or define a new data type

Can errors occur?
- use Haskell \(\text{Maybe } a\) or define a new data type

Does the language manipulate state or use naming?
- use a **function type**
Denotational Semantics

Basic Domain Theory
   Introduction and history
   Primitive and lifted domains
   Sum and product domains
   Function domains

Meaning of Recursive Definitions
   Compositionality and well-definedness
   Least fixed-point construction
   Internal structure of domains
What is domain theory?

**Domain theory**: a mathematical framework for constructing **semantic domains**

Recall …

A denotational semantics relates each **term** to a **denotation**

- an abstract syntax tree
- a value in some **semantic domain**

**Semantic domain**: captures the set of possible meanings of a program/term
Historical notes

Origins of domain theory:

- **Christopher Strachey**, 1964
  - early work on denotational semantics
  - used *lambda calculus* for denotations

- **Dana Scott**, 1975
  - goal: denotational semantics for lambda calculus itself
  - created domain theory for meaning of recursive functions

More on Dana Scott:

- Turing award in 1976 for nondeterminism in automata theory
- PhD advisor: **Alonzo Church**, 20 years after **Alan Turing**
Two views of denotational semantics

View #1: **Translation** from one formal system to another
- e.g. translate object language into lambda calculus

View #2: “**True meaning**” of a program as a mathematical object
- e.g. map programs to elements of a semantic domain
- need **domain theory** to describe set of meanings
Domains as semantic algebras

A **semantic domain** can be viewed as an **algebraic structure**

- a set of **values** the meanings of the programs
- a set of **operations** on the values used to compose meanings of parts

Domains also have internal structure: **complete partial ordering** (later)
Outline

Denotational Semantics

Basic Domain Theory
   Introduction and history
   Primitive and lifted domains
   Sum and product domains
   Function domains

Meaning of Recursive Definitions
   Compositionality and well-definedness
   Least fixed-point construction
   Internal structure of domains
Primitive domains

Values are **atomic**
- often correspond to **built-in types** in Haskell
- **nullary operations** for naming values explicitly

**Domain: Bool**
- `true : Bool`
- `false : Bool`
- `not : Bool → Bool`
- `and : Bool × Bool → Bool`
- `or : Bool × Bool → Bool`

**Domain: Int**
- `0, 1, 2, ... : Int`
- `negate : Int → Int`
- `plus : Int × Int → Int`
- `times : Int × Int → Int`

**Domain: Unit**
- `() : Unit`

*Also: Nat, Name, Addr, …*
Lifted domains

**Construction**: add $\bot$ (bottom) to an existing domain

$$A_\bot = A \cup \{\bot\}$$

**New operations**

$$\bot : A_\bot$$

$$map : (A \rightarrow B) \times A_\bot \rightarrow B_\bot$$

$$maybe : B \times (A \rightarrow B) \times A_\bot \rightarrow B$$
Encoding lifted domains in Haskell

**Option #1: Maybe**

```haskell
data Maybe a = Nothing
  | Just a

fmap :: (a -> b) -> Maybe a -> Maybe b
maybe :: b -> (a -> b) -> Maybe a -> b
```

Can also use pattern matching!

**Option #2: new data type with nullary constructor**

```haskell
data Value = Success Int | Error
```

Best when combined with other constructions
Outline

Denotational Semantics

Basic Domain Theory
  Introduction and history
  Primitive and lifted domains
  **Sum and product domains**
  Function domains

Meaning of Recursive Definitions
  Compositionality and well-definedness
  Least fixed-point construction
  Internal structure of domains
Sum domains

**Construction:** the **disjoint union** of two existing domains
- contains a value from either one domain or the other

\[ A \oplus B = A \uplus B \]

**New operations**

- \( inL : A \rightarrow A \oplus B \)
- \( inR : B \rightarrow A \oplus B \)
- \( case : (A \rightarrow C) \times (B \rightarrow C) \times (A \oplus B) \rightarrow C \)
Encoding sum domains in Haskell

Option #1: **Either**

```haskell
data Either a b = Left a  
  | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
```

Can also use pattern matching!

Option #2: new data type with multiple constructors

```haskell
data Value = I Int  | B Bool
```

Best when combined with other constructions, or more than two options
Example: a language with multiple types

\[ b \in \text{Bool} ::= \text{true} \mid \text{false} \]
\[ n \in \text{Nat} ::= 0 \mid 1 \mid 2 \mid \ldots \]
\[ e \in \text{Exp} ::= \text{add} \ e \ e \mid \text{neg} \ e \mid \text{equal} \ e \ e \mid \text{cond} \ e \ e \ e \mid n \mid b \]

Design a denotational semantics for \( Exp \)

1. How should we define our semantic domain?
2. Define a valuation semantics function

- **neg** – negates either a numeric or boolean value
- **equal** – compares two values of the same type for equality
- **cond** – equivalent to if–then–else
Solution

\[ [\text{Exp}] : (\text{Int} \oplus \text{Bool}) \perp \]

\[ [\text{add } e_1 \ e_2] = \begin{cases} [e_1] + [e_2] & [e_1] \in \text{Int}, [e_2] \in \text{Int} \\ \bot & \text{otherwise} \end{cases} \]

\[ [\text{neg } e] = \begin{cases} -[e] & [e] \in \text{Int} \\ \neg [e] & [e] \in \text{Bool} \\ \bot & \text{otherwise} \end{cases} \]

\[ [\text{equal } e_1 \ e_2] = \begin{cases} [e_1] =_{\text{Int}} [e_2] & [e_1] \in \text{Int}, [e_2] \in \text{Int} \\ [e_1] =_{\text{Bool}} [e_2] & [e_1] \in \text{Bool}, [e_2] \in \text{Bool} \\ \bot & \text{otherwise} \end{cases} \]

\[ [\text{cond } e_1 \ e_2 \ e_3] = \begin{cases} [e_2] & [e_1] = \text{true} \\ [e_3] & [e_1] = \text{false} \\ \bot & \text{otherwise} \end{cases} \]

\[ [n] = n \]
\[ [b] = b \]
**Construction:** the **cartesian product** of two existing domains

- contains a value from both domains

\[ A \otimes B = \{(a, b) \mid a \in A, b \in B\} \]

**New operations**

- `pair : A × B \rightarrow A \otimes B`
- `fst : A \otimes B \rightarrow A`
- `snd : A \otimes B \rightarrow B`
Encoding product domains in Haskell

Option #1: Tuples

```
type Value a b = (a,b)

fst :: (a,b) -> a
snd :: (a,b) -> b
```

Can also use pattern matching!

Option #2: new data type with multiple arguments

```
data Value = V Int Bool
```

Best when combined with other constructions, or more than two
Outline

Denotational Semantics

Basic Domain Theory
  Introduction and history
  Primitive and lifted domains
  Sum and product domains
  Function domains

Meaning of Recursive Definitions
  Compositionality and well-definedness
  Least fixed-point construction
  Internal structure of domains
**Function space domains**

**Construction:** the set of **functions** from one domain to another

\[ A \to B \]

Create a function: \( A \to B \)

Lambda notation: \( \lambda x. y \)
where \( \Gamma, x : A \vdash y : B \)

Eliminate a function

\( apply : (A \to B) \times A \to B \)
Denotational semantics of naming

**Environment**: a function associating names with things

\[ \text{Env} = \text{Name} \rightarrow \text{Thing} \]

**Naming concepts**
- **declaration**: add a new name to the environment
- **binding**: set the thing associated with a name
- **reference**: get the thing associated with a name

**Example semantic domains for expressions with ...**
- **immutable** variables (Haskell): \( \text{Env} \rightarrow \text{Val} \)
- **mutable** variables (C/Java/Python): \( \text{Env} \rightarrow \text{Env} \otimes \text{Val} \)
Example: Denotational semantics of \textbf{let} language

1. Abstract syntax

\begin{align*}
i \in \text{Int} & : := (\text{any integer}) \\
v \in \text{Var} & : := (\text{any variable name}) \\
e \in \text{Exp} & : := i \\
& \quad \mid \text{add } e e \\
& \quad \mid \text{let } v e e \\
& \quad \mid v
\end{align*}

2. Identify semantic domain

\begin{enumerate}
\item Result of evaluation: \( \text{Int}_\perp \)
\item Environment: \( \text{Env} = \text{Var} \rightarrow \text{Int}_\perp \)
\item Semantic domain: \( \text{Env} \rightarrow \text{Int}_\perp \)
\end{enumerate}

3. Define a valuation function

\[ [\text{Exp}] : (\text{Var} \rightarrow \text{Int}_\perp) \rightarrow \text{Int}_\perp \]

\[ [i] = \lambda m. i \]

\[ [\text{add } e_1 e_2] = \lambda m. [e_1](m) +_\perp [e_2](m) \]

\[ [\text{let } v e_1 e_2] = \lambda m. [e_2](\lambda w. \text{if } w = v \text{ then } [e_1](m) \text{ else } m(w)) \]

\[ [v] = \lambda m. m(v) \]

\[ i +_\perp j = \begin{cases} i + j & i \in \text{Int}, j \in \text{Int} \\ \perp & \text{otherwise} \end{cases} \]
What is mutable state?

**Mutable state**: stored information that a program can read and write.

Typical semantic domains with state domain $S$:

- $S \rightarrow S$  state mutation as **main effect**
- $S \rightarrow S \otimes \text{Val}$  state mutation as **side effect**

Often: lifted codomain if mutation can fail

**Examples**

- the memory cell in a calculator  $S = \text{Int}$
- the stack in a stack language  $S = \text{Stack}$
- the store in many programming languages  $S = \text{Name} \rightarrow \text{Val}$
Example: Single register calculator language

1. Abstract syntax

\[
\begin{align*}
    i & \in \text{Int} & ::= & \text{(any integer)} \\
    e & \in \text{Exp} & ::= & i \\
    & & | & e + e \\
    & & | & \text{save } e \\
    & & | & \text{load}
\end{align*}
\]

Examples:

- \text{save (3+2) + load} \\
  \rightarrow 10

- \text{save 1 +} \\
  (\text{save 10 + load) + load} \\
  \rightarrow 31

2. Identify semantic domain

i. State (side effect): \text{Int}

ii. Result: \text{Int}

iii. Semantic domain: \text{Int} \rightarrow \text{Int} \otimes \text{Int}
Example: Single register calculator language

1. Abstract syntax

\( i \in \text{Int} ::= \text{(any integer)} \)

\( e \in \text{Exp} ::= i \)
  | \hspace{1em} e + e
  | \hspace{1em} \text{save } e
  | \hspace{1em} \text{load} \)

Examples:

- \( \text{save } (3+2) + \text{load} \)
  \( \rightsquigarrow \) \( 10 \)

- \( \text{save } 1 + (\text{save } 10 + \text{load}) + \text{load} \)
  \( \rightsquigarrow \) \( 31 \)

3. Define valuation function

\[ [\text{Exp}] : \text{Int} \to \text{Int} \otimes \text{Int} \]

\[ [i] = \lambda s. (s, i) \]

\[ [e_1 + e_2] = \lambda s. \text{let } (s_1, i_1) = [e_1](s) \]

\( (s_2, i_2) = [e_2](s_1) \in (s_2, i_1 + i_2) \)

\[ [\text{save } e] = \lambda s. \text{let } (s', i) = [e](s) \text{ in } (i, i) \]

\[ [\text{load } e] = \lambda s. (s, s) \]
Outline

Denotational Semantics

Basic Domain Theory
   Introduction and history
   Primitive and lifted domains
   Sum and product domains
   Function domains

Meaning of Recursive Definitions
   Compositionality and well-definedness
   Least fixed-point construction
   Internal structure of domains
Compositionality and well-definedness

Recall: a **denotational semantics** must be **compositional**
- a term’s denotation is built from the denotations of its parts

**Example: integer expressions**

\[
\begin{align*}
i & \in \text{Int} \quad ::= \quad \text{(any integer)} \\
e & \in \text{Exp} \quad ::= \quad i \ | \ \text{add } e \ e \ | \ \text{mul } e \ e
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{Exp} \rrbracket : \text{Int} \\
\llbracket i \rrbracket = i \\
\llbracket \text{add } e_1 \ e_2 \rrbracket = \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket \\
\llbracket \text{mul } e_1 \ e_2 \rrbracket = \llbracket e_1 \rrbracket \times \llbracket e_2 \rrbracket
\end{align*}
\]

Compositionality ensures the semantics is **well-defined** by **structural induction**

Each AST has **exactly one** meaning
A non-compositional (and ill-defined) semantics

Anti-example: while statement

\[ t \in \text{Test} \quad ::= \quad \ldots \]
\[ c \in \text{Cmd} \quad ::= \quad \ldots \quad | \quad \text{while } t \ c \]

\[ T[\text{Test}] : S \rightarrow \text{Bool} \]
\[ C[\text{Cmd}] : S \rightarrow S \]
\[ C[\text{while } t \ c] = \lambda s. \text{if } T[t](s) \text{ then } C[\text{while } t \ c](C[c](s)) \text{ else } s \]

Meaning of \textbf{while } t \ c \text{ in state } s:

1. evaluate \( t \) in state \( s \)
2. if true:
   a. run \( c \) to get updated state \( s' \)
   b. re-evaluate \textbf{while} \text{ in state } s' \quad \text{(not compositional)}
3. otherwise return \( s \) unchanged

Translational view:
meaning is an \textbf{infinite} expression!

Mathematical view:
may have \textbf{infinitely many} meanings!
Extensional vs. operational definitions of a function

Mathematical function
Defined **extensionally**:  
- a relation between inputs and outputs

Computational function (e.g. Haskell)
Usually defined **operationally**:  
- compute output by sequence of reductions

Example (intensional definition)
\[
fac(n) = \begin{cases} 
1 & n = 0 \\ 
n \cdot fac(n - 1) & \text{otherwise} 
\end{cases}
\]

Extensional meaning
\[
\{\ldots, (2, 2), (3, 6), (4, 24), \ldots\}
\]

Operational meaning
\[
fac(3) \leadsto 3 \cdot fac(2) \\
\leadsto 3 \cdot 2 \cdot fac(1) \\
\leadsto 3 \cdot 2 \cdot 1 \cdot fac(0) \\
\leadsto 3 \cdot 2 \cdot 1 \cdot 1 \\
\leadsto 6
\]

Meaning of Recursive Definitions
Extensional meaning of recursive functions

\[
grow(n) = \begin{cases} 
1 & n = 0 \\
grow(n + 1) - 2 & \text{otherwise}
\end{cases}
\]

Best extension (use \(\bot\) if undefined):
- \((0, 1), (1, \bot), (2, \bot), (3, \bot), (4, \bot), \ldots\) 

Other valid extensions:
- \((0, 1), (1, 2), (2, 4), (3, 6), (4, 8) \ldots\)
- \((0, 1), (1, 5), (2, 7), (3, 9), (4, 11) \ldots\)
- \(\ldots\)

Goal: best extension = only extension
Connection back to denotational semantics

A function space domain is a set of mathematical functions

Anti-example: while statement

\[ t \in \text{Test} ::= \ldots \]
\[ c \in \text{Cmd} ::= \ldots \mid \text{while } t \ c \]

\[ T[\text{Test}] : S \rightarrow \text{Bool} \]
\[ C[\text{Cmd}] : S \rightarrow S \]
\[ C[\text{while } t \ c] = \lambda s. \text{if } T[t](s) \text{ then } C[c](s) \text{ else } s \]

Ideal semantics of \( \text{Cmd} \):
- semantic domain: \( S \rightarrow S_\bot \)
- contains \( (s, s') \) if \( c \) terminates
- contains \( (s, \bot) \) if \( c \) diverges
Outline

Denotational Semantics

Basic Domain Theory
  Introduction and history
  Primitive and lifted domains
  Sum and product domains
  Function domains

Meaning of Recursive Definitions
  Compositionality and well-definedness
  Least fixed-point construction
  Internal structure of domains
Least fixed points

Basic idea:
1. a **recursive** function defines a **set** of **non-recursive, finite** subfunctions
2. its meaning is the “**union**” of the meanings of its subfunctions

Iteratively grow the extension until we reach a **fixed point**
- essentially encodes computational functions as mathematical functions
Recursive definition

\[ \text{fac}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fac}(n - 1) & \text{otherwise}
\end{cases} \]

Non-recursive, finite subfunctions

\[ \text{fac}_0(n) = \bot \]
\[ \text{fac}_1(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fac}_0(n - 1) & \text{otherwise}
\end{cases} \]
\[ \text{fac}_2(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fac}_1(n - 1) & \text{otherwise}
\end{cases} \]
\[ \vdots \]
\[ \text{fac}_i(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fac}_{i-1}(n - 1) & \text{otherwise}
\end{cases} \]

\[ \text{fac} = \bigcup_{i=0}^{\infty} \text{fac}_i \]

Fine print:

- each \( \text{fac}_i \) maps all other values to \( \bot \)
- \( \bigcup \) operation prefers non-\( \bot \) mappings
Computing the fixed point

In general

\[ \text{fac}_0(n) = \bot \]
\[ \text{fac}_i(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fac}_{i-1}(n-1) & \text{otherwise} \end{cases} \]

A template to represent all \( \text{fac}_i \) functions:

\[ F = \lambda f. \lambda n. \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n - 1) & \text{otherwise} \end{cases} \]

Fixpoint operator

\[ \text{fix} : (A \rightarrow A) \rightarrow A \]
\[ \text{fix}(g) = \text{let } x = g(x) \text{ in } x \]

\[ \text{fix}(h) = h(h(h(h(h(\ldots))))) \]

Factorial as a fixed point

\[ \text{fac} = \text{fix}(F) \]
Outline

Denotational Semantics

Basic Domain Theory
  - Introduction and history
  - Primitive and lifted domains
  - Sum and product domains
  - Function domains

Meaning of Recursive Definitions
  - Compositionality and well-definedness
  - Least fixed-point construction
  - Internal structure of domains
Why domains are not flat sets

**Internal structure** of domains supports the least fixed-point construction

Recall fine print from factorial example:

- each $\text{fac}_i$ maps all other values to $\bot$
- $\cup$ operation prefers non-$\bot$ mappings

How can we **generalize** and **formalize** this idea?
Partial orderings and joins

<table>
<thead>
<tr>
<th>Partial ordering: $\sqsubseteq : D \times D \rightarrow \mathbb{B}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>• reflexive: $\forall x \in D. \quad x \sqsubseteq x$</td>
</tr>
<tr>
<td>• antisymmetric: $\forall x, y \in D. \quad x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$</td>
</tr>
<tr>
<td>• transitive: $\forall x, y, z \in D. \quad x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Join: $\sqcup : D \times D \rightarrow D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall a, b \in D$, the element $c = a \sqcup b \in D$, if it exists,</td>
</tr>
<tr>
<td>is the smallest element that is larger than both $a$ and $b$</td>
</tr>
</tbody>
</table>

i.e. $a \sqsubseteq c$ and $b \sqsubseteq c$, and there is no $d = a \sqcup b \in D$ where $d \sqsubseteq c$
(Scott) domains are directed-complete partial orderings

A domain is a directed-complete partial ordered (dcpo) set:

• every directed subset (related by \(\sqsubseteq\)) of a domain has \(\bot\)

The meaning of a (Scott-continuous) recursive function \(f\) is:

\[
\bigwedge_{i=0}^{\infty} f_i
\]

where \(f_i\) are the finite approximations of \(f\)
### Syntax

\[
\begin{align*}
    t \in Test & ::= \ldots \\
    c \in Cmd & ::= \ldots \mid \text{while } t \ c
\end{align*}
\]

### Semantics

\[
\begin{align*}
    T[\Test] & : S \rightarrow \text{Bool} \\
    C[\Cmd] & : S \rightarrow S \\
    C[\text{while } t \ c] & = \text{fix} (\lambda f. \lambda s. \text{if } T[t](s) \text{ then } f(C[c](s)) \text{ else } s)
\end{align*}
\]