Formalising Domain Theory with Lean

Anna Williams

1 Introduction

One of the key motivations for domain theory, as established in the seminal paper by Dana Scott [Sco70], is that of unrestricted functions. These are functions to which we wish to apply any argument, including the function itself. This is inconsistent with set theory. For example, take a function $f : A \to B$, then if we want to consistently apply f to itself, notice that f must belong to the set A too. However $A \neq (A \to B)$ for any values of A and B.

Domain theory is now a rich field with major applications in computer science, particularly in the theory of programming languages.

1.1 Why Formalisation?

To look at arguments for formalisation of mathematics, we examine the contexts of existing and new proofs.

Existing Proofs

- We can back up existing proofs.
- For longer and harder to understand proofs, it is easier to accept a formalised proof in which you only have to trust the proof assistant, than it is to understand and accept the proof as a whole. Though this may mean you don't understand the proof as well, it allows you focus on the key ideas instead.

New Proofs

- We can create new proofs that are more trustworthy or are completely new. For example, the proof of the four colour theorem by Georges Gonthier is more well accepted than previous less precise computer proofs using software which had no in built checks for correctness.
- It makes writing new proofs with larger numbers of participants easier as all proofs are checked by the proof assistant as opposed to having to be checked by other participants. This could help make mathematics more accessible to less experienced mathematicians.

1.2 Aims and Goals

The main aim of this project was to explore the possibilities of formalisation of domains using Lean, with some intent to contribute to the mathlib4 library to expand some of the key definitions and work in this area.

2 Domain Theory

Dana Scott motivates a lot of the key ideas behind Domain Theory well, so we will follow the reasoning he gives in his initial paper [Sco70]. We focus in particular on the "data types" that we base computations on and explore how these are structured.

2.1 Data Types

When talking about functions, or computations, we want to talk about the underlying set, or data type, we are operating with. First exploring the structure of a data type, notice that for any two elements of our underlying set, there might exist a relation between them; call this $\sqsubseteq$. This could be a sort of estimation – for example a finite sequence can "estimate" infinite sequence by being correct to n digits, a lower bound can estimate a target value, etc. Recall that a relation is defined as follows.

Definition 2.1. A *relation* on a set A, is a subset $R \subseteq A \times A$. To denote that $(a, b) \in R$, we write $a \, R \, b$.

So then, what properties does this relation have? Naturally we would want any item to estimate itself (reflexivity), if we have a chain of estimations, then we should have that the start of the chain estimates the end (transitivity), and finally if two elements estimate one another then they must be equal (anti-symmetry). This leads us to our first key definition.

Definition 2.2 (Partially Ordered Set)**.** A *partially ordered set*, or poset for short, consists of an underlying set D and relation on the set, given by $\sqsubseteq$, such that

i) $\forall x \in D, x \sqsubseteq x$,

ii) $\forall x, y, z \in D$, where $x \sqsubseteq y$ and $y \sqsubseteq z$, then $x \sqsubseteq z$, and

iii) $\forall x, y \in D$, where $x \sqsubseteq y$ and $y \sqsubseteq x$, then $x = y$.

Note. Notice that i) is exactly reflexivity, ii) is transitivity and iii) is anti-symmetry.

Throughout this section, we will use the natural numbers with infinity, $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$, and the relation $\leq$ as an example to try to support in the understanding of definitions.

Example 1. $\mathbb{N}_\infty$ with $\leq$ is a poset.

Now that we have a relation, consider the following sequence:

$$x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_i \sqsubseteq x_{i+1} \sqsubseteq \ldots$$

Then we might say that the x_n are tending towards a limit. Say that the limit is x, then we write $\bigsqcup x_n = x$. We first look at sets which have least upper bounds for finite subsets. These are called directed sets, as they have a "direction" which they go in.

Definition 2.3. A *directed set* is a non-empty set, A, such that for all pairs $x, y \in A$ there is a least upper bound z such that $x \sqsubseteq z$ and $y \sqsubseteq z$.

Notation. We write $A \subseteq_{\text{dir}} D$ to mean that A is a directed subset of D.

Example 2. Continuing our example with $\mathbb{N}_\infty$, we can prove that every subset is a directed set.

Proof. We need to show that for any given nonempty subset $A \subseteq \mathbb{N}_\infty$ for all $x, y \in A$, there is some $z \in A$ such that $x \leq z$ and $y \leq z$. Notice that $\max(x, y)$, defined in the usual way satisfies this property. First see that $\max(x, y) \in A$ as $\max(x, y) = x$ or $\max(x, y) = y$. Then notice that $x \leq \max(x, y)$ and $y \leq \max(x, y)$ as required. □

Now expanding to include least upper bounds for infinite sets, we find the definition for a dcpo.

Definition 2.4 (Directed Complete Partial Order)**.** A *directed complete partial order*, or dcpo for short, is a poset such that every directed set has a least upper bound.

Example 3. Continuing with our example of $\mathbb{N}_\infty$, we need to show that every directed set (so every set) has a least upper bound.

Proof. We split into two cases: where the set is finite and where the set is infinite. In the case that the set is finite, we simply take the maximum element of the set. In the case the set is infinite, there is no upper bound within the natural numbers and so the least upper bound is ∞. □

We now define approximation, a new relation based upon the existing one which "gives some room" between elements.

Definition 2.5. We say that x *approximates* y, and write $x \ll y$, if for all $A \subseteq_{\text{dir}} D$ such that $y \sqsubseteq \bigsqcup A$, then there exists $a \in A$ such that $x \sqsubseteq a$.

Example 4. Looking at $\mathbb{N}_\infty$, we can see that $x \ll y$ if $x \leq y$, except that $\infty \not\ll \infty$.

Proof. Let $x, y \in \mathbb{N}_\infty$ and assume that $x \leq y$. We now split into two cases, where $y = \infty$ and where $y \neq \infty$.

Let $A \subseteq \mathbb{N}_\infty$, with $y \leq \bigsqcup A$. If A is finite, then we simply pick the maximum element $a = \bigsqcup A \leq y$ and $x \leq y \leq a$, so $x \leq a$. If A is infinite, then there must be some element $a \in A$ which is larger than y and as $x \leq y \leq a$, $x \leq a$. □

This finally leads us onto our definition for a continuous domain.

Definition 2.6 (Continuous Domain)**.** A *continuous domain* is a dcpo such that the set $\twoheaddownarrow x := \{a \mid a \ll x\}$ is directed and such that $\bigsqcup \twoheaddownarrow x = x$.

That is every element can be reconstructed via its constituent parts (those that approximate it) [ES99].

Example 5. We now prove that $\mathbb{N}_\infty$ is a continuous domain.

Proof. Directed by the fact that every set is directed. If $x \neq \infty$, then $x \ll x$, so the set $\twoheaddownarrow x = \{a | a \leq x\}$, thus the least upper bound is x. If $x = \infty$, then $\twoheaddownarrow \infty = \mathbb{N}$ and thus the least upper bound is ∞. □

2.2 The Interval Domain

One interesting domain is the interval domain because it gives the real numbers a computational structure as per [ES99]. It is the collection of closed intervals on the real numbers, with a least element (the real numbers).

$$I = \{[a, b] \mid a, b \in \mathbb{R} \text{ and } a \leq b\} \cup \{\mathbb{R}\}$$

We order these by the superset relation, that is $\mathbb{R} \sqsubseteq A$ for all $A \in I$ and $[a, b] \sqsubseteq [c, d]$ if $a \leq c$ and $d \leq b$. We can demonstrate this domain visually using a triangle, where the widest

ranges are towards the bottom and the top line consists of single element ranges, such as $[2, 2]$. We place the element $[a, b]$ in the triangle, such that a line drawn parallel to the left edge of the triangle intersects $[a, a]$ at the top line and a line drawn parallel the right of the triangle intersects $[b, b]$ at the top line.

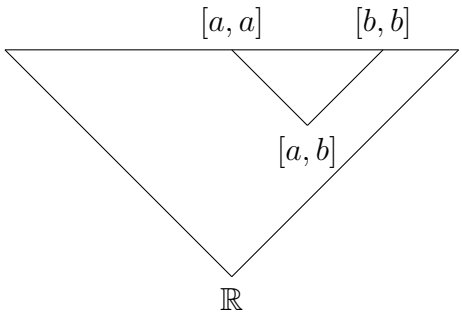

Figure 1: The interval domain, with element $[a, b]$ labelled.

3 Lean

Let's take a look at the tool we are working with to formalise the definitions and examples given in the previous section. There are few different steps involved with formalisation, namely stating definitions, stating theorems and proving theorems.

3.1 Stating Theorems

We first look at stating theorems, which is the simplest of the three and allows us to get familiar with the notation. To get started, we will look some of the important symbols we use to formalise statements.

Lean symbol	Natural language
$\rightarrow$	Implies
$\wedge$	And
$\vee$	Or
$\exists$	Exists
$\forall$	For all

Now to show how we use these, we will state an example theorem statement we might want to formalise and the corresponding Lean statement.

Theorem 3.1. *For all $n, m \in \mathbb{N}$, such that $n \leq m$ there exists $p \in \mathbb{N}$ such that $n + p = m$.*

```
theorem lessThanImpliesDiff : ∀ n m : ℕ, n ≤ m → ∃ p : ℕ, n + p = m
```

We can split this statement up into chunks to see what each section does.

Code	In plain text
`theorem lessThanImpliesDiff : ...`	State a new theorem called "lessThanImpliesDiff"
`∀ n m : ℕ, ...`	for all $n, m \in \mathbb{N}$, ...
`n ≤ m → ...`	$n \leq m$ implies that ...
`∃ p : ℕ, n + p = m`	there exists $p \in \mathbb{N}$, such that $n + p = m$

3.2 Stating Definitions

Stating definitions is a little more complex. We mainly use the following methods:

- An inductive data type,
- a structure, which restricts existing data types, and
- a new definition which builds upon existing definitions.

For an example of an inductive data type, we define the two item set. When using this method, we give it a name and list the ways of constructing an element of it. In this case, an element is either bot ($\bot$), or top ($\top$), this gives us the following definition.

```
inductive TwoSet where
  | bot
  | top
```

For an example using a structure, suppose we want to define the even numbers, then this is exactly given by the natural numbers which satisfy the rule divisible by two. So we base our structure off of a natural number n and restrict it such that we only have the n which can be represented by 2 times another natural number z. This gives us the following definition.

```
structure Even (n : Nat) where
    divByTwo : ∃ z : Nat, n = 2 * z
```

For our last method, we define a predicate. This takes an element of a type and returns a `True` or `False` value. We use it a lot to represent the idea of subsets, where an element is in a subset if the predicate applied to that element returns True and not if it returns False. We define the predicate on a type X as the type of all functions from the given type X to `Prop`.

```
def Pred (X : Type) : Type :=
  X → Prop
```

3.3 Proving Results

Now comes for the hard bit: proving the results we state.

3.3.1 Goals and hypotheses

When we are writing a proof, we are given a *goal*, the thing we are aiming to prove, and a set of *hypotheses*, which we can use in our attempt to prove the goal. For example, we have the following theorem and context, where `Prop` is the type whose elements are either `True` or `False`.

```
theorem orRight : ∀ (a : Prop), a ∨ True := ...
```

```
⊢ ∀ (a : Prop), a ∨ True
```

On the left of the turnstile ($\vdash$), are the hypotheses, of which we currently have none. On the right of the turnstile is the current goal, which at present is the statement of our theorem. How do we move on from here?

3.3.2 Tactics

Lean has this interesting feature called "tactics", which takes a step towards natural language proof and thus helps to make proving theorems simpler and more clear. Each line (or step) of a proof we use a different keyword to describe what we want to do at that point.

Tactic	What the tactic does
`intro`	takes assumptions and makes them a hypothesis
`apply`	takes a 'hypothesis' and applies it to the goal
`simp`	attempts to simplify hypothesis/goal
`have`	allows sub proof for new hypothesis
`exact`	states that the goal is exactly the given thing
`match`	allows case by case analysis for inductive types

We start with the keyword "by" which allows us to use tactics. Then we want to work with some specific value of `a`, to do this we use `intro`. This gives us the following.

```
theorem orRight : ∀ (a : Prop), a ∨ True := by
    intro a
    ...

a : Prop
⊢ a ∨ True
```

Notice now have a slightly different goal: intro has taken `a` from being in the for all section and instead made it a hypothesis. Our goal is now to prove `a ∨ True`. Here comes the slightly tricky part - we have to use some of Lean's built in functions to prove our goal. What we need to say here is that we can prove that the goal is always true because the right hand proposition of the or is always true.

```
theorem orRight : ∀ (a : Prop), a ∨ True := by
    intro a
    apply Or.inr

a : Prop
⊢ True
```

Our goal is now to prove `True` and we can just use the introduction rule for `True`.

```
theorem orRight : ∀ (a : Prop), a ∨ True := by
    intro a
    apply Or.inr
    exact True.intro
```

We now have no context because we have achieved our goal and therefore finished the proof.

4 Formalising Domain Theory

We will now go through some of the Lean code I have written, so that you can get more familiar with the language and look through the rest of the codebase independently. First we define a relation. Recall that a relation on X is a subset of the set $X \times X$ and that we model a subset using a predicate. Therefore we can define a relation as a function which takes two arguments and returns a prop as follows.

```
def Relation (X : Type) : Type :=
  X → X → Prop
```

Now that we have relations defined, we can define a poset, where D is the underlying set and `rel` is the relation on this set. We take these two and check that they follow the three required rules as given in Definition 2.2.

```
structure Poset (D : Type) (rel : Relation D) where
  reflexive : ∀ x : D, rel x x
  transitive : ∀ x y z : D, rel x y → rel y z → rel x z
  antisymmetric : ∀ x y : D, rel x y → rel y x → x = y
```

Now let's look at an example: the two item set from earlier. First, we define the relation on the set as follows:

```
def TSRel : Relation TwoSet
  | ⊤, ⊥ => False
  | _, _ => True
```

We split the cases of the two argument into either $(\top, \bot)$ or some other pair. The underscore here acts as a wildcard and means that any value is accepted. This therefore says that the relation $\top \sqsubseteq \bot$ does not hold, but every other relation does hold.

We can now prove that this indeed forms a poset. At the first step, we make a poset. Lean then creates three different context to prove the three different axioms: reflexivity, transitivity and antisymmetry. We then prove that each of these cases holds individually.

```
theorem TSPoset : Poset TwoSet TSRel := by
  apply Poset.mk

  .case reflexive
    => intro x
       match x with
       | ⊥ => simp [TSRel] -- have ⊥ ⊑ ⊥
       | ⊤ => simp [TSRel] -- have ⊤ ⊑ ⊤

  .case transitive
    => intro x y z lxy lyz
       match x, y with
       | ⊥, _ => simp [TSRel]
       | ⊤, ⊥ => simp [TSRel] at lxy
       | ⊤, ⊤ => exact lyz

  .case antisymmetric
    => intro x y lxy lyx
       match x, y with
       | ⊥, ⊥ => exact rfl
       | ⊤, ⊤ => exact rfl
       | ⊤, ⊥ => simp [TSRel] at lxy -- we derive False
       | ⊥, ⊤ => simp [TSRel] at lyx -- we derive False
```

See the whole codebase at [Wil].

5 Conclusions and Further Work

I have formalised all definitions given in Section 2 and have attempted to formalise the three examples given: namely the two item set, $\mathbb{N}_\infty$ and the interval domain. I have proven that the two item set is a poset and a continuous domain, but proving it was a dcpo required some

classical axioms – the law of excluded middle. This is not a problem, but I think it would be nice to prove this constructively – that is, without classical axioms. I hit a similar barrier when proving that $\mathbb{N}_\infty$ was a dcpo and it turns out that both of these are impossible to prove constructively, at least in the current way they are defined [Jon23]. I also proved that the interval domain was a poset, but decided to leave the dcpo and continuous domain proof as I thought these would be held back at similar points.

For further work, I would like to read more about constructive proofs of the two item set being a dcpo and the natural numbers with infinity also being a dcpo . I would also like to work up to harder definitions and proofs so that I can contribute to the mathlib library, both in terms of reading more about domain theory and in terms of getting more familiar with Lean.

References

[AJ95] Samson Abramsky and Achim Jung. "Domain Theory". In: *Handbook of logic in computer science* (Oct. 1995). DOI: 10.1093/oso/9780198537625.003.0001.

[ES99] Abbas Edalat and Philipp Sünderhauf. "A domain-theoretic approach to computability on the real line". In: *Theoretical Computer Science* 210.1 (1999). Real Numbers and Computers, pp. 73–98. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(98)00097-8. URL: https://www.sciencedirect.com/science/article/pii/S0304397598000978.

[Gie+12] Gerhard Gierz et al. *A Compendium of Continuous Lattices*. Jan. 2012. ISBN: 9783642676802. DOI: 10.1007/978-3-642-67678-9.

[Gou13] Jean Goubault-Larrecq. *Non-Hausdorff Topology and Domain Theory: Selected Topics in Point-Set Topology*. New Mathematical Monographs. Cambridge University Press, 2013.

[Jon23] Tom De Jong. *Domain Theory in Constructive and Predicative Univalent Foundations*. Jan. 2023. DOI: 10.48550/arXiv.2301.12405.

[Sco70] Dana Scott. *OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION*. Tech. rep. PRG02. OUCL, Nov. 1970, p. 30.

[Wil] Anna Williams. https://codeberg.org/awsloth/LeanDomainTheory.