Domain Theory and its Topological Connections

Topology in and via Logic project presentation

Alyssa Renata, Paulius Skaisgiris, Sarah Dukic, Joel Saarinen

ILLC, University of Amsterdam
February 1, 2023

Table of Contents

Table of Contents

Denotational Semantics

- Origins of Domain Theory lie in 1970s, work by Dana Scott and Christopher Strachey
- Denotational semantics: assigning "meaning" to a given program/expression or a datatype in a programming language.
- More formally: given a programming language P, for each datatype D (eg. expressions, commands, integers, etc.) in that language, there is a valuation function v that maps a phrase of syntax in that category to a denotation in a semantic structure D - the **domain** of interpretation.

Denotational semantics

Example

- Consider the following program:
  def $g(x)\{$return $x + 4 \stackrel{?}{=} 8\}$
- $g(x) \in \{0, 1\}$, so regardless of input, we can come up with a denotation for g.

Denotational Semantics for datatypes themselves

Syntax of (some) datatypes

$Command$ ::= **if** $Bool$ **then** $Command$ **else** $Command$
  | **while** $Bool$ **do** $Command$ | **def** x **:=** $Value$ | **run** x
  | $Command$ **;** $Command$ | **skip**

$Bool$ ::= **tt** | **ff** | x | $Bool$ **and** $Bool$ | $Bool$ **or** $Bool$ | ...

Int ::= **0** | **1** | ... | x | - Int | Int **+** Int | ...

$Value$ ::= $Bool$ | Int | $Command$

Semantics of (some) datatypes

$[\![Command]\!] = State \rightarrow State$
$State = Vars \rightarrow [\![Value]\!]$
$[\![Bool]\!] = \mathbb{B}$
$[\![Int]\!] = \mathbb{Z}$
$[\![Value]\!] = [\![Bool]\!] \cup [\![Int]\!] \cup [\![Command]\!]$

[Plotkin, 1983, p. 1-2]

Issues with denotational semantics: infinitely looping functions

- ▸ What happens if we try to associate a function/value with the following program?

- ▸ $f : \mathbb{N} \to \mathbb{N}, \ f(x) = f(x) + 1$

- ▸ ```
  def f(x) {
      return f(x) + 1
  }
  ```

- ▸ It will loop and not map to a single number:

$$f(m) = f(m) + 1$$
$$f(m) = f(m) + 1 + 1$$
$$f(m) = f(m) + 1 + 1 + 1$$

# Issues with denotational semantics: recursively generated semantic spaces

- A similar problem arises when we try to come up with semantics for recursively generated structures. For example, as the valuation for the datatype **value** is defined as follows:
  $[\![Value]\!] = [\![Bool]\!] \cup [\![Int]\!] \cup [\![Command]\!]$
  where
  $[\![Command]\!] = State \rightarrow State$
  $State = Vars \rightarrow [\![Value]\!]$
  $[\![Bool]\!] = \mathbb{B}$
  $[\![Int]\!] = \mathbb{Z}$

- Since one of the terms in finding the meaning of the datatype Value requires us to find the meaning of Value again, the same procedure is repeated indefinitely, not settling upon a single interpretation.

- Moreover, if: $|state| = n$ then $|state \rightarrow state| = n^n$.

[Vickers, 1989, p. 134]

# Criteria for a solution: fixpoints

- ▶ Observation: every recursively defined function can be expressed as a non-recursive function.

- ▶ Take $f : \mathbb{N} \to \mathbb{N}, f(x) = f(x) + 1$ as seen in the previous slides. We can define a non-recursive higher-order function $\Phi$, where $\Phi(f) = z \mapsto f(z) + 1$. We can then rewrite $f$ as $f = \Phi(f)$.

- ▶ If we let $\Phi : A \to A$, then $x \in A$ is called a *fixed point* of $\Phi$ if $\Phi(x) = x$. [Plotkin, 1983, p. 4]

- ▶ Idea: element in semantic space that a given recursive function is mapped to could be the fixed point of the non-recursive function $\Phi$ that we rewrite $f$ in terms of.

- ▶ However, a function might have no fixpoints, or rather several - so how do we define a denotational semantics that captures these cases?

# Fixpoints for recursively generated semantic spaces

- Recall: finding $[\![Value]\!] \coloneqq [\![Bool]\!] \cup [\![Int]\!] \cup [\![Command]\!]$
  involves finding $[\![Command]\!]$. But
  $[\![Command]\!] = state \to state$, where

$$state = var \to [\![Value]\!] \qquad (1)$$

- We can rewrite (1) as:
  $state = var \to [\![Bool]\!] \cup [\![Int]\!] \cup state \to state$

- Problem simplifies to:

$$state \cong state \to state \qquad (2)$$

# Fixpoints for recursively generated semantic spaces

- We showed that we arrive at the above difficulty by trying to evaluate the meaning of Value. However, one encounters similar difficulties with semantics of other datatypes, having to deal with equations similar to (2), only with mathematical constructions other than "state".

- Ultimately, we seek to solve:

$$D \cong D \to D$$

# Evolution of the datatype

datatype $\longrightarrow$ ? $\longrightarrow$ ?

Non-trivial
solution to $\longleftarrow$ ?
$D \cong [D \to D]$

# Table of Contents

# Proposed solution: partial functions

- Problem: we cannot use total functions to map between datatypes because there are functions with no fixpoints.
- Solution: use partial functions. We can take progressively better finite approximations of our infinitely recurring function, and take the limit of these.

## Example

See blackboard.

# The structure of datatypes: DCPO's

The datatype requirements presented in the following slides closely follow Scott's original paper [Scott, 1970] as well as George Hutton's notes [Hutton, 1994]

1. A datatype is partially ordered.
   - Want to represent that one datatype might contain the same information as another. $f \sqsubseteq g$ captures the intuition that $g$ is a consistent extension of $f$.
   - Set theoretically, $f \sqsubseteq g \iff f \subseteq g$. So $g$ can compute what $f$ can, and more: e.g. $f = \{(0,1),(1,2)\}$ and $g = \{(0,1),(1,2),(2,3)\}$

# The structure of datatypes: DCPO's

2. Datatypes are directed complete, with a bottom element.

  ▸ A subset $X \subseteq D$ where any two points $x, y \in X$ have an upper bound $z \in X$.



  [Hutton, 1994, Lec. 2]

  ▸ We want our datatypes to have consistent specifications of information.
  ▸ For any directed subset, we want an element containing all its information: a least upper bound.
  ▸ Thus, every directed subset of a datatype has a least upper bound. Our datatypes are therefore **directed complete partial orders**.



  [Hutton, 1994, Lec. 2]

# Mappings on datatypes

3. Mappings between datatypes are monotonic.
   - A function $f : D' \to D$ should be sensitive to the accuracy of the input.
   - Consider $\phi(f)$ versus $\phi(g)$ where $f := \{(0,1), (1,1)\}$ and $g := \{(0,1), (1,1), (2,2)\}$. Then $\phi(g)$ is defined whenever $\phi(f)$ is defined, but the converse is not true. So
   $$f \sqsubseteq g \to \phi(f) \sqsubseteq \phi(g)$$

# Mappings on datatypes

4. Mappings between datatypes are continuous.
   - We want functions to preserve limits: "finite" information in the output should entail "finite" information in the input.
   - LUB's of directed sets should be preserved:
     $f : D \to D'$ is continuous iff $f(\bigsqcup X) = \bigsqcup\{f(x) : x \in X\}$

     

     [Hutton, 1994, Lec. 2]

   - Continuity gives us the **DCPO fixpoint theorem:** Any continuous function $f : A \to A$ on a DCPO $A$ has a LFP computed as the limit of $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$ , i.e.
     $LFP(f) = \bigsqcup\{f^n(\bot) | n \in \mathfrak{N}\}$

     [Abramsky and Jung, 1994, p. 16]

# Summary

- We now have the desired structure for datatypes. DCPO's both capture our intuitions about how information should behave, and also gives us a way of specifying the denotation of recursive functions non-recursively.
- The presence of a bottom element lets us characterise functions with no output
- Computable functions are monotonic and continuous.
- The DCPO fixpoint theorem tells us that there is always an LFP: we have a denotation for any function.

# Evolution of the datatype

datatype $\longrightarrow$ **DCPO** $\longrightarrow$ ?

Non-trivial
solution to $\longleftarrow$ ?
$D \cong [D \to D]$

# Table of Contents

# Topological Intuitions

- Any computable function must be *monotone* and *preserves directed joins*.
- Informally, directed joins = limits, so join-preservation = continuity.
- Can we make this analogy formal?

# Alexandroff Topology

### Reminder
*Given partial order $(P, \leq)$, $U$ is open in the Alexandroff topology on $P$ iff $U$ is upwards closed (if $x \in U$ and $x \leq y$ then $y \in U$).*

- Alexandroff-continuity = monotone
- But what about join-preservation?

# Not all Alexandroff-continuous functions preserve d-joins

Consider the following monotone function from $\omega + 1$ to $\omega + 2$:

# Not all Alexandroff-continuous functions preserve d-joins

**General Problem:** Even if $f(\bigsqcup A) \neq \bigsqcup f[A]$, $f$ can still be continuous around $f(\bigsqcup A)$ - for any open neighborhood $U$ of $f(\bigsqcup A)$, the image of the open neighborhood $V = \uparrow \bigsqcup A$ lies in $U$.

# The Remedy

**Want**: For every open neighborhood $V$ of $\bigsqcup A$, $f[V]$ must contain $\bigsqcup f[A]$. **Solution**: Force every such $V$ to contain an element of $A$.

### Definition (Scott Topology)

*Let $D$ be a DCPO. We define the **Scott topology** $\sigma_D$ on $D$ by defining the **Scott-open** sets as follows.*

1. *$U \in \sigma_D$ iff (i) $U$ is upwards closed and (ii) for any directed set $A$, if $\bigsqcup A \in U$ then $U \cap A$ is non-empty.*
2. *When $U$ satisfies condition (ii) above, we say that $U$ is **inaccessible by directed joins**.*

*[Gierz et al., 2003, p. 134]*

### Proposition

*$S$ is **Scott-closed** iff $S$ is downwards closed and closed under directed joins.*

# The Scott Topology

- Intuition: open set = finitely observable property. Hence, if we can finitely observe a property of $\bigsqcup A$ then this property should already be evident in some component $a \in A$ of $\bigsqcup A$.
- The following theorems formalise the idea that directed joins = limits and preservation of directed joins = continuity.

## Theorem

1. Let $A$ be a directed set in DCPO $D$. Then in the Scott topology $\sigma_D$, $\bigsqcup A$ is a limit of the filter generated by closing $\{\uparrow a \cap A \mid a \in A\}$ upwards.

2. $f : D \to E$ is continuous under the Scott topology iff it is monotone and preserves directed joins.

## Proof.
See blackboard. $\qquad \square$

# Example of Scott Topology: $\mathbb{N} \rightharpoonup \mathbb{N}$

- ▸ The Scott topology on the DCPO of partial functions is generated by subbasic opens of the form $\uparrow \{(m, n)\}$.

- ▸ Closure under intersections give upsets of all finite partial functions:

$$\uparrow \{(m_0, n_0), \ldots (m_k, n_k)\} = \uparrow \{(m_0), n_0)\} \cap \ldots \cap \uparrow \{(m_k, n_k)\}$$

- ▸ Closure under union generates all the Scott open sets, but notice that we never end up generating upsets of infinite partial functions - any Scott open set contains a finite partial function.

# Table of Contents

# Up-sets are no longer open

- In general, $\uparrow x$ is not open in the Scott Topology.
- As a substitute, we can still take the interior $int(\uparrow x)$.
- Intuition: $y \in int(\uparrow x)$ means $x$ is relatively small (often this even means finite) compared to $y$ - i.e. $x$ is wayyy below $y$.

### Definition
$x$ **is way below** $y$ (denoted $x \ll y$) iff for any directed $A$, $y \sqsubseteq \bigsqcup A$ implies there is some $a \in A$ s.t. $x \sqsubseteq a$.

### Proposition
$y \in int(\uparrow x)$ implies $x \ll y$. In particular, if $\bigsqcup A \in int(\uparrow x)$ then there is some $a \in A$ such that $x \sqsubseteq a$.

### Example
Consider the DCPO of partial functions on $\mathbb{N}$. Then
$\{(0,0),(1,2)\} \ll (x \mapsto 2x)$ but $(x \text{ even} \mapsto 2x) \not\ll (x \mapsto 2x)$.

# Compact Elements

When $x \ll x$, then $x$ is a small approximation of itself, which is possible only if $x$ is small in some absolute sense.

## Definition
*If $x \ll x$, we say that $x$ is **compact**. If $D$ is a DCPO, let $D_c = \{x \in D | x \ll x\}$ be the set of its compact elements.*

## Example

- Any finitely defined partial function is compact.
- If $D$ is a finite or flat DCPO, $D = D_c$.
- In $\omega + 1$, only $\omega$ is not compact.
- More generally, in any ordinal DCPO $\alpha + 1$, the compact elements are the successor ordinals and 0.

# Algebraicity

- If we are to interpret a datatype as a DCPO, then every element must be computable as a limit of finitely computable elements.
- Compactness is an abstraction of being finitely computable.

### Axiom (Algebraicity)

*A datatype must have a "basis" of compact elements: for each $x \in D$, the set $approx(x) = \downarrow x \cap D_c$ must be directed with $x = \bigsqcup approx(x)$.*

### Example

Many of the standard DCPOs such as the DCPO of partial functions etc. are algebraic.

# Algebraic DCPOs are determined by their compact elements

### Proposition

1. Let $D$ and $E$ be algebraic DCPOs. Then $f : D \to E$ is continuous iff $f(x) = \bigsqcup f[approx(x)]$.

2. Let $D$ and $E$ be DCPOs with $D$ algebraic. Each monotone function $f : D_c \to E$ extends uniquely to a continuous $\overline{f} : D \to E$.

# Evolution of the datatype

datatype $\longrightarrow$ DCPO $\longrightarrow$ **algebraic DCPO**

Non-trivial
solution to $\longleftarrow$ ?
$D \cong [D \to D]$

# Table of Contents

# Semantics for function types is hard

- Clearly, having function types is useful as it provides semantics for *command*
- This was a source of difficulty because the amount of all possible functions ($n^n$) is far more than the amount of states we have ($n$)

# Semantics for function types is hard

Now that we have some tools and definitions, we can attempt to fully untangle this problem. What we want:

- "Carve out" computable functions as these functions will be executed on a physical machine
  - We know that computable functions are monotone and continuous
- We need the set of all functions (function space) between two datatypes to also be a datatype
  - Thus far, datatype = algebraic DCPO
  - So, remains to show that for two algebraic DCPOs $D, E$, the set of all monotone and continuous functions from $D$ to $E$, $[D \to E]$, is an algebraic DCPO.

## Theorem (?)

*Let $D, E$ be algebraic DCPOs. Then, $[D \to E]$ is an algebraic DCPO.*

# What DCPO structure should $[D \to E]$ have?

- The set of all functions $D \to E$ is essentially the product $\Pi_{x \in D} E$ of $D$ copies of $E$, so we can equip it with the product topology constructed from the Scott topology of each $E$.

- We consider the monotone and continuous subspace of $\Pi_{x \in D} E$, denoted $\int_{x \in D} E$.

### Definition
*The topological space $\int_{x \in D} E$ of **pointwise convergence** on $[D \to E]$ is the topology generated by the basis*

$$\left\{ \int_{x \in D} U_x \mid \forall x \in D. U_x \in \sigma_E \right\}$$

*with the condition that only finitely many $U_d \neq E$ in each $(U_x)_{x \in D}$.*

### Proposition
*Given a filter $F$ in $\int_{x \in D} E$, $F \to f$ iff $F(x) \to f(x)$ for each $x \in D$.*

# What DCPO structure should $[D \to E]$ have?

The order induced by the pointwise convergence topology on $[D \to E]$ yields the following DCPO:

## Proposition

*If D and E are DCPOs, then the partial order on $[D \to E]$ defined as*

$$f \sqsubseteq g \iff \forall x \in D.f(x) \sqsubseteq g(x)$$

*is a DCPO, with $(\bigsqcup F)(x) = \bigsqcup F(x)$.*

**TO DO**: Citation

Unfortunately, it's **NOT** algebraic.

# $[D \to E]$ is not necessarily algebraic

- A monotone continuous function $f : D \to E$ is constructed as the limit of compact approximations of the form

$$\langle d; e \rangle(x) := \begin{cases} e & \text{if } x \sqsubseteq d \\ \bot & \text{otherwise} \end{cases}$$

- However, the set of compact approximations $approx(f)$ is not necessarily directed: consider how one constructs a compact upper bound of $\langle d_1; e_1 \rangle$ and $\langle d_2; e_2 \rangle$.



[Abramsky and Jung, 1994, p. 54-55]

# $[D \rightarrow E]$ is not necessarily algebraic



- The issue is that we cannot find what the upper bound of $\langle d_1; e_1 \rangle$ and $\langle d_2; e_2 \rangle$ should map to when given an element of $A = \uparrow d_1 \cap \uparrow d_2$, although ideally it would be $e_1 \sqcup e_2$.
- $e_1$ and $e_2$ are arbitrary elements, other than the fact that they are upper bounded by $f(a)$ where $a$ is some element of $A$. Hence, to fix this, we require the existence of certain additional least upper bounds.

[Abramsky and Jung, 1994, p. 54-55]

# Scott Domains

### Definition

*A partial order is **consistently complete** iff every set with an upper bound has a least upper bound (join).*

### Axiom (Consistently Complete)

*A datatype has to be consistently complete.*

This concludes our search for a suitable class of structures to represent domains of interpretation, for we are now in the position to solve the $D \cong [D \to D]$ equation.

### Definition (Scott Domains)

*A partial order is a (Scott) domain iff it is an **a**lgebraic, **c**onsistently **c**omplete, **d**irected **c**omplete **p**artial **o**rder with a bottom element.*

a c3po →

# Function space as a datatype revisited

### Theorem
*Let $D, E$ be domains. Then the function space $[D \to E]$ is a domain.*

*Proof.* By putting blind faith in us. (We don't have time for the details of the rest of the proof)

# Evolution of the datatype

datatype $\longrightarrow$ DCPO $\longrightarrow$ algebraic DCPO

Non-trivial
solution to
$D \cong [D \to D]$ $\longleftarrow$ consistently-complete
algebraic DCPO
(Scott Domains)

# Table of Contents

## Domain equations

The last thing remaining is to find a general datatype to serve as a model for denotational computation. Essentially, we want our datatype to be a solution to the equation:

$$D \cong [D \to D]$$

Note that the trivial solution is $D = \{\bot\}$, but, clearly, we want a more interesting solution.

## Non-trivial solution: $D_\infty$

Let $D$ be a domain. Set $D_0 = D$ and define inductively $D_n$ for each $n$ by

$$D_{n+1} \cong [D_n \to D_n]$$

"It turns out that there is a **natural** way of isomorphically embedding each $D_n$ successively into the next space $D_{n+1}$"

(One could make a whole presentation on this "natural" isomorphism; for a more in-depth proof check [Stoltenberg-Hansen et al., 1994, Ch. 12.3] and [Gierz et al., 2003, Ch. IV-7])

These embeddings allows us to take the limit of this equation, obtaining the limit space

$$D_\infty \cong [D_\infty \to D_\infty]$$

[Scott, 1970]

# Non-trivial solution: $D_\infty$

The construction idea starts by taking the limit of a sequence of domains obtained by iterating the function space construction.

That is, take $f = (f_0, f_1, f_2, \ldots)$. We may apply this function sequence onto itself getting $(f_1(f_0), f_2(f_1), f_3(f_2), \ldots)$.

[Stoltenberg-Hansen et al., 1994, p. 319]

# Non-trivial solution: $D_\infty$

This is a solution to the self-application problem because

- ▶ We have restricted the amount of functions considered to allow for $|D_\infty| = |[D_\infty \to D_\infty]|$



[Hutton, 1994, Lec. 5]

# Non-trivial solution: $D_\infty$

This is a solution to the self-application problem because

- We have restricted the amount of functions considered to allow for $|D_\infty| = |[D_\infty \to D_\infty]|$
- Each element of $D_\infty$ can be regarded as a continuous function on $D_\infty$ into $D_\infty$, and every such continuous function can be regarded as an element.

# Significance of $D_\infty$

### Quote

"Finding a non-trivial model of the untyped $\lambda$-calculus was Scott's original motivation for developing domain theory. The construction of such a model in 1972 is one of the most significant results in the history of theoretical computer science." [Hutton, 1994, Lec. 5]

### Quote

"Technically speaking, what we have here is the first known, 'mathematically' defined model of the so-called $\lambda$-calculus of Curry-Church." [Scott, 1970]

# Table of Contents

# Closing summary

- Domain theory is a field providing necessary tools for giving denotational semantics to programming languages (or, $\lambda$-calculus, generally)
- Partial orders with extra structure are chosen as the basic elements to represent datatypes, coined Scott Domains
- This order induces a topology which can be used as a supplement to order-theoretic treatment of the theory
- We illustrated a way to provide semantics to `command` by defining $D_\infty$: a domain which is isomorphic to its function space

# References I

📄 Abramsky, S. and Jung, A. (1994).
Domain theory.
In Abramsky, S., Gabbay, D. M., and Maibaum, T. S. E.,
editors, *Handbook of Logic in Computer Science*, volume 3,
pages 1–168. Clarendon Press.

📄 Gierz, G., Hofmann, K., Keimel, K., Lawson, J., Mislove, M.,
and Scott, D. (2003).
*Continuous Lattices and Domains*.
EBSCO ebook academic collection. Cambridge University
Press.

📄 Hutton, G. (1994).
Introduction to domain theory.

📄 Plotkin, G. (1983).
*Domains*.
University of Edinburgh.

# References II

📄 Scott, D. (1970).
*Outline of a mathematical theory of computation*.
Oxford University Computing Laboratory, Programming Research Group Oxford.

📄 Stoltenberg-Hansen, V., Lindström, I., and Griffor, E. R. (1994).
*Mathematical Theory of Domains*.
Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

📄 Vickers, S. (1989).
*Topology via Logic*.
Cambridge University Press, USA.