1 Introduction

The idea for this project sparkled from a ILLC taught course[1], which explored a substantial part of Duality Theory, a branch of mathematics whose main research interest, loosly speaking, is to understand how certain mathematicals structures can be transformed in other mathematical structures and represented in terms of objects pertaing to these new structures.

The core of this study area consists in defining translations that allow to jump from one structure to another and viceversa: given a structure of kind A we can translate it to a structure of kind B (its "dual") and translate back the structure of kind B to a structure of kind A which is identical to the one we began with, up to isomorphism. This means for example that it is possible to transfer morphisms between two structures of the same kind to morphisms between structures obtained through the translation. This is particularly helpful because it allows to work around problems in one mathematical field (e.g. Abstract Algebra), by solving them in another one (e.g. Topology).[2]

Our main objective for this project is to concentrate on two particular kinds of structures (Distributive Lattices and Priestley Spaces), and, basing our work upon known results in the field, encode an effective translation between them.

In order to meet this objective, we will have to construct encodings for a various range of mathematical objects, whose definition can become quite involved: we have therefore structured this report in such a way that the reader is spared having to go through all the mathematical details in one single section; we shall instead gradually introduce them, along with their Haskell implementation. This will allow even the casual user to get confortably accostumed to the various notions at play and to quickly understand the reasoning behind their encoding.

The main sections of the report correspond to the main mathematical objects we introduce. The guiding idea is to start from the most simple structures and work our way up to the more complicated ones.

First of all we want to be able to treat functions as objects in our implementation, that is we want to state facts and define properties about them: this is taken care in our first section. In the second section instead we look at Partially ordered sets which will serve as founding blocks and prepare the road for section 3, in which we will properly implement Distributive Lattices and the Lattice Homomorphism. In the fourth section we switch structures: we implement topological spaces[3], Priestley Spaces and the Priestley Space homeomorphism. TO BE CONTINUED SINCE THESE PART HAVE STILL TO BE COMPLETED

By the end we will have a program endowed with a library built to work with numerous different mathematical structures (posets, arbitrary lattices, distributive lattices, arbitrary topological spaces, priestley spaces) in order to study their properies, run `QuickCheck` tests, and to effectively compute the duals of these structures.

Moreover the library of this project can serve as basis for other projects which intend to explore other duality-linked structures (like Boolean Algebras and Stone Spaces), since foundamental concepts like lattices and topologies over sets are already defined, or to apply these concepts to other domains. For example, since we only consider finite cases and since every finite distributive lattice is a Heyting Algebra, the structures here defined can serve as a semantic for intuitionistic

[1] The Mathematical Structures in Logic course was offerend by the ILLC in 2025 at UvA.

[2] For the readers interested in the mathematical background of this project we refer to balbalbalbalababalab

[3] to this end project -name of project- of –, proved to be a valuable source of inspiration.

propositional logic, and therefore this project can serve for such logical applications.

In this picture the use of Haskell as the programming language for this project comes in quite naturally: the fact that it is a "functional" programming language is an obviuos advantage when working

- "functional" nature: the fact that the "first class citizens" in Haskell are functions and that the focus is definition rather then orders, is an obvious advantage when dealing with mathematical objects. It is quite an impressive feature that the ecodings are, for the most part, almost identical to the mathematical definition.
- purity: the fact that there is a strict separation between the pure functions and those that have side effects, allows to work in an environment close to the idelised one of mathematics. Moreover it makes quite more manageble to prove propositions about code, which combined with the fact that the encodings are so similar to proper mathematical definitions, makes programming in this environment closer to standard mathematical practice.
- static type system: it allows to have precisely defined types for different kind of structures. This allows to simulate, within the implementation, a mathematical univers in smaller scale and to treat the objects of one Haskell type precisely as their mathematical counterparts.

Before continuing some practicalities are in order: we shall use the math notation "P" to refer to mathematical structures, and the code notation "`P`" to denote Haskell object of some particular type. In practice we shall often just say "the mathematical structure P of type `a`" when it helps explaining how the structure is encoded and when no confusion arises.