# Domain Theory

## Part 1: First Steps to Scott Domains

Dr. Liam O'Connor

based on material from
Graham Hutton, Dana Scott, Joseph E. Stoy, Carl Gunter, Glynn Winskel

March 4, 2024

## 1 Denotational Semantics

In this course we are concerned with *programming languages*, which, as with natural languages, consist of a syntax and semantics.

The syntax of a language is for our purposes just an inductively-defined tree structure (i.e. abstract syntax), defined using notation similar to BNF.

**Example** (A Simple Imperative Language)

Let's define a syntax for a simple imperative language, called $\mathcal{C}$:

$$\begin{array}{lcl} \mathcal{E} & ::= & n \mid x \mid \mathcal{E}_1 + \mathcal{E}_2 \mid \mathcal{E}_1 - \mathcal{E}_2 \\ \mathcal{B} & ::= & \mathsf{false} \mid \mathsf{true} \mid \neg\mathcal{B} \mid \mathcal{E}_1 = \mathcal{E}_2 \\ \mathcal{C} & ::= & \mathsf{skip} \mid x := \mathcal{E} \mid \mathcal{C}_1; \mathcal{C}_2 \mid \mathsf{if}\ \mathcal{B}\ \mathsf{then}\ \mathcal{C}_1\ \mathsf{else}\ \mathcal{C}_2 \\ x & \in & \mathcal{V} \quad \textit{(variables)} \\ n & \in & \mathbb{Z} \end{array}$$

Domain theory comes in handy when we use denotational semantics. The structure of such a semantics corresponds to the syntactic structure of the language. In the example above, there are three *syntactic categories*[1]: $\mathcal{E}$, $\mathcal{B}$, and $\mathcal{C}$. A denotational semantics consists of, for each syntactic category $\mathcal{X}$:

1. A semantic domain $D$ which can be any set of mathematical objects (although, as we will see later, it is helpful if it obeys certain properties).
2. A valuation function $[\![\cdot]\!] : \mathcal{X} \to D$. We require that this function be a homomorphism, that is that it is compositional. This means that the denotation (or valuation) of an expression should be made from the denotation of its components. In other words, for each syntactic constructor $\mathsf{C}(e_1, e_2, \ldots)$, we should have a mathematical operation $f(x_1, x_2, \ldots)$ such that the denotation of C can be defined as:

$$[\![\mathsf{C}(e_1, e_2, \ldots)]\!] = f([\![e_1]\!], [\![e_2]\!], \ldots)$$

The choices of these objects is, essentially, arbitrary: we choose objects that reflect those aspects of our programs that we are interested in. For most of the simple languages we will examine, we are only concerned with the *results* of the computation, which is a semantics

[1] not that kind

suitable for reasoning about program behaviour and correctness. However, there also exist denotational *cost models* that compositionally assign a measure of program performance to syntax. This measure is just another kind of denotational semantics.

**Example**

Defining a denotational semantics for our language above, we shall first select a semantic domain for each syntactic category:

$$\begin{array}{lcl} \mathbf{E} & \triangleq & \Sigma \to \mathbb{Z} \\ \mathbf{B} & \triangleq & \Sigma \to \mathbb{B} \\ \mathbf{C} & \triangleq & \Sigma \to \Sigma \end{array}$$

Here, $\Sigma$ represents the set of *states*, which contains the values assigned to all variables:

$$\Sigma \triangleq \mathcal{V} \to \mathbb{Z}$$

Now, we define our valuation functions:

$$\begin{array}{lcl} \llbracket \cdot \rrbracket_{\mathcal{E}} : \mathcal{E} \to \mathbf{E} & & \\ \llbracket n \rrbracket_{\mathcal{E}}\ \sigma & = & n \\ \llbracket x \rrbracket_{\mathcal{E}}\ \sigma & = & \sigma(x) \\ \llbracket e_1 + e_2 \rrbracket_{\mathcal{E}}\ \sigma & = & \llbracket e_1 \rrbracket_{\mathcal{E}}\ \sigma + \llbracket e_2 \rrbracket_{\mathcal{E}}\ \sigma \\ \llbracket e_1 - e_2 \rrbracket_{\mathcal{E}}\ \sigma & = & \llbracket e_1 \rrbracket_{\mathcal{E}}\ \sigma - \llbracket e_2 \rrbracket_{\mathcal{E}}\ \sigma \end{array}$$

$$\begin{array}{lcl} \llbracket \cdot \rrbracket_{\mathcal{B}} : \mathcal{B} \to \mathbf{B} & & \\ \llbracket \mathsf{false} \rrbracket_{\mathcal{B}}\ \sigma & = & \mathsf{F} \\ \llbracket \mathsf{true} \rrbracket_{\mathcal{B}}\ \sigma & = & \mathsf{T} \\ \llbracket \neg b \rrbracket_{\mathcal{B}}\ \sigma & = & \begin{cases} \mathsf{F} & \text{if } \llbracket b \rrbracket_{\mathcal{B}}\ \sigma = \mathsf{T} \\ \mathsf{T} & \text{otherwise} \end{cases} \\ \llbracket e_1 = e_2 \rrbracket_{\mathcal{B}}\ \sigma & = & \begin{cases} \mathsf{T} & \text{if } \llbracket e_1 \rrbracket_{\mathcal{E}}\ \sigma = \llbracket e_2 \rrbracket_{\mathcal{E}}\ \sigma \\ \mathsf{F} & \text{otherwise} \end{cases} \end{array}$$

$$\begin{array}{lcl} \llbracket \cdot \rrbracket_{\mathcal{C}} : \mathcal{C} \to \mathbf{C} & & \\ \llbracket \mathsf{skip} \rrbracket_{\mathcal{C}}\ \sigma & = & \sigma \\ \llbracket x := e \rrbracket_{\mathcal{C}}\ \sigma & = & \sigma\,(x \mapsto \llbracket e \rrbracket_{\mathcal{E}}) \\ \llbracket c_1; c_2 \rrbracket_{\mathcal{C}}\ \sigma & = & \llbracket c_2 \rrbracket_{\mathcal{C}}\ (\llbracket c_1 \rrbracket_{\mathcal{C}}\ \sigma) \\ \llbracket \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rrbracket_{\mathcal{C}}\ \sigma & = & \begin{cases} \llbracket c_1 \rrbracket_{\mathcal{C}}\ \sigma & \text{if } \llbracket b \rrbracket_{\mathcal{B}}\ \sigma = \mathsf{T} \\ \llbracket c_2 \rrbracket_{\mathcal{C}}\ \sigma & \text{otherwise} \end{cases} \end{array}$$

## 2 Recursion

So far, our semantic domains have just been (functions of) *sets*. While these have the advantage of being mathematically simple and intuitive, two language features make these sets insufficient for our purposes:

1. *Recursively defined programs*, for dealing with recursion and loops, and
2. *Recursively defined semantic domains*, for dealing with higher-order programs.

## 2.1 Recursively Defined Programs

Suppose we extended our above example with a while loop construct:

$$\mathcal{C} \ ::= \ \cdots \mid \mathsf{while}\ \mathcal{B}\ \mathsf{do}\ \mathcal{C}\ \mathsf{od}$$

The intuitive way to assign a semantics would be to use a *recursive function*:

$$[\![\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}]\!]_{\mathcal{C}}\ \sigma \ = \ \begin{cases} [\![\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{od}]\!]_{\mathcal{C}}\ ([\![c]\!]_{\mathcal{C}}\ \sigma) & \text{if } [\![b]\!]_{\mathcal{B}}\ \sigma = \mathsf{T} \\ \sigma & \text{otherwise} \end{cases}$$

However, such an equation is not a good definition. If we consider the trivial infinite loop program $L \triangleq \mathsf{while\ true\ do\ skip\ od}$, and compute its semantics, we end up with:

$$[\![\mathsf{while\ true\ do\ skip\ od}]\!]_{\mathcal{C}}\ \sigma \ = \ [\![\mathsf{while\ true\ do\ skip\ od}]\!]_{\mathcal{C}}\ \sigma$$

This equation is satisfied by *any* function $\Sigma \to \Sigma$, so it doesn't tell us which function corresponds to the program L.

More generally, allowing our functions to be (generally) recursive causes these issues. The loop program L gives rise to the recursive equation $\ell(x) = \ell(x)$, which has an infinite number of solutions.

If we add recursion to our programming language, we could have programs that give rise to more complex recursive equations like $f(x) = f(x) + 1$. By contrast to $\ell(x)$, $f(x)$ *has no solutions*[2].

**Upshot**

We need an *explicit* notion of "non-termination" on the semantics level, to properly deal with general recursion (or iteration).

## 2.2 Recursively Defined Semantic Domains

Suppose we extend our notion of expressions with parameterless higher-order (parameterless) procedures:

$$\mathcal{E} \ ::= \ \cdots \mid \mathsf{proc}\ \mathcal{C}$$

**Example** (Higher-order procedures)

We can store procedures in variables, so the program:

$$inc := (\mathsf{proc}\ a := a + 1); inc; inc$$

has the same effects on the variable $a$ as the program:

$$a := a + 1; a := a + 1$$

Our semantic domains now take this form, where blue parts are new, and $\uplus$ denotes disjoint union:

$$\begin{array}{lcl} \mathbf{E} & \triangleq & \Sigma \to \mathbb{Z} \\ \mathbf{B} & \triangleq & \Sigma \to \mathbb{B} \\ \mathbf{C} & \triangleq & \Sigma \to \Sigma \\ \Sigma & \triangleq & \mathcal{V} \to (\mathbb{Z} \uplus \mathbf{C}) \end{array}$$

[2] Assuming $f(x)$ operates on integers

Unfolding the definition of $\Sigma$, we end up with a recursive equation for the definition of $\mathbf{C}$:

$$\mathbf{C} \;=\; (\mathcal{V} \to (\mathbb{Z} \uplus \mathbf{C})) \to (\mathcal{V} \to (\mathbb{Z} \uplus \mathbf{C}))$$

Such equations have *no* set-theoretic solution, even if we weaken equality to mere set isomorphism ($\simeq$).

**Why?**

As a simpler example, consider the recursive equation $X = X \to \mathbb{B}$. Cantor's theorem says that there is no set $X$ such that $X \simeq \mathcal{P}(X)$ (where $\mathcal{P}(X)$ is the power set of $X$), and seeing as $\mathcal{P}(X) \simeq (X \to \mathbb{B})$ it follows that $X = X \to \mathbb{B}$ has no solution.

Any kind of higher-order construct leads to such recursive domain equations.

**Upshot**

There are no (nontrivial) set-theoretic solutions to the recursive domain equations that arise from higher-order language constructs.

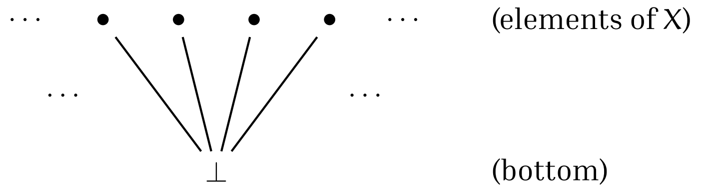
We will return to this problem later on! For now, let's focus on representing non-termination.

## 3 Flat Domains

We shall, following Scott, introduce a special bottom value $\bot$ to each of our elementary semantic domains.

$$\bot \text{ represents } \begin{cases} \text{an undefined value;} \\ \text{an error value;} \\ \text{a } \textit{non-terminating} \text{ computation.} \end{cases}$$

Given a set $X$, the flat domain (or *lifted set*) $X_\bot$ is just the set $X \cup \{\bot\}$ (where $\bot \notin X$). There is a natural information ordering $\sqsubseteq$ on $X_\bot$:




Formally, we say $x \sqsubseteq y$ iff $(x = y \vee x = \bot)$.

**Example**

Consider again our function $f(x) = f(x) + 1$. If we extend addition to operate on the flat domain $\mathbb{Z}_\bot$ such that $\bot + x = \bot$, then this equation has a single unique solution: $f(x) = \bot$. That is, the function $f$ always returns bottom, indicating non-termination.

For our equation $\ell(x) = \ell(x)$, which has an infinite number of solutions, we may now pick the *least* solution in our information ordering, which is similarly $\ell(x) = \bot$.

**Warning**: Don't confuse the information ordering on numbers, i.e. $\sqsubseteq$ on $\mathbb{Z}_\bot$, with the *numerical* ordering $\leqslant$ on $\mathbb{Z}$. We know $0 \leqslant 1$, but 0 and 1 are not comparable in our flat information ordering.

# 4 Combining Domains

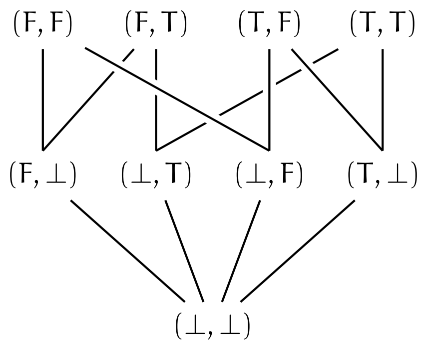
Some of our semantics may depend on the combination of multiple domains, i.e. semantic functions of multiple arguments. As an example, the semantics of an if statement combines the semantics of the condition and the semantics of each of the two branches. Let's consider functions $f : X \times Y \to Z$, where X and Y are flat domains and where $X \times Y$ is the cartesian product of X and Y.

For such semantics, flat domains are no longer sufficient. We may define the information ordering of the product $X \times Y$ in terms of the flat orderings on X and Y separately:

$$(x, y) \sqsubseteq (x', y') \quad \text{iff} \quad x \sqsubseteq x' \wedge y \sqsubseteq y'$$

Intuitively, this says that "the information content of a pair of values is increased by increasing the information of either or both of its component values".

This semantic domain is no longer flat, but it is still a pointed partial order where the bottom value $\bot_{A\times B}$ is $(\bot_A, \bot_B)$. Consider $\mathbb{B}_\bot \times \mathbb{B}_\bot$:




**Theorem**

If two sets X and Y are pointed posets, then so is $X \times Y$ with $\bot_{X\times Y} = (\bot_X, \bot_Y)$.

# 5 Monotonic Functions

If we model our semantic domains for values with pointed posets, then programs are modelled by functions between such posets. But, not all functions are suitable.

**Example**

The function $H : \mathbb{B}_\bot \to \mathbb{B}_\bot$ seems to let us solve the halting problem, assuming $\bot$ represents non-termination:

$$H(v) = \begin{cases} F & \text{if } v = \bot \\ T & \text{otherwise} \end{cases}$$

It stands to reason that the amount of information we get out of our functions should grow as we increase the amount of information we put into them. Formally, we require of a function $f : X \to Y$ between posets X and Y, for all $x, y \in X$:

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y)$$

Such functions are called monotonic. These functions preserve the information ordering, but they are not required to preserve the bottom element $\bot$. Functions for which $f(\bot) = \bot$ are called strict.

**Thesis**

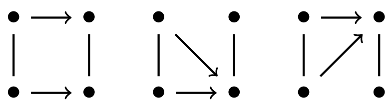
Computable functions are monotonic (observe that H is not).

## Exercises

1. Consider the functions $\mathbb{B}_\perp \to \mathbb{B}_\perp$. Which ones are monotonic? There are a total of 27 such functions but only three significant classes.

2. Let $K_K$ denote the chain of values $x_1, x_2, x_3, \ldots, x_K$ where $a \leqslant b$ implies $x_a \sqsubseteq x_b$. There is one monotonic function $K_1 \to K_1$:

   And there are three monotonic functions $K_2 \to K_2$:



   a) Write down the monotonic functions $K_3 \to K_3$.

   b) Write a simple recursive program to calculate the number of monotonic functions $K_n \to K_m$.

3. Give a semantics to the `proc` construct:

   a) $[\![\mathsf{proc}\ c]\!]_{\mathcal{E}}\ \sigma \ = ?$

   b) $[\![x]\!]_{\mathcal{C}}\ \sigma \ = ?$

## Glossary

**antisymmetric** A relation R is antisymmetric if, for all $x$ and $y$, $x\ R\ y$ and $y\ R\ x$ implies $x = y$ (but sometimes this equality is weakened to some kind of isomorphism). 7

**bijection** A bijection between A and B is a mapping (or homomorphism) $f : A \to B$ and an *inverse* $f^{-1} : B \to A$ such that $f \circ f^{-1}$ and $f^{-1} \circ f$ are identity functions. 7

**BNF** The *Backus-Naur Form* notation for writing grammars. 1

**bottom** An information-free value that is added to our domains to represent undefined or non-terminating computations, written $\perp$. It is always the least value of the information ordering. 4, 5, 7

**cartesian product** The cartesian product of two sets A and B, written $A \times B$, is the set of pairs $\{(a, b) \mid a \in Ab \in B\}$. 5

**compositional** The requirement that the denotation of an expression be defined in terms of the denotations of its subexpressions. 1, 7

**denotational semantics** The definition of semantics by the compositional assignment of a mathematical object to each piece of syntax. 1, 2

**disjoint union** Also called a sum, the union of two sets where "tags" are added to ensure disjointness: $A \uplus B = \{(0, x) \mid x \in A\} \cup \{(1, x) \mid x \in B\}$. 3

**flat domain** A flat domain (or lifted set) $X_\perp$ is just the set $X$ augmented with an additional bottom value $\perp$. 4, 5

**higher-order** *Higher-order* programming constructs allow programs to be treated as first-class citizens, i.e. values. 2–4

**homomorphism** A *structure-preserving* map. In the case of a valuation function, it means that it is compositional. 1, 6, 7

**identity function** A function characterised by $f(x) = x$. 6

**information ordering** An ordering, written $\sqsubseteq$, usually a partial order, on elements of the semantic domain, with the bottom element as the least value.. 4, 5

**monotonic** A function $f : X \to Y$ on posets $X$ and $Y$ is *monotonic* if, for all $x, y \in X$, $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ . 5, 6

**partial order** A partial order on $X$ is a relation $\preceq$ on $X$ which is reflexive, transitive, and anti-symmetric.. 5, 7

**pointed** A set $X$ is pointed if it contains one element $x \in X$. If $X$ is a poset this is the bottom element. 5, 7

**poset** A set $X$ equipped with a partial order on $X$. 5, 7

**power set** The power set of $X$ is the set of all subsets of $X$, often written $\mathcal{P}(X)$. 4

**reflexive** A relation $R$ is reflexive if, for all $x$, $x \mathrel{R} x$. 7

**semantic domain** A set of mathematical objects which model the semantics of a language's syntax. 1, 2, 7

**semantics** The *meaning* of a term in a language. 1, 6, 7

**set isomorphism** Two sets $A$ and $B$ are isomorphic if there exists a bijection between them. . 4

**strict** A function $f : X \to Y$ on pointed sets $X$ and $Y$ is *strict* if $f(\perp_X) = \perp_Y$. 5

**syntax** The grammatical structure of a language, usually represented as an inductively-defined tree structure. 1, 6, 7

**transitive** A relation $R$ is transitive if, for all $x$, $y$ and $z$, $x \mathrel{R} y$ and $y \mathrel{R} z$ implies $x \mathrel{R} z$. 7

**valuation function** A function that assigns to an input expression an element of that expression's semantic domain. Normally required to be compositional, i.e. a homomorphism . 1, 2, 7