Denotational Semantics

and

Domain Theory

Outline

Denotational semantics

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

an abstract syntax tree

a value in some **semantic domain**

Valuation function

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

Valuation function in Haskell

```
eval :: Term -> Value
```

Semantic domains

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

what is a meaning? – it depends on the language!

Example semantic domains

Language	Meaning
Boolean expressions	Boolean value
Arithmetic expressions	Integer
Imperative language	State transformation
SQL query	Set of relations
ActionScript	Animation
MIDI	Sound waves

Defining a language with denotational semantics

Example encoding in Haskell:

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

   ```
   data Term = ...
   ```

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

   ```
   type Value = ...
   ```

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

   ```
   sem :: Term -> Value
   ```

Example: simple arithmetic expressions

1. Define abstract syntax

$$
\begin{array}{rcl}
n \in \mathit{Nat} & ::= & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \ldots \\
e \in \mathit{Exp} & ::= & \mathtt{add}\ e\ e \\
& \mid & \mathtt{mul}\ e\ e \\
& \mid & \mathtt{neg}\ e \\
& \mid & n
\end{array}
$$

2. Define semantic domain

Use the set of all integers, *Int*

Comes with some operations:
$+, \times, -, \mathit{toInt} : \mathit{Nat} \rightarrow \mathit{Int}, \ldots$

3. Define the valuation function

$$
\begin{array}{rcl}
[\![\mathit{Exp}]\!] & : & \mathit{Int} \\
[\![\mathtt{add}\ e_1\ e_2]\!] & = & [\![e_1]\!] + [\![e_2]\!] \\
[\![\mathtt{mul}\ e_1\ e_2]\!] & = & [\![e_1]\!] \times [\![e_2]\!] \\
[\![\mathtt{neg}\ e]\!] & = & -[\![e]\!] \\
[\![n]\!] & = & \mathit{toInt}(n)
\end{array}
$$

Encoding denotational semantics in Haskell

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

```
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 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 **op**

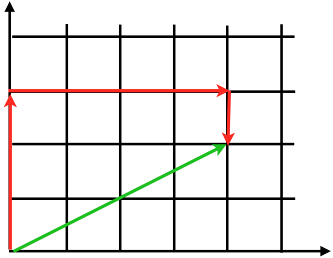
Implications:

- 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



```
go N 3; go E 4; go S 1;
```

Abstract syntax

$$
\begin{array}{rcl}
n \in \mathit{Nat} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
d \in \mathit{Dir} & ::= & \mathtt{N} \mid \mathtt{S} \mid \mathtt{E} \mid \mathtt{W} \\
s \in \mathit{Step} & ::= & \mathtt{go}\ \ d\ \ n \\
m \in \mathit{Move} & ::= & \epsilon \mid s\ \mathtt{;}\ m
\end{array}
$$

Semantics of move language

1. Abstract syntax

$$\begin{aligned} n \in Nat &::= \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\ d \in Dir &::= \mathtt{N} \mid \mathtt{S} \mid \mathtt{E} \mid \mathtt{W} \\ s \in Step &::= \mathtt{go}\ \ d\ \ n \\ m \in Move &::= \epsilon \mid s\ \mathtt{;}\ m \end{aligned}$$

2. Semantic domain

$Pos = Int \times Int$

Domain: $Pos \to Pos$

3. Valuation function (*Step*)

$$\begin{aligned} S[\![\,Step\,]\!] &: Pos \to Pos \\ S[\![\mathtt{go\ N}\ k]\!] &= \lambda(x,y).\ (x, y+k) \\ S[\![\mathtt{go\ S}\ k]\!] &= \lambda(x,y).\ (x, y-k) \\ S[\![\mathtt{go\ E}\ k]\!] &= \lambda(x,y).\ (x+k, y) \\ S[\![\mathtt{go\ W}\ k]\!] &= \lambda(x,y).\ (x-k, y) \end{aligned}$$

3. Valuation function (*Move*)

$$\begin{aligned} M[\![\,Move\,]\!] &: Pos \to Pos \\ M[\![\epsilon]\!] &= \lambda p.\ p \\ M[\![s\ \mathtt{;}\ m]\!] &= M[\![m]\!] \circ S[\![s]\!] \end{aligned}$$

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

$$
\begin{aligned}
S_D[\![\,Step\,]\!] &: Int \\
S_D[\![\textbf{go}\ d\ k]\!] &= k
\end{aligned}
$$

$$
\begin{aligned}
M_D[\![\,Move\,]\!] &: Int \\
M_D[\![\epsilon]\!] &= 0 \\
M_D[\![s\ \textbf{;}\ m]\!] &= S_D[\![s]\!] + M_D[\![m]\!]
\end{aligned}
$$

Combined trip information

$$
\begin{aligned}
M_C[\![\,Move\,]\!] &: Int \times (Pos \to Pos) \\
M_C[\![m]\!] &= (M_D[\![m]\!], M[\![m]\!])
\end{aligned}
$$

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 `Either a b` or define a new data type

Can errors occur?

- use Haskell `Maybe a` or define a new data type

Does the language manipulate state or use naming?

- use a **function type**

Outline

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

Dana Scott

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

Primitive domains

Values are **atomic**

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

Domain: $Bool$

$$\begin{aligned} true &: Bool \\ false &: Bool \\ not &: Bool \to Bool \\ and &: Bool \times Bool \to Bool \\ or &: Bool \times Bool \to Bool \end{aligned}$$

Domain: Int

$$\begin{aligned} 0, 1, 2, \ldots &: Int \\ negate &: Int \to Int \\ plus &: Int \times Int \to Int \\ times &: Int \times Int \to Int \end{aligned}$$

Domain: $Unit$

$$() : Unit$$

Also: Nat, $Name$, $Addr$, $\ldots$

Lifted domains

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

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

New operations

$$\begin{aligned} \bot &: A_\bot \\ map &: (A \to B) \times A_\bot \to B_\bot \\ maybe &: B \times (A \to B) \times A_\bot \to B \end{aligned}$$

Encoding lifted domains in Haskell

Option #1: `Maybe`

```
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

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

Best when combined with other constructions

Outline

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

$$inL : A \to A \oplus B$$
$$inR : B \to A \oplus B$$
$$case : (A \to C) \times (B \to C) \times (A \oplus B) \to C$$

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

$$
\begin{array}{rcl}
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \\
n \in \mathit{Nat} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
e \in \mathit{Exp} & ::= & \texttt{add}\ e\ e \\
& \mid & \texttt{neg}\ e \\
& \mid & \texttt{equal}\ e\ e \\
& \mid & \texttt{cond}\ e\ e\ e \\
& \mid & n \\
& \mid & b
\end{array}
$$

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

$$[\![\, Exp \,]\!] \;:\; (Int \oplus Bool)_\perp$$

$$[\![\texttt{add} \;\; e_1 \;\; e_2]\!] = \begin{cases} [\![e_1]\!] + [\![e_2]\!] & [\![e_1]\!] \in Int, [\![e_2]\!] \in Int \\ \perp & \textit{otherwise} \end{cases}$$

$$[\![\texttt{neg} \;\; e]\!] = \begin{cases} -[\![e]\!] & [\![e]\!] \in Int \\ \neg [\![e]\!] & [\![e]\!] \in Bool \\ \perp & \textit{otherwise} \end{cases}$$

$$[\![\texttt{equal} \;\; e_1 \;\; e_2]\!] = \begin{cases} [\![e_1]\!] =_{Int} [\![e_2]\!] & [\![e_1]\!] \in Int, [\![e_2]\!] \in Int \\ [\![e_1]\!] =_{Bool} [\![e_2]\!] & [\![e_1]\!] \in Bool, [\![e_2]\!] \in Bool \\ \perp & \textit{otherwise} \end{cases}$$

$$[\![\texttt{cond} \;\; e_1 \;\; e_2 \;\; e_3]\!] = \begin{cases} [\![e_2]\!] & [\![e_1]\!] = \texttt{true} \\ [\![e_3]\!] & [\![e_1]\!] = \texttt{false} \\ \perp & \textit{otherwise} \end{cases}$$

$$[\![n]\!] = n$$

$$[\![b]\!] = b$$

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

$$
\begin{aligned}
pair &: A \times B \to A \otimes B \\
fst &: A \otimes B \to A \\
snd &: A \otimes B \to B
\end{aligned}
$$

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

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

$$Env = Name \to Thing$$

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)	$Env \to Val$
mutable variables (C/Java/Python)	$Env \to Env \otimes Val$

Example: Denotational semantics of **let** language

1. Abstract syntax

$$
\begin{array}{rcl}
i \in Int & ::= & \text{(any integer)} \\
v \in Var & ::= & \text{(any variable name)} \\
e \in Exp & ::= & i \\
& \mid & \mathbf{add}\ e\ e \\
& \mid & \mathbf{let}\ v\ e\ e \\
& \mid & v
\end{array}
$$

2. Identify semantic domain

i. Result of evaluation: $Int_\perp$

ii. Environment: $Env = Var \to Int_\perp$

iii. Semantic domain: $Env \to Int_\perp$

3. Define a valuation function

$$[\![\, Exp \,]\!] : (Var \to Int_\perp) \to Int_\perp$$

$$
\begin{aligned}
[\![i]\!] &= \lambda m.\, i \\
[\![\mathbf{add}\ e_1\ e_2]\!] &= \lambda m.\, [\![e_1]\!](m) +_\perp [\![e_2]\!](m) \\
[\![\mathbf{let}\ v\ \ e_1\ e_2]\!] &= \lambda m.\, [\![e_2]\!](\lambda w.\, \mathbf{if}\ w = v \\
&\qquad\qquad \mathbf{then}\ [\![e_1]\!](m) \\
&\qquad\qquad \mathbf{else}\ m(w)) \\
[\![v]\!] &= \lambda m.\, m(v)
\end{aligned}
$$

$$
i +_\perp j = \begin{cases} i + j & i \in Int,\ j \in Int \\ \perp & \textit{otherwise} \end{cases}
$$

What is mutable state?

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

Typical semantic domains with state domain S:

$S \to S$	state mutation as **main effect**
$S \to S \otimes \mathit{Val}$	state mutation as **side effect**

Often: lifted codomain if mutation can fail

Examples

- the memory cell in a calculator $S = \mathit{Int}$
- the stack in a stack language $S = \mathit{Stack}$
- the store in many programming languages $S = \mathit{Name} \to \mathit{Val}$

Example: Single register calculator language

1. Abstract syntax

$$
\begin{array}{rcl}
i \in \mathit{Int} & ::= & \text{(any integer)} \\
e \in \mathit{Exp} & ::= & i \\
& | & e \;\mathtt{+}\; e \\
& | & \mathtt{save}\; e \\
& | & \mathtt{load}
\end{array}
$$

2. Identify semantic domain

i. State (side effect): Int

ii. Result: Int

iii. Semantic domain: $\mathit{Int} \to \mathit{Int} \otimes \mathit{Int}$

Examples:

- **save (3+2) + load**
  $\rightsquigarrow$ **10**
- **save 1 +**
  (save 10 + load) + load
  $\rightsquigarrow$ **31**

Example: Single register calculator language

1. Abstract syntax

$$
\begin{array}{rcl}
i \in \mathit{Int} & ::= & \text{(any integer)} \\
e \in \mathit{Exp} & ::= & i \\
& | & e \texttt{ + } e \\
& | & \texttt{save } e \\
& | & \texttt{load}
\end{array}
$$

Examples:

- `save (3+2) + load`
  $\rightsquigarrow$ `10`
- `save 1 +`
  `(save 10 + load) + load`
  $\rightsquigarrow$ `31`

3. Define valuation function

$$[\![\, \mathit{Exp} \,]\!] : \mathit{Int} \to \mathit{Int} \otimes \mathit{Int}$$

$$
\begin{array}{rcl}
[\![i]\!] & = & \lambda s.\, (s, i) \\
[\![e_1 \texttt{ + } e_2]\!] & = & \lambda s.\, \textbf{let}\ (s_1, i_1) = [\![e_1]\!](s) \\
& & \qquad\qquad (s_2, i_2) = [\![e_2]\!](s_1) \\
& & \qquad\quad \textbf{in}\ (s_2, i_1 + i_2) \\
[\![\texttt{save } e]\!] & = & \lambda s.\, \textbf{let}\ (s', i) = [\![e]\!](s)\ \textbf{in}\ (i, i) \\
[\![\texttt{load } e]\!] & = & \lambda s.\, (s, s)
\end{array}
$$

Outline

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

$$
\begin{aligned}
i \in \mathit{Int} \quad &::= \quad \text{(any integer)} \\
e \in \mathit{Exp} \quad &::= \quad i \mid \textbf{add}\ e\ e \mid \textbf{mul}\ e\ e
\end{aligned}
$$

$$
\begin{aligned}
[\![\, \mathit{Exp} \,]\!] &: \mathit{Int} \\
[\![i]\!] &= i \\
[\![\textbf{add}\ e_1\ e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
[\![\textbf{mul}\ e_1\ e_2]\!] &= [\![e_1]\!] \times [\![e_2]\!]
\end{aligned}
$$

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

$$
\begin{array}{rcll}
t \in \mathit{Test} & ::= & \ldots & \\
c \in \mathit{Cmd} & ::= & \ldots & \mid \; \mathtt{while}\ t\ c
\end{array}
$$

$$T[\![\,\mathit{Test}\,]\!] \;:\; S \to \mathit{Bool}$$

$$C[\![\,\mathit{Cmd}\,]\!] \;:\; S \to S$$

$$
\begin{aligned}
C[\![\mathtt{while}\ t\ c]\!] = \lambda s.\ &\textbf{if}\ T[\![t]\!](s)\ \textbf{then} \\
&C[\![\mathtt{while}\ t\ c]\!](C[\![c]\!](s)) \\
&\textbf{else}\ s
\end{aligned}
$$

Meaning of **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 **while** in state s'
   (not compositional)
3. otherwise return s unchanged

Translational view:
meaning is an **infinite** expression!

Mathematical view:
may have **infinitely many** meanings!

Extensional vs. operational definitions of a function

Mathematical function

Defined **extensionally**:

- a relation between inputs and outputs

Extensional meaning

$\{\ldots, (2,2), (3,6), (4,24), \ldots\}$

Computational function (e.g. Haskell)

Usually defined **operationally**:

- compute output by sequence of reductions

Operational meaning

$$\begin{aligned} fac(3) &\rightsquigarrow 3 \cdot fac(2) \\ &\rightsquigarrow 3 \cdot 2 \cdot fac(1) \\ &\rightsquigarrow 3 \cdot 2 \cdot 1 \cdot fac(0) \\ &\rightsquigarrow 3 \cdot 2 \cdot 1 \cdot 1 \\ &\rightsquigarrow 6 \end{aligned}$$

Example (intensional definition)

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

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:

- $\{(0,1), (1,2), (2,4), (3,6), (4,8) \ldots\}$
- $\{(0,1), (1,5), (2,7), (3,9), (4,11) \ldots\}$
- ...

Goal: best extension = **only** extension

Connection back to denotational semantics

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

Anti-example: while statement

$$
\begin{array}{rcl}
t \in \mathit{Test} & ::= & \ldots \\
c \in \mathit{Cmd} & ::= & \ldots \mid \texttt{while}\ t\ c
\end{array}
$$

$$\mathcal{T}[\![\, \mathit{Test} \,]\!] : S \rightarrow \mathit{Bool}$$

$$\mathcal{C}[\![\, \mathit{Cmd} \,]\!] : S \rightarrow S$$

$$
\begin{aligned}
\mathcal{C}[\![\texttt{while}\ t\ c]\!] = \lambda s.\ & \textbf{if}\ \mathcal{T}[\![t]\!](s)\ \textbf{then} \\
& \mathcal{C}[\![\texttt{while}\ t\ c]\!](\mathcal{C}[\![c]\!](s)) \\
& \textbf{else}\ s
\end{aligned}
$$

Ideal semantics of Cmd:

- semantic domain: $S \rightarrow S_\perp$
- contains (s, s') if c terminates
- contains $(s, \perp)$ if c diverges

Outline

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

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

Non-recursive, finite subfunctions

$$fac_0(n) = \perp$$

$$fac_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot fac_0(n-1) & \text{otherwise} \end{cases}$$

$$fac_2(n) = \begin{cases} 1 & n = 0 \\ n \cdot fac_1(n-1) & \text{otherwise} \end{cases}$$

$$\ldots$$

$$\begin{aligned} fac_0 &= \{\} \\ fac_1 &= \{(0,1)\} \\ fac_2 &= \{(0,1),(1,1)\} \\ fac_3 &= \{(0,1),(1,1),(2,2)\} \\ &\ldots \end{aligned}$$

$$fac = \bigcup_{i=0}^{\infty} fac_i$$

Fine print:

- each fac_i maps all other values to $\perp$
- $\cup$ operation prefers non-$\perp$ mappings

Computing the fixed point

In general

$$fac_0(n) = \perp$$

$$fac_i(n) = \begin{cases} 1 & n = 0 \\ n \cdot fac_{i-1}(n-1) & \text{otherwise} \end{cases}$$

A template to represent all fac_i functions:

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

takes fac_{i-1} as input

Fixpoint operator

$\mathbf{fix} : (A \to A) \to A$

$\mathbf{fix}(g) = \mathbf{let}\ x = g(x)\ \mathbf{in}\ x$

$$\mathbf{fix}(h) = h(h(h(h(h(\ldots)))))$$

Factorial as a fixed point

$fac = \mathbf{fix}(F)$

Outline

Why domains are not flat sets

Internal structure of domains supports the least fixed-point construction

Recall fine print from factorial example:

- each fac_i maps all other values to $\perp$
- $\cup$ operation prefers non-$\perp$ mappings

How can we **generalize** and **formalize** this idea?

Partial orderings and joins

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

- **reflexive**: $\forall x \in D. \quad x \sqsubseteq x$
- **antisymmetric**: $\forall x, y \in D. \quad x \sqsubseteq y \ \wedge \ y \sqsubseteq x \implies x = y$
- **transitive**: $\forall x, y, z \in D. \quad x \sqsubseteq y \ \wedge \ y \sqsubseteq z \implies x \sqsubseteq z$

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 $d \sqsubseteq c$

(Scott) domains are directed-complete partial orderings

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

- every directed subset (related by $\sqsubset$) of a domain has $\perp$

The meaning of a (Scott-continuous) recursive function f is: $\displaystyle\bigsqcup_{i=0}^{\infty} f_i$

where f_i are the finite approximations of f

Well-defined semantics for the while statement

Syntax

$$
\begin{array}{rcll}
t \in \mathit{Test} & ::= & \ldots & \\
c \in \mathit{Cmd} & ::= & \ldots & \mid \ \texttt{while}\ t\ c
\end{array}
$$

Semantics

$$
\begin{array}{rcl}
T[\![\, \mathit{Test}\,]\!] & : & S \to \mathit{Bool} \\
C[\![\, \mathit{Cmd}\,]\!] & : & S \to S \\
C[\![\texttt{while}\ t\ c]\!] & = & \mathbf{fix}(\lambda f.\lambda s.\ \mathbf{if}\ T[\![t]\!](s)\ \mathbf{then}\ f(C[\![c]\!](s))\ \mathbf{else}\ s)
\end{array}
$$