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

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

### Valuation function in Haskell

```
eval :: Term -> 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

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”

Example encoding in Haskell:

```haskell
data Term = ...

type Value = ...

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

1. Define abstract syntax

\[
\begin{align*}
n \in \text{Nat} & \quad ::= \quad 0 \mid 1 \mid 2 \mid \ldots \\
e \in \text{Exp} & \quad ::= \quad \text{add} \; e \; e \\
& \quad \mid \quad \text{mul} \; e \; e \\
& \quad \mid \quad \text{neg} \; e \\
& \quad \mid \quad n
\end{align*}
\]

2. Define semantic domain

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

Comes with some operations:

\(+, \times, -, \text{toInt} : \text{Nat} \to \text{Int}, \ldots\)

3. Define the valuation function

\[
\begin{align*}
[\text{Exp}] & \colon \text{Int} \\
[\text{add} \; e_1 \; e_2] & = [e_1] + [e_2] \\
[mul \; e_1 \; e_2] & = [e_1] \times [e_2] \\
[neg \; e] & = -[e] \\
[n] & = \text{toInt}(n)
\end{align*}
\]
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

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

1. Abstract syntax

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

2. Semantic domain

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

\[
\text{Domain: } \text{Pos} \rightarrow \text{Pos}
\]

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

\[
M_C[\text{Move}] : \text{Int} \times (\text{Pos} \to \text{Pos}) \\
M_C[m] = (M_D[m], M[m])
\]
Picking the right semantic domain

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 Int or Bool
  - use Haskell \(\text{Either } a \text{ } b\) 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**
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
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
Primal domains

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

**Domain: \( \text{Bool} \)**
- \( \text{true} : \text{Bool} \)
- \( \text{false} : \text{Bool} \)
- \( \text{not} : \text{Bool} \to \text{Bool} \)
- \( \text{and} : \text{Bool} \times \text{Bool} \to \text{Bool} \)
- \( \text{or} : \text{Bool} \times \text{Bool} \to \text{Bool} \)

**Domain: \( \text{Int} \)**
- \( 0, 1, 2, \ldots : \text{Int} \)
- \( \text{negate} : \text{Int} \to \text{Int} \)
- \( \text{plus} : \text{Int} \times \text{Int} \to \text{Int} \)
- \( \text{times} : \text{Int} \times \text{Int} \to \text{Int} \)

**Domain: \( \text{Unit} \)**
- \( () : \text{Unit} \)

Also: \( \text{Nat} \), \( \text{Name} \), \( \text{Addr} \), \( \ldots \)
Lifted domains

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

\[
A_\bot = A \cup \{ \bot \}
\]

**New operations**

\[
\bot : A_\bot \\
map : (A \rightarrow B) \times A_\bot \rightarrow B_\bot \\
maybe : B \times (A \rightarrow B) \times A_\bot \rightarrow B_\bot
\]
Option #1: **Maybe**

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

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

\[
\begin{align*}
\text{inL} : A &\rightarrow A \oplus B \\
\text{inR} : B &\rightarrow A \oplus B \\
\text{case} : (A \rightarrow C) \times (B \rightarrow D) \times (A \oplus B) &\rightarrow C \oplus D
\end{align*}
\]
Encoding sum domains in Haskell

Option #1: Either

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

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

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

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

Design a denotational semantics for \(\text{Exp}\)

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 \textit{if–then–else}
Solution

$$[[\text{Exp}]] : (\text{Int} \oplus \text{Bool})_\bot$$

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

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

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

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

$$[[n]] = n$$
$$[[b]] = b$$
**Product domains**

**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: \( A \times B \rightarrow A \otimes B \)
- \( \text{fst}: A \otimes B \rightarrow A \)
- \( \text{snd}: A \otimes B \rightarrow B \)
Encoding product domains in Haskell

**Option #1: Tuples**

```haskell
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**

```haskell
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

$$A \rightarrow B$$

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

Lambda notation: \( \lambda x. y \)

where \( \Gamma, x : A \vdash y : B \)

Eliminate a function

\( \text{apply} : (A \rightarrow B) \times A \rightarrow B \)
Denotational semantics of naming

Environment: a function associating names with things

$$Env = Name \rightarrow Thing$$

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): $$Env \rightarrow Val$$
- **mutable** variables (C/Java/Python): $$Env \rightarrow Env \otimes Val$$
Example: Denotational semantics of \texttt{let} language

1. Abstract syntax

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

2. Identify semantic domain

i. Result of evaluation: \( Int_\perp \)

ii. Environment: \( Env = Var \rightarrow Int_\perp \)

iii. Semantic domain: \( Env \rightarrow Int_\perp \)

3. Define a valuation function

\[
\begin{align*}
\llbracket Exp \rrbracket : (Var \rightarrow Int_\perp) \rightarrow Int_\perp \\
\llbracket i \rrbracket & = \lambda m. i \\
\llbracket \text{add } e_1 \ e_2 \rrbracket & = \lambda m. \llbracket e_1 \rrbracket (m) +_\perp \llbracket e_2 \rrbracket (m) \\
\llbracket \text{let } v \ e_1 \ e_2 \rrbracket & = \lambda m. \llbracket e_2 \rrbracket (\lambda w. \text{if } w = v \ \text{then } \llbracket e_1 \rrbracket (m) \ \text{else } m(w)) \\
\llbracket v \rrbracket & = \lambda m. m(v)
\end{align*}
\]

\[
i +_\perp j = \begin{cases} 
  i + j & \text{if } i \in Int, \ j \in Int \\
  \perp & \text{otherwise}
\end{cases}
\]
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**

- $S \rightarrow S \otimes Val$  
  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  
  $S = \text{Name} \rightarrow \text{Val}$
1. Abstract syntax

\[
i \in \text{Int} \quad ::= \quad \text{(any integer)}
\]
\[
e \in \text{Exp} \quad ::= \quad i
| \quad e + e
| \quad \text{save } e
| \quad \text{load}
\]

Examples:

- save \((3+2) + \text{load}\)  
  \(\leadsto 10\)
- save \(1 + (\text{save } 10 + \text{load}) + \text{load}\)  
  \(\leadsto 31\)

2. Identify semantic domain

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

ii. Result:  \(\text{Int}\)

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

1. Abstract syntax

\[ i \in \text{Int} ::= \text{(any integer)} \]
\[ e \in \text{Exp} ::= i \mid e + e \mid \text{save } e \mid \text{load} \]

Examples:

- \text{save } (3+2) + \text{load} \quad \leadsto \quad 10
- \text{save } 1 + (\text{save } 10 + \text{load}) + \text{load} \quad \leadsto \quad 31

3. Define valuation function

\[ \lbrack \text{Exp} \rbrack : \text{Int} \rightarrow \text{Int} \otimes \text{Int} \]

\[ \lbrack i \rbrack = \lambda s. (s, i) \]
\[ \lbrack e_1 + e_2 \rbrack = \lambda s. \text{let } (s_1, i_1) = \lbrack e_1 \rbrack (s) \]
\[ (s_2, i_2) = \lbrack e_2 \rbrack (s_1) \]
\[ \text{in } (s_2, i_1 + i_2) \]
\[ \lbrack \text{save } e \rbrack = \lambda s. \text{let } (s', i) = \lbrack e \rbrack (s) \text{ in } (i, i) \]
\[ \lbrack \text{load } e \rbrack = \lambda s. (s, s) \]
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
Recall: a \textbf{denotational semantics} must be \textbf{compositional}

- a term’s denotation is built from the denotations of its parts

\textbf{Example: integer expressions}

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

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

Compositionality ensures the semantics is \textbf{well-defined} by \textbf{structural induction}

Each AST has \textbf{exactly one} meaning.
A non-compositional (and ill-defined) semantics

Anti-example: while statement

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

\[ T[\text{Test}] : S \rightarrow \text{Bool} \]
\[ C[\text{Cmd}] : S \rightarrow S \]

\[ C[\text{while } t \ c] = \lambda s. \ \text{if } T[t](s) \ \text{then} \]
\[ \quad C[\text{while } t \ c](C[c](s)) \]
\[ \text{else } s \]

Meaning of \textbf{while } t \ c \ 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} in state \( s' \)
      (not compositional)
3. otherwise return \( s \) unchanged

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

Mathematical view:
may have \textit{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)

\[
\begin{align*}
\text{fac}(n) &= \left\{ 
\begin{array}{ll}
1 & \quad n = 0 \\
n \cdot \text{fac}(n - 1) & \quad \text{otherwise}
\end{array}
\right.
\end{align*}
\]

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

**Operational meaning**

\[
\begin{aligned}
\text{fac}(3) &\mapsto 3 \cdot \text{fac}(2) \\
&\mapsto 3 \cdot 2 \cdot \text{fac}(1) \\
&\mapsto 3 \cdot 2 \cdot 1 \cdot \text{fac}(0) \\
&\mapsto 3 \cdot 2 \cdot 1 \cdot 1 \\
&\mapsto 6
\end{aligned}
\]
Extensional meaning of recursive functions

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

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

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
A **function space domain** is a set of **mathematical functions**

### Anti-example: while statement

\[
\begin{align*}
  t & \in \text{Test} \quad ::= \quad \ldots \\
  c & \in \text{Cmd} \quad ::= \quad \ldots \mid \textbf{while} \ t \ c \\
  T[\text{Test}] & : S \to \text{Bool} \\
  C[\text{Cmd}] & : S \to S \\
  C[\textbf{while} \ t \ c] & = \lambda s. \ \text{if} \ T[t](s) \ \text{then} \\
  & \quad \quad \quad C[\textbf{while} \ t \ c](C[c](s)) \\
  & \quad \quad \quad \text{else} \ s
\end{align*}
\]

### Ideal semantics of \(\text{Cmd}\):
- **semantic domain**: \(S \to S_\bot\)
- **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.
Example: unfolding a recursive definition

### Recursive definition

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

### Non-recursive, finite subfunctions

\[
\begin{align*}
\text{\textit{fac}}_0(n) &= \bot \\
\text{\textit{fac}}_1(n) &= \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{\textit{fac}}_0(n - 1) & \text{otherwise}
\end{cases} \\
\text{\textit{fac}}_2(n) &= \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{\textit{fac}}_1(n - 1) & \text{otherwise}
\end{cases} \\
\text{\textit{fac}}_3(n) &= \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{\textit{fac}}_2(n - 1) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\text{\textit{fac}}_i(n) = \bigcup_{i=0}^{\infty} \text{\textit{fac}}_i
\]

Fine print:
- each \textit{fac}_i maps all other values to \bot
- \bigcup operation prefers non-\bot mappings
Computing the fixed point

In general

\[ \text{fac}_0(n) = \perp \]

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

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

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

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

Partial ordering: \( \sqsubseteq : D \times D \rightarrow \mathbb{B} \)

- reflexive: \( \forall x \in D. \ x \sqsubseteq x \)
- antisymmetric: \( \forall x, y \in D. \ x \sqsubseteq y \land y \sqsubseteq x \implies x = y \)
- transitive: \( \forall x, y, z \in D. \ x \sqsubseteq y \land y \sqsubseteq z \implies x \sqsubseteq z \)

Join: \( \sqcup : D \times D \rightarrow D \)

\( \forall a, b \in D, \) the element \( c = a \sqcup b \in D, \) if it exists, is the smallest element that is larger than both \( a \) and \( b \)

i.e. \( a \sqsubseteq c \) and \( b \sqsubseteq c, \) and there is no \( d = a \sqcup b \in D \) where \( d \sqsubseteq c \)
A **domain** is a **directed-complete partial ordered** (dcpo) set:

- Every directed subset (related by \(\sqsubseteq\)) of a domain has \(\perp\)

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

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

where \(f_i\) are the finite approximations of \(f\)
Well-defined semantics for the while statement

Syntax

\[
\begin{align*}
t & \in Test \quad ::= \quad ... \\
c & \in Cmd \quad ::= \quad ... \quad | \quad \text{while } t \ c
\end{align*}
\]

Semantics

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