View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Brock University Digital Repository

$\mathcal{L}$-Fuzzy Relations in Coq

Ethan Jackson

Department of Computer Science

Supervisor:
Dr. Michael Winter

Submitted in partial fulfilment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

©Ethan Jackson 2014

BROCK UNIVERSITY

Abstract

Faculty of Graduate Studies
Department of Computer Science

Master of Science

$\mathcal{L}$-Fuzzy Relations in Coq

by Ethan JACKSON

Heyting categories, a variant of Dedekind categories, and Arrow categories provide a convenient framework for expressing and reasoning about fuzzy relations and programs based on those methods. In this thesis we present an implementation of Heyting and arrow categories suitable for reasoning and program execution using Coq, an interactive theorem prover based on Higher-Order Logic (HOL) with dependent types. This implementation can be used to specify and develop correct software based on $\mathcal{L}$-fuzzy relations such as fuzzy controllers. We give an overview of lattices, $\mathcal{L}$-fuzzy relations, category theory and dependent type theory before describing our implementation. In addition, we provide examples of program executions based on our framework.

Contents

Abstract **i**

Contents **ii**

1 Introduction **1**
1.1 Introduction . . . 1

2 Preliminaries **3**
2.1 Classical Relations . . . 3
2.1.1 Example . . . 3
2.1.2 Matrix Representation . . . 4
2.2 Fuzzy Relations . . . 5
2.2.1 Example . . . 5
2.3 Mathematical Constructions for $\mathcal{L}$-Fuzziness . . . 6
2.3.1 Partially-Ordered Sets and Hasse Diagrams . . . 6
2.3.1.1 Upper and Lower Bounds . . . 7
2.3.1.2 Least and Greatest Elements . . . 7
2.3.1.3 Join and Meet . . . 8
2.3.2 Lattices . . . 8
2.3.3 Heyting Algebras . . . 9
2.4 $\mathcal{L}$-Fuzzy Relations . . . 10
2.4.1 Crispness, Scalars and Alpha Cuts . . . 12
2.4.2 Arrow Operations . . . 13
2.4.3 Example . . . 13
2.4.3.1 Meet and Join of Relations . . . 15
2.4.3.2 Alpha Cuts and Arrow Operations . . . 15
2.4.3.3 Converse and Composition . . . 16

3 Categories of Relations **18**
3.1 Category Theory . . . 18
3.1.1 Categories . . . 18
3.2 Heyting, Dedekind and Arrow Categories . . . 21
3.2.1 Heyting and Dedekind Categories . . . 21
3.2.2 Arrow Categories . . . 22

4 Depedent Type Theory **24**
4.1 Dependent Types . . . 24

4.1.1 Dependent Type Structures 24
4.1.2 The Calculus of Constructions 25
4.1.2.1 Type Inhabitance 25
4.2 Russel's and Girard's Paradoxes 26
4.3 Predicative Types and Universe Variables 27

5 The Coq Proof Assistant 29
5.1 Set, Prop and Type 29
5.2 Proofs and Tactics 30
5.3 Structures 34
5.4 Functions 35
5.4.1 Fixpoint 36
5.4.2 Lambda Expressions 36
5.5 Infix Operators, 'Notation' 37
5.6 Prop vs. bool 37

6 $\mathcal{L}$-Fuzzy Relations in Coq 39
6.1 Types 39
6.1.1 Decidable Types 39
6.1.2 Finite Decidable Non-Empty Types 41
6.2 Heyting Algebras 41
6.2.1 Operations 41
6.2.2 Axioms 42
6.3 $\mathcal{L}$-Powersets 43
6.3.1 Operations 44
6.3.2 Properties 44
6.4 $\mathcal{L}$-Fuzzy Relations 45
6.4.1 Operations 45
6.4.2 Properties 47
6.4.2.1 Category Properties 48
6.4.2.2 Dedekind Properties 48
6.4.2.3 Arrow Properties 49

7 Arrow Categories in Coq 50
7.1 Categories 50
7.1.1 `Category` 50
7.1.2 Category `LREL` 51
7.2 Heyting Categories 51
7.2.1 `Heyting Category` 52
7.2.2 HeytingCategory `HLREL` 53
7.3 Arrow Categories 53
7.3.1 `ArrowCategory` 53
7.3.2 Arrow Category `ALREL` 54

8 Computing with `LRels` 55
8.1 Heyting Algebra 55
8.2 Source and Target Types 57
8.3 Relations and Operations 57

9 Conclusion and Future Work **60**

10 Appendix **61**

Bibliography **62**

Chapter 1

Introduction

1.1 Introduction

The purpose of this thesis is to construct and discuss a framework for specifying and implementing certifiably-correct software based on $\mathcal{L}$-fuzzy relations in the context of Dedekind and Arrow categories. The motivation for this work is to have a proof-of-concept that algebraically-described programs based on $\mathcal{L}$-fuzzy relations can be straightforward to implement and reason about using proof assistants based on higher-order logic (HOL) and functional programming featuring dependent types.

There are many applications of relational methods in computer science for which correctness is an important consideration. To this end, we sought to construct a framework for $\mathcal{L}$-fuzzy relations in the category theoretical context using a proof assistant. This framework can be used to specify and correctly implement programs based on $\mathcal{L}$-fuzzy relations, such as fuzzy controllers.

Aside from the practical applications, it is interesting to discuss the relationship between abstract mathematics, type theory, and programming languages. We will discuss the theoretical basis for many of the choices taken during the planning and development of this work, especially those which have an impact on logical consistency or affect the ease of completing proofs.

It is important to realize that program verification is generally difficult, and that standard general-purpose programming languages are not well-suited for reasoning. In fact, standard programming paradigms usually do not support reasoning. Functional programming together with HOL offer an alternative paradigm which combines programming with reasoning. Many proof assistants based on this feature natural deduction style reasoning, which is familiar to many mathematicians and computer scientists.

In order to implement programs based on mathematical constructions, a suitable environment should be chosen for programming and proving. These are usually separate tasks, but in order to define types which are faithful to their mathematical definitions and use them in programs, they should be done together. To this end, our work uses Coq - a French-developed proof assistant implementing a functional programming language and tactics-based theorem proving [8]. The key advantage Coq affords over other systems is that programming and proving are possible using the same language while avoiding logical inconsistencies in the type theory [7].

Prior to discussing our implementation, an overview of mathematical preliminaries is given in Chapter 2 before recalling the definitions of Heyting, Dedekind and Arrow categories, the abstract framework for $\mathcal{L}$-fuzzy relations as described in [31], in Chapter 3. We then discuss in Chapters 4 and 5 important theoretical considerations for selecting Coq and detail the subset of features of Coq which are relevant to our implementation. A discussion of the implementation follows in Chapters 6 and 7 with details about types, functions and selected proofs. The implementation itself is included as a digital appendix. We give some examples of programming with $\mathcal{L}$-fuzzy relations in Chapter 8 and give concluding remarks and an outline of future work in Chapter 9.

Chapter 2

Preliminaries

In this section we give an overview of classical relations, fuzzy relations and $\mathcal{L}$-fuzzy relations in particular.

2.1 Classical Relations

Classical relations are defined as sets of ordered pairs. We use the usual notation for sets, pairs and set operators. Given a source set A and a target set B, a classical relation R is a subset of $A \times B$, denoted $R \subseteq A \times B$. The inclusion of a pair $(a, b) \in R$ denotes that $a \in A$ is in relation to $b \in B$ via R. For convenience, we denote a relation R with source A and target B as $R : A \to B$.

Another representation of classical relations uses the characteristic function, which maps pairs of elements from A and B to a Boolean value. In other terms, a classical relation R with source A and target B is a binary function $R : A \times B \to \mathbb{B}$.

2.1.1 Example

Let us consider a relation representing the popularity of various sports in the set of continents. We have a source set $SPRT$ = {*curling, hockey, soccer, tennis*} and a target set $CONT$ = {*Africa, Antarctica, Asia, Australia, Europe, North America, South America*}. A classical relation $S_1 : SPRT \to CONT$ then denotes only whether a sport is popular on a given continent:

S_1 = {*(curling, North America), (curling, Europe), (hockey, North America), (hockey, Europe), (soccer, Africa), (soccer, Asia), (soccer, Europe), (soccer, South America), (tennis, Australia), (tennis, Europe)*}

The above representation enumerates only the pairs which are related. Other formats exist to visualize relations, such as directed graphs or matrices.

2.1.2 Matrix Representation

Matrix representations for relations provide a convenient way to quickly visualize relations. We follow the format detailed in [26], where matrix rows and columns correspond to the enumerated elements of the source and target sets, respectively, and the matrix entries are either 0 or 1 for denoting exclusion and inclusion in the relation, respectively.

For example, the relation S_1 can be visualized by:

$$S_1 = \begin{array}{c} \\ \mathit{curling} \\ \mathit{hockey} \\ \mathit{soccer} \\ \mathit{tennis} \end{array} \begin{array}{c} \begin{array}{ccccccc} \mathit{Afr.} & \mathit{Ant.} & \mathit{Asia} & \mathit{Aus.} & \mathit{Eur.} & \mathit{N.A.} & \mathit{S.A.} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

When an appropriate order is available and obvious from the context, we often drop the labels from the matrix representation. S_1 can be visualized by:

$$S_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

where the elements from $SPRT$ and $CONT$ are ordered alphabetically from top to bottom and left to right, respectively.

This representation is convenient for visualization and computation, but only when the source and target sets are finite, reasonably small and are enumerable in a convenient way. Contrary to the subset-of-pairs representation, we have to display a value for each pair in $SPRT \times CONT$ to represent inclusion or exclusion, which is somewhat clumsy when inclusion in the relation is sparse. But for many of the relations and operations we will discuss, this representation works well.

2.2 Fuzzy Relations

Fuzzy relations are based on fuzzy sets, introduced by Zadeh in [37]. Fuzzy relations, like fuzzy sets, attribute a membership value to each element, or to each pair in the relation. As mentioned earlier, a classical relation $R : A \to B$ can be viewed as a function $A \times B \to \mathbb{B}$. This notion is extended to use fuzzy values for membership, rather than being limited to inclusion or exclusion.

Fuzzy membership values are usually taken from the unit interval, i.e. $[0 \ldots 1] = \{x \in \mathbb{R} | 0 \leq x \leq 1\}$, a totally-ordered subset of the real numbers. We distinguish fuzzy relations which use the unit interval for membership degrees as real-valued fuzzy relations. A real-valued fuzzy relation $S : A \to B$ can be defined as a function $A \times B \to [0 \ldots 1]$.

Classical relations are embedded in fuzzy relations when the membership function evaluates to only 0 or 1, representing false and true respectively. These are called crisp relations in the fuzzy world.

2.2.1 Example

Popularity illustrates a type of relation which is difficult to define classically. The notion of popularity is probably not accurately representable using absolute inclusion or exclusion. Its meaning depends on context or interpretation. Such a notion, as it stands, is imprecise.

We can express a more accurate notion of sports popularity using fuzzy relations. As mentioned earlier, a relation describing the popularity of a sport is an imprecise notion. By using a fuzzy relation, we can express a richer notion of popularity without any additional context. For example we can define a real-valued (unit) fuzzy relation $S_2 : SPRT \to CONT$

$$S_2 = \begin{array}{c} \\ \textit{curling} \\ \textit{hockey} \\ \textit{soccer} \\ \textit{tennis} \end{array} \begin{array}{c} \begin{array}{ccccccc} \textit{Afr.} & \textit{Ant.} & \textit{Asia} & \textit{Aus.} & \textit{Eur.} & \textit{N.A.} & \textit{S.A.} \end{array} \\ \begin{pmatrix} 0.01 & 0 & 0.2 & 0.03 & 0.85 & 0.9 & 0.02 \\ 0.03 & 0 & 0.4 & 0.05 & 0.9 & 0.95 & 0.04 \\ 0.98 & 0 & 0.9 & 0.4 & 0.95 & 0.2 & 0.99 \\ 0.4 & 0 & 0.3 & 0.8 & 0.85 & 0.3 & 0.2 \end{pmatrix} \end{array}$$

We could infer from S_2 that higher membership degrees in the relation imply a higher degree of popularity. Unlike the classical relation S_1 from Section 2.1.1, notions of partial and relative popularity can now be inferred. For example, we can deduce from S_2 that

hockey is more popular in North America than it is in Europe, or that soccer is not *completely* unpopular in North America.

However the semantics of membership degrees is not clear without further explanation. If the relation S_2, for example, is just meant to describe the relative popularity of sports, then we could arbitrarily choose sufficient membership values from a very narrow range of the unit interval to describe this notion. Why is soccer popular in Europe with degree 0.95 and not 0.94 or 0.96? When membership values are taken from the unit interval there isn't necessarily any associated meaning other than the ordering of the reals.

2.3 Mathematical Constructions for $\mathcal{L}$-Fuzziness

In our work we move to $\mathcal{L}$-fuzzy relations, where membership degrees are selected from a lattice of values. This allows us to introduce membership values which are partially ordered. This means that relations can be described using membership degrees which relate elements under different kinds of criteria. By using a lattice, we can introduce membership values which are based on arbitrary phrases of everyday language to give them abstract semantic meaning. As we will see, such relations are much more expressive and can describe situations in greater detail.

Before further discussing of $\mathcal{L}$-fuzzy relations, we recall definitions for partially-ordered sets, lattices and in particular complete Heyting algebras, with important operations and properties.

2.3.1 Partially-Ordered Sets and Hasse Diagrams

Partially-ordered sets (posets) provide the ordering of elements which we will later use to define lattices of membership values, as well as lattices of relations.

Definition 2.1. $(A, \leq)$ is a poset if and only if for all x, y and z in A,

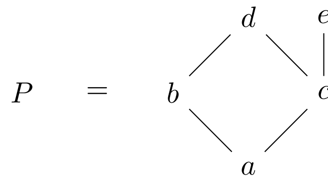
$$\begin{aligned} & x \leq x && (reflexive) \\ & \text{if } x \leq y \text{ and } y \leq x \text{ then } x = y && (antisymmetric) \\ & \text{if } x \leq y \text{ and } y \leq z \text{ then } x \leq z && (transitive) \end{aligned}$$

We can visualize posets using Hasse diagrams. Though there is a formal definition for Hasse diagrams, we only use them informally to visualise the order relation. Consider the following example of a poset.

Take $P = (A, \leq)$ to be a poset over a 5-element set $A = \{a, b, c, d, e\}$, where the order relation $\leq$ is given by

$$P = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

The poset P can be visualised by:



We use this example to help define important operations.

2.3.1.1 Upper and Lower Bounds

For a subset S of elements of A, the upper bounds are those elements $a \in A$ such that for all $x \in S$, $x \leq a$. For example the upper bounds of $\{a\}$ are $\{a, b, c, d, e\}$ while the upper bound of $\{b, c\}$ is just $\{d\}$.

Lower bounds are defined dually with respect to the poset ordering. For example, the lower bounds of $\{d\}$ are $\{a, b, c\}$ and the lower bound of $\{b, c\}$ is just $\{a\}$.

2.3.1.2 Least and Greatest Elements

Next, we consider the least and greatest elements of subsets. Given a subset S of elements from A, the least element of S is a single $s' \in S$ such that for all $x \in S$, $s' \leq x$. In other words, s' is a lower bound of S that belongs to S. For example the least element of the subset $\{c, d, e\}$ is c, and the least element of the subset $\{b, c\}$ does not exist. The least element of the poset P is a.

Greatest elements are also defined dually. For example, the greatest element of $\{b, c, d\}$ is d, and the greatest element of the poset P does not exist.

2.3.1.3 Join and Meet

We also have the join or least upper bound (LUB) and the meet or greatest lower bound (GLB) of subsets of a poset. The join of a subset S of elements from A, if it exists, is a single element from A which is the least element of the upper bounds of S. For example the join of $\{b, c\}$ is d, and the join of the empty subset is the least element of the poset. If the greatest element of a poset exists, then it is the least upper bound of the entire poset.

The meet is defined dually. For example, the meet of $\{a, b, c\}$ is a, and the meet of $\{b, e\}$ is also a. We often need to compute the meet or join for pairs of elements rather than arbitrary subsets. If we are only considering two elements, the meet z of two elements x and y in A, denoted by $x \sqcap y$, is given by:

$$\begin{aligned} &z \leq x \text{ and } z \leq y && \textit{(z is a lower bound)} \\ &\text{for all } w \in A, \text{ if } w \leq x \text{ and } w \leq y \text{ then } w \leq z && \textit{(z is the greatest lower bound).} \end{aligned}$$

Similarly the join of two elements x and y in A, denoted by $x \sqcup y$, is defined dually with respect to the ordering.

2.3.2 Lattices

Lattices can be defined equivalently in order theory or universal algebra.

In order theory, a lattice is defined as a poset $(L, \leq)$ for which each pair of elements in L has a join and a meet.

The algebraic definition of a lattice is as follows.

Definition 2.2. A lattice $\mathcal{L}$ is a triple $(L, \sqcup, \sqcap)$ where L is a set and $\sqcup$ and $\sqcap$ denote binary join and meet operations such that

1. $x \sqcap y = y \sqcap x$ $\quad x \sqcup y = y \sqcup x$ $\quad$ (commutativity)
2. $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ $\quad (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ $\quad$ (associativity)
3. $x \sqcap (x \sqcup y) = x$ $\quad x \sqcup (x \sqcap y) = x$ $\quad$ (absorption)

Again, it is important to note that the two definitions are equivalent[6]. This allows one to use whichever definition is most convenient in an interchangeable fashion. The order relation can be described from the algebraic definition by for all $x, y \in L$, $x \leq y \iff x \sqcap y = x$.

In order to capture the notion of crispness in fuzzy relations, we need to enforce that the lattice of membership values we use always has a least and a greatest element. This is given by a bounded lattice.

Definition 2.3. A bounded lattice $(\mathcal{L}, 0, 1)$ is a lattice $\mathcal{L}$ where 0 and 1 respectively are least and greatest elements such that for all $x \in L$, $x \sqcup 0 = x$ and $x \sqcap 1 = x$.

For some further constructions, we need to define distributive lattices.

Definition 2.4. A lattice $\mathcal{L}$ is distributive iff its meet and join operations distribute over each other such that for all $x, y, z \in L$, $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.

A distributive lattice can also be axiomatized by the dual distributive property, i.e. where the join operation distributes over the meet operation. Both definitions are equivalent as one property follows from the other, as we have implemented in our framework.

2.3.3 Heyting Algebras

In this section we focus on a special class of lattices called Heyting algebras.

A Heyting algebra is a bounded lattice with an additional binary implication operation denoted by $\rightarrow$. The implication operation provides a means to describe a weaker form of complementation called the relative pseudo-complement. The formal definition of a Heyting algebra is as follows.

Definition 2.5. A Heyting algebra $\mathcal{H} = (\mathcal{L}, \rightarrow)$ is a bounded lattice $\mathcal{L}$ together with an implication operation $\rightarrow$ such that

1. $x \rightarrow x = 1$
2. $x \sqcap (x \rightarrow y) = x \sqcap y$
3. $y \sqcap (x \rightarrow y) = y$
4. $x \sqcap (y \rightarrow z) = (x \rightarrow y) \sqcap (x \rightarrow z)$

In [31], the implication operation of a Heyting algebra is equivalently axiomatized by for all x, y and z in $\mathcal{L}$,

$$z \leq x \rightarrow y \iff x \sqcap z \leq y$$

As lattices, Heyting algebras are always distributive. In the case of of a complete Heyting algebra, which is a Heyting algebra and complete as a lattice the first infinite distributive law holds, which states that the binary meet distributes over the subset join operation

for any subset. Precisely, given a complete Heyting algebra H, for any element $x \in H$ and for any subset $Y \subseteq H$,

$$x \sqcap \bigsqcup Y = \bigsqcup \{x \sqcap y | y \in Y\}.$$

where $\bigsqcup$ denotes the general LUB operation, as opposed to a binary join.

Though we do not use complete Heyting algebras in the implementation, they are an important component of Dedekind categories - the theoretical basis for the work discussed later.

A simple example of a complete Heyting algebra characterizes a three-valued lattice of possible membership degrees for $\mathcal{L}$-fuzzy relations. This represents a lattice where elements are completely related, completely unrelated or are related by an intermediate degree. We can visualize this as follows:

$$\mathcal{L} \quad := \quad \begin{array}{c} 1 \\ | \\ m \\ | \\ 0 \end{array}$$

The use of a diagram makes the ordering easy to see. It is easy to verify that $\mathcal{L}$ forms a complete Heyting algebra. It is easy to see that $\mathcal{L}$ is a complete lattice and we can define a relative pseudo-complement as a function $(\mathcal{L} \times \mathcal{L}) \to \mathcal{L}$:

$$x \to y := \begin{array}{c} \\ {\scriptstyle 1} \\ {\scriptstyle m} \\ {\scriptstyle 0} \end{array} \begin{array}{c} \begin{array}{ccc} {\scriptstyle 1} & {\scriptstyle m} & {\scriptstyle 0} \end{array} \\ \begin{pmatrix} 1 & m & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

In the implementation we will have to prove that $\mathcal{L}$ is complete and that $\to$ is indeed a relative pseudo-complement in order to build a complete Heyting algebra.

2.4 $\mathcal{L}$-Fuzzy Relations

We use the same notation for $\mathcal{L}$-fuzzy relations as for classical relations. For a complete Heyting algebra $\mathcal{L}$, an $\mathcal{L}$-fuzzy relation between two sets A and B is a function from $A \times B$ to $\mathcal{L}$. Such a relation R is denoted $R : A \to B$. Again, classical relations can be

described as $\mathcal{L}$-fuzzy relations where $\mathcal{L} = \mathbb{B}$, and all of what follows applies equally to classical relations in that setting.

In the following definitions we define important operations on $\mathcal{L}$-fuzzy relations, relational inclusion, the identity relation and the least and greatest relations.

Definition 2.6. Given $\mathcal{L}$-fuzzy relations $Q, R : A \to B$ and $S : B \to C$

$$\begin{aligned}
(Q \sqcap R)(x, y) &:= Q(x, y) \sqcap R(x, y), && (meet)\\
(Q \sqcup R)(x, y) &:= Q(x, y) \sqcup R(x, y), && (join)\\
Q^{\smile}(x, y) &:= Q(y, x), && (converse)\\
(Q; S)(x, z) &:= \bigsqcup_{y \in B} (Q(x, y) \sqcap S(y, z)) && (composition)
\end{aligned}$$

Definition 2.7. For $Q, R : A \to B$, the relational inclusion operator, denoted by $\sqsubseteq$ is defined by

$$Q \sqsubseteq R \iff \forall x \in A, y \in B : Q(x, y) \leq R(x, y)$$

Definition 2.8. For any set A, the identity relation on A is defined by

$$\mathbb{I}_A(x, y) := \begin{cases} 1 & \text{iff } x = y \\ 0 & \text{otherwise.} \end{cases}$$

Definition 2.9. For all source and target sets A and B, the least and greatest relations are respectively defined by

$$\perp\!\!\!\perp_{AB}(x, y) := 0 \qquad \top\!\!\!\top_{AB}(x, y) := 1$$

Later we will use the fact that the set of all $\mathcal{L}$-fuzzy relations between sets A and B is itself a complete Heyting algebra [31], with least and greatest elements defined as follows.

In order to define further constructions, we note some of the properties of these operations.

Lemma 2.10. *For $Q, Q_2 : A \to B$, $R : B \to C$, $S : C \to D$ and $T : A \to C$ we have*

$$Q;\mathbb{I}_B = Q \text{ and } \mathbb{I}_B;R = R \qquad \textit{(identity laws)}$$

$$Q;(R;S) = (Q;R);S \qquad \textit{(associativity of composition)}$$

$$(Q \sqcap Q_2)^{\smile} = Q^{\smile} \sqcap Q_2^{\smile}$$

$$(Q;R)^{\smile} = R^{\smile};Q^{\smile}$$

$$(Q^{\smile})^{\smile} = Q \qquad \textit{(converse laws)}$$

$$Q;R \sqcap T \sqsubseteq Q;(R \sqcap Q^{\smile};T) \qquad \textit{(modular law)}$$

$$Q;\perp\!\!\!\perp_{BC} = \perp\!\!\!\perp_{AC} \qquad \textit{(composition with least relation)}$$

Proofs of these properties can be found in [31], and we discuss them again in the implementation section. In the remainder of this section we discuss useful operations on relations and continue with our example.

2.4.1 Crispness, Scalars and Alpha Cuts

In many applications of fuzzy relations, we need to have a notion of crispness. Crisp relations are those which only use membership degrees of 0 or 1 - the least and greatest elements of the lattice in the case of $\mathcal{L}$-fuzzy relations. In other terms, a relation R is crisp iff for all x and y, $R(x, y) = 0$ or $R(x, y) = 1$.

Scalar relations form another important class. These are relations which are used to abstractly identify classes of $\mathcal{L}$-fuzzy relations with the lattice $\mathcal{L}$. Scalar relations, introduced by Furusawa and Kawahara [20], are equivalent to the notion of ideals introduced by Jónsson and Tarski [17], which are relations $R : A \to B$ satisfying $\top\!\!\!\top_{AA};R;\top\!\!\!\top_{BB} = R$.

Definition 2.11. A relation $\alpha : A \to A$ is called a scalar on A iff $\alpha \sqsubseteq \mathbb{I}_A$ and $\top\!\!\!\top_{AA};\alpha = \alpha;\top\!\!\!\top_{AA}$.

Scalar relations are partial identity relations. For example if A is a 3-element set, the relation

$$\alpha : A \to A = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$$

where a is any element from $\mathcal{L}$ is a scalar on A.

An alpha cut is an operation on an $\mathcal{L}$-fuzzy relation which produces a crisp relation based on a threshold element from $\mathcal{L}$.

Definition 2.12. Given a relation R and an element $\alpha \in \mathcal{L}$, an alpha-cut R_α is defined by

$$R_\alpha = \begin{cases} 1 & \text{iff } R(x, y) \geq \alpha \\ 0 & \text{otherwise.} \end{cases}$$

As we can see, this operation generates a crisp relation where pairs which were related by a degree of at least a in the original relation are set to 1. Those which were related to a degree less than a are set to 0. Again, this can be thought of as a threshold or a high-pass operation. The alpha cut can also be defined in terms of scalars and other operations which are introduced later.

2.4.2 Arrow Operations

Arrow operations provide additional functionality for working with crisp relations in the fuzzy world. We use two such operations.

Given an $\mathcal{L}$-fuzzy relation R we define two unary operations $\uparrow$ and $\downarrow$ which when applied to R produce crisp relations. $R^\uparrow$ maps R to the least crisp relation that contains R, and $R^\downarrow$ maps R to the greatest crisp relation contained in R. Such relations are sometimes called the support and kernel of R, respectively.

Definition 2.13. Given an $\mathcal{L}$-fuzzy relation R we define the arrow operations by

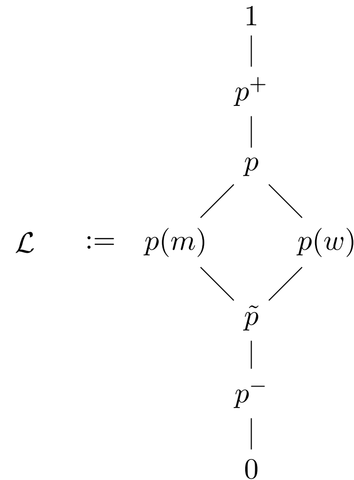
$$R^\uparrow(x, y) = \begin{cases} 0 & \text{iff } R(x, y) = 0 \\ 1 & \text{otherwise} \end{cases} \qquad R^\downarrow(x, y) = \begin{cases} 1 & \text{iff } R(x, y) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

In other words, $R^\uparrow$ lifts any membership degrees which are greater than 0 in R to 1. Conversely, $R^\downarrow$ cuts membership degrees which are less than 1 in R to 0. The $\downarrow$ operation is equivalent to a 1-cut or R_1.

Using these operations, we can assert that a relation R is crisp iff $R^\uparrow = R$ and, equivalently, $R^\downarrow = R$.

2.4.3 Example

We continue our example on popularity of sports moving to an $\mathcal{L}$-fuzzy relation. Rather than using the unit interval for membership degrees, we use the following lattice.



Recall that the relations S_1 and S_2 are meant to describe the popularity of sports on certain continents. By using the lattice $\mathcal{L}$ for membership degrees, the semantics of the relation is clearer. Here we use 0, p^-, $\tilde{p}$, $p(m)$, $p(w)$, p, p^+ and 1 to respectively denote "totally unpopular", "mostly unpopular", "somewhat popular", "popular with men", "popular with women", "popular", "very popular", and "totally popular".

We can now define an $\mathcal{L}$-fuzzy relation $S_3 : SPRT \to CONT$

$$S_3 := \begin{array}{c} \\ \textit{curling} \\ \textit{hockey} \\ \textit{soccer} \\ \textit{tennis} \end{array} \begin{array}{c} \begin{array}{ccccccc} \textit{Afr.} & \textit{Ant.} & \textit{Asia} & \textit{Aus.} & \textit{Eur.} & \textit{N.A.} & \textit{S.A.} \end{array} \\ \begin{pmatrix} p^- & 0 & \tilde{p} & p^- & p(m) & p^+ & p^- \\ p^- & 0 & \tilde{p} & 0 & p & p^+ & p^- \\ p^+ & 0 & p & \tilde{p} & p^+ & p(w) & 1 \\ p(w) & 0 & \tilde{p} & p & p & p(m) & \tilde{p} \end{pmatrix} \end{array}$$

The relation S_3 combined with the lattice $\mathcal{L}$ is both richer in information and simpler to interpret. Unlike in the previous example, there is no need to select arbitrary elements from the unit interval for membership degrees. The lattice can be refined or extended as needed and provides a means for indicating membership degrees based on separate, unordered criteria, such as the notion of popularity based on gender.

Next we show examples of important operations on relations. For some examples, we use another relation $S_4 : SPRT \to CONT$ to provide an alternative description of the

popularity of sports on each continent.

$$S_4 := \begin{array}{c} \\ \textit{curling} \\ \textit{hockey} \\ \textit{soccer} \\ \textit{tennis} \end{array} \begin{array}{c} \begin{array}{ccccccc} \textit{Afr.} & \textit{Ant.} & \textit{Asia} & \textit{Aus.} & \textit{Eur.} & \textit{N.A.} & \textit{S.A.} \end{array} \\ \begin{pmatrix} p^- & 0 & p^- & p^- & p^+ & p^+ & p^- \\ p^- & 0 & p^- & p^- & p^+ & p^+ & p^- \\ p^+ & 0 & p^+ & p^+ & p^+ & p^- & p^+ \\ p^- & 0 & p^- & p^- & p^+ & p^- & p^+ \end{pmatrix} \end{array}$$

Note that after the initial definition of a relation, the row and column labels are omitted, but the order of the source and target elements is maintained.

2.4.3.1 Meet and Join of Relations

The meet and join of two relations is defined component-wise using the corresponding binary operations on lattice elements discussed earlier.

$$S_3 \sqcap S_4 = \begin{pmatrix} p^- & 0 & p^- & p^- & p(m) & p^+ & p^- \\ p^- & 0 & p^- & 0 & p^+ & p^+ & p^- \\ p^+ & 0 & p & \tilde{p} & p^+ & p^- & p^+ \\ p^- & 0 & p^- & p & p & p^- & \tilde{p} \end{pmatrix}$$

The meet of two relations results in a new relation where pairs are related by degrees smaller than or equal to those in both the originals, i.e. $S_3 \sqcap S_4 \sqsubseteq S_3$ and $S_3 \sqcap S_4 \sqsubseteq S_4$.

Conversely, the join of two relations results in pairs which are related by degrees greater than or equal than originally, i.e. $S_3 \sqsubseteq S_3 \sqcup S_4$ and $S_4 \sqsubseteq S_3 \sqcup S_4$.

$$S_3 \sqcup S_4 = \begin{pmatrix} p^- & 0 & \tilde{p} & p^- & p^+ & p^+ & p^- \\ p^- & 0 & \tilde{p} & p^- & p^+ & p^+ & p^- \\ p^+ & 0 & p^+ & p^+ & p^+ & p^+ & 1 \\ p(w) & 0 & \tilde{p} & p^+ & p^+ & p^+ & p^+ \end{pmatrix}$$

2.4.3.2 Alpha Cuts and Arrow Operations

An alpha cut S_{3p}, for example, is a crisp relation identifying pairs which were in the relation S_3 to a degree of at least p. These pairs are lifted to degree 1, while the others

are cut down to 0.

$$S_{3p} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Arrow operations also produce crisp relations. For example, the relation $S_3^{\downarrow}$ cuts all values in S_3 smaller than 1 to 0,

$$S_3^{\downarrow} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and the relation $S_4^{\uparrow}$ lifts all values in S_4 greater than 0 to 1.

$$S_4^{\uparrow} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

2.4.3.3 Converse and Composition

Let us consider an example combining converse and composition. Suppose that in addition to S_4, we have a set of music genres, $MUS = \{country, electronic, metal, pop, rock\}$, and a relation $M_1 : MUS \to CONT$,

$$M_1 := \begin{array}{c} \\ ctry. \\ elec. \\ metal \\ pop \\ rock \end{array} \begin{array}{c} \begin{array}{ccccccc} Afr. & Ant. & Asia & Aus. & Eur. & N.A. & S.A. \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & p^- & p^- & p^+ & 0 \\ p^- & 0 & p & \tilde{p} & p^+ & \tilde{p} & p^- \\ p^- & 0 & p^- & p(m) & p^+ & p(m) & p^+ \\ p & 0 & p & p & p & p & p \\ \tilde{p} & 0 & \tilde{p} & p^+ & p^+ & p^+ & p^+ \end{pmatrix} \end{array}.$$

Now suppose we wish to relate the popularity of sports to the popularity of music. Using the converse and composition operations, we can compute a new relation to tell

us to what degree pairs of sports and music genres are most commonly popular on some continent.

We have so far $S_4 : SPRT \to CONT$ and $M_1 : MUS \to CONT$, which cannot be composed directly. Recall that composition requires that the target of the left operand is the same as the source of the right operand.

Instead we can compose S_4 with $M_1^{\smile} : CONT \to MUS$,

$$S_4; M_1^{\smile} = \begin{pmatrix} p^- & 0 & p^- & p^- & p^+ & p^+ & p^- \\ p^- & 0 & p^- & p^- & p^+ & p^+ & p^- \\ p^+ & 0 & p^+ & p^+ & p^+ & p^- & p^+ \\ p^- & 0 & p^- & p^- & p^+ & p^- & p^+ \end{pmatrix}; \begin{pmatrix} 0 & p^- & p^- & p & \tilde{p} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & p & p^- & p & \tilde{p} \\ p^- & \tilde{p} & p(m) & p & p^+ \\ p^- & p^+ & p^+ & p & p^+ \\ p^+ & \tilde{p} & p(m) & p & p^+ \\ 0 & p^- & p^+ & p & p^+ \end{pmatrix}$$

Notice here that the matrix representation of $M_1^{\smile}$ is the transpose of M_1. To compute the result of this composition, we can take even more intuition from matrix operations.

$S_4; M_1^{\smile}(x, z)$ has to be computed by taking the join of pairwise meets $(x, y) \sqcap (y, z)$ for all $y \in CONT$. We can compute the composition the same way we perform matrix multiplication in linear algebra, except that instead of taking the sum of pairwise multiplications, we take the join of pairwise meets. The resulting relation is then given by

$$S_4; M_1^{\smile} = \begin{array}{c} \\ \textit{curling} \\ \textit{hockey} \\ \textit{soccer} \\ \textit{tennis} \end{array} \begin{array}{c} \begin{array}{ccccc} \textit{ctry.} & \textit{elec.} & \textit{metal} & \textit{pop} & \textit{rock} \end{array} \\ \begin{pmatrix} p^+ & p^+ & p^+ & p & p^+ \\ p^+ & p^+ & p^+ & p & p^+ \\ p^- & p^+ & p^+ & p & p^+ \\ p^- & p^+ & p^+ & p & p^+ \end{pmatrix} \end{array}.$$

This relation contains information that was not available earlier. In the context of this example, it tells us that soccer and country music are not very popular on any common continent.

Composition is a very useful operation for computing new relations. It can even be extended to use other operations than meet and join, which is useful in applications, and will be discussed later.

Chapter 3

Categories of Relations

3.1 Category Theory

In this chapter we will discuss a basic algebraic representation for $\mathcal{L}$-fuzzy relations. We want to frame this in the setting of a formal algebraic theory well-suited to typing in a functional programming language. To this end we use category theory.

3.1.1 Categories

A category is a mathematical construction which abstractly describes objects and morphisms between objects. For computer scientists, they are interesting because they provide a formalized language for specifying and reasoning abstractly about data types and programs. We use an enriched category to serve as our basic theory of $\mathcal{L}$-fuzzy relations.

Definition 3.1. A category $\mathcal{C}$ is

1. A collection of objects $Ob_{\mathcal{C}}$,
2. A collection of morphisms $\mathcal{C}[A, B]$, for every pair of objects A and B,
3. An associative, binary composition operation ; which maps morphisms f in $\mathcal{C}[A, B]$ and g in $\mathcal{C}[B, C]$ to a morphism $f; g$ in $\mathcal{C}[A, C]$,
4. An identity morphism denoted by $\mathbb{I}_A$ for all objects A. For all f in $\mathcal{C}[A, B]$ and g in $\mathcal{C}[B, A]$ we have that $\mathbb{I}_A; f = f$ and $g; \mathbb{I}_A = g$.

We denote a morphism f from A to B by $f : A \to B$. Notice that we use the same notation here as we do for relations, which is appropriate since in the next section we will introduce relations as the morphisms of a particular category.

Computer scientists and mathematicians should already be familiar with several categories. Take for example the category **Set** of sets and functions, or the category **Vect** of vector spaces and linear transformations. Category theory abstracts these constructions and provides an algebraic framework for reasoning about them.

There are several categories which serve as a framework for relations. The category **Rel** has sets for objects and classical relations for morphisms, while $\mathcal{L}$-**Rel** substitutes the morphisms with $\mathcal{L}$-fuzzy relations. Both of these are shown indeed to be categories by the properties in Lemma 2.3, namely those showing the satisfaction of the identity laws and the associativity of composition. Interestingly, $\mathcal{L}$-**Rel** is too weak to algebraically express the notion of crispness, and so for our work, a stronger category is required which is introduced in the next section.

Furthermore, we mention the class of Cartesian closed categories (CCCs), which provides an interpretation for the simply-typed lambda calculus, the fundamental theory on which Coq and other functional programming languages are based. To define CCCs we need to recall some other definitions from category theory.

Note that we sometimes use commuting diagrams to express properties in category theory, i.e. diagrams where morphisms compose in the direction of the arrows and composed paths must be equal.

Definition 3.2. A terminal object T of a category $\mathcal{C}$ is one such that for all objects X in $\mathcal{C}$, there exists a unique morphism $X \to T$.

In the categories **Set** the terminal object is any singleton set, in **Rel** it is the empty set.

A product $A \times B$ of objects A and B in $\mathcal{C}$ is the categorical generalization for the Cartesian product of sets, and for other constructions. Products are equipped with two projection morphisms which abstractly identify its constituent objects.

Definition 3.3. A product of objects A and B in $\mathcal{C}$ is an object $A \times B$ together with projection morphisms $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that for any $f_1 \in \mathcal{C}[C, A]$ and $f_2 \in \mathcal{C}[C, B]$, there exists a unique $F \in \mathcal{C}[C, A \times B]$ for which the following commutes:

$$\begin{array}{ccccc} & & C & & \\ & \swarrow^{f_1} & \downarrow F & \searrow^{f_2} & \\ A & \xleftarrow[\pi_1]{} & A \times B & \xrightarrow[\pi_2]{} & B \end{array}$$

e.g. In **Set** the product $A \times B$ of objects A and B is simply the Cartesian product of A and B.

As a theory of functional programming, the product alone is insufficient to express higher-order functions. Instead we need an object that represents sets of morphisms. In set theory, this corresponds to an object representing a function space. To express this in category theory, the notion of exponents was introduced.

Definition 3.4. An exponent of objects A and B in $\mathcal{C}$ is an object B^A along with an evaluation morphism $eval : B^A \times A \to B$ such that for every morphism $f : C \times A \to B$, there exists a unique morphism $h : C \to B^A$ for which the following commutes:

$$\begin{array}{ccccc} C & & C \times A & \xrightarrow{f} & B \\ \downarrow h & & \downarrow h \times \mathbb{I}_A & \nearrow_{eval} & \\ B^A & & B^A \times A & & \end{array}$$

e.g. In **Set** the exponent B^A of objects A and B is the collection of all functions $A \to B$.

Finally, we define the class of Cartesian closed categories as follows:

Definition 3.5. A category $\mathcal{C}$ is Cartesian closed iff

1. $\mathcal{C}$ has a terminal object T,
2. For all objects X and Y in $\mathcal{C}$, there exists a product $X \times Y$ in $\mathcal{C}$,
3. For all objects Y and Z in $\mathcal{C}$, there exists an exponent Z^Y in $\mathcal{C}$.

CCCs are part of a very important isomorphism along with intuitionistic propositional logic and typed combinatory logic known as the Curry-Howard-Lambek isomorphism. This isomorphism provides a sound foundation for proof assistants such as Coq, which is discussed later.

3.2 Heyting, Dedekind and Arrow Categories

In this section we discuss Heyting categories as our basic theory of relations. Our implementation uses a variation of the original theory of Dedekind categories from [31].

3.2.1 Heyting and Dedekind Categories

We use the theory of Dedekind categories [21, 22] as the basic theory of relations throughout this work. These categories are called locally complete division allegories in [10]. As mentioned in the introduction, the motivation for doing this is to establish a framework for abstract algebraic reasoning about $\mathcal{L}$-fuzzy relations. We introduce Heyting categories as an adaptation of Dedekind categories more suitable for working with programming language constructions.

Definition 3.6. A Heyting category $\mathcal{R}$ is a category satisfying the following:

1. For all objects A and B the collection $\mathcal{R}[A, B]$ is a Heyting algebra. Meet, join, the induced ordering, the least and the greatest element are denoted by $\sqcap$, $\sqcup$, $\sqsubseteq$, $\perp\!\!\!\perp_{AB}$ and $\top\!\!\!\top_{AB}$ respectively.
2. There is a monotone converse operation $\smile$ mapping a relation $Q : A \to B$ to $Q^{\smile} : B \to A$ such that for all relations $Q : A \to B$ and $R : B \to C$ we have $(Q; R)^{\smile} = R^{\smile}; Q^{\smile}$ and $(Q^{\smile})^{\smile} = Q$.
3. For all relations $Q : A \to B, R : B \to C$ and $S : A \to C$ the modular law $(Q; R) \sqcap S \sqsubseteq Q; (R \sqcap (Q^{\smile}; S))$ holds.
4. There is a residual operation / such that for all objects A, B and C and for all $R, R' : A \to C$, $S : B \to C$ and $T : A \to B$,
   (a) $(R \sqcap R')/S \sqsubseteq R/S \sqcap R'/S$
   (b) $T \sqsubseteq (T; S)/S$
   (c) $(R/S); R \sqsubseteq R$

A Dedekind category is a Heyting category in which the collection of morphisms must be a complete Heyting algebra.

The axiomatization of the residual operation used here is equivalent to one more commonly used in several other sources ([21],[20], [31], etc.) which states that for all relations $R : B \to C$ and $S : A \to C$ the residual operation / is defined by $S/R : A \to B$ such that for all $X : A \to B$, $X; R \sqsubseteq S \iff X \sqsubseteq S/R$. [11]

Heyting categories extend allegories, which are a more general category of relations, by adding the join operation and the requirement of a complete lattice of morphisms. The category **$\mathcal{L}$-Rel** of $\mathcal{L}$-fuzzy relations indeed forms a Dedekind category, as shown in [31] using the properties of Lemma 2.3.

In Section 2.4 we introduced the notion of scalar relations with examples. In Heyting categories, these are used to abstractly identify the membership values used by relations. This is important for many applications, such as fuzzy controllers, where operations involving partial identities are required.

In our implementation we decided to restrict all lattices to being Heyting algebras - with no completeness requirement. Since we are interested in using this framework to implement programs, we try to use only programming constructions which are easy to work with. In that respect, we use finite lists to model subsets of lattice elements and exclude the completeness property required to form a Dedekind category.

3.2.2 Arrow Categories

In some applications, we require an abstract mechanism for working with crisp relations. This is particularly important in fuzzy controllers, as many of the operations require crisp relations. As mentioned earlier, an $\mathcal{L}$-fuzzy relation is called crisp iff it contains membership degrees of only 0 or 1. It was shown in [31] that the notion of crispness cannot be abstractly expressed using Dedekind categories. To address this, arrow categories were also introduced there.

Arrow categories add two new operations $\uparrow$ and $\downarrow$ which can be understood here just as they were presented in section 2.4. In [31], arrow categories were introduced based on Dedekind categories. For the purpose of this thesis, we allow an arrow category to be based on either a Heyting category or a Dedekind category.

Definition 3.7. An arrow category $\mathcal{A}$ is a Heyting category (or Dedekind category) with $\top\!\!\!\top_{AB} \neq \bot\!\!\!\bot_{AB}$ for all A, B and two operations $\uparrow$ and $\downarrow$ satisfying:

1. $R^\uparrow, R^\downarrow : A \to B$ for all $R : A \to B$
2. $(\uparrow, \downarrow)$ is a Galois correspondence, i.e., $Q^\uparrow \sqsubseteq R$ iff $Q \sqsubseteq R^\downarrow$ for all $Q, R : A \to B$.
3. $(R^\smile; S^\downarrow)^\uparrow = R^{\uparrow\smile}; S^\downarrow$ for all $R : B \to A$ and $S : B \to C$
4. $(Q \sqcap R^\downarrow)^\uparrow = Q^\uparrow \sqcap R^\downarrow$ for all $Q, R : A \to B$
5. If $\alpha_A \neq \bot\!\!\!\bot_{AA}$ is a non-zero scalar then $\alpha_A^\uparrow = \mathbb{I}_A$.

Arrow categories provide a sufficient theory of $\mathcal{L}$-fuzzy relations, providing the algebraic mechanisms we need to specify and reason about their applications in abstract and concrete terms. And because we are interested in using them as a framework for programs, we define an alternative category which again omits the requirement for completeness.

For the remainder of this thesis all references to arrow categories should be understood as arrow categories extending a Heyting category rather than a Dedekind category.

Chapter 4

Depedent Type Theory

4.1 Dependent Types

Dependent types are types dependent on terms. This is an important notion that forms the cornerstone of the type system used by Coq and some other functional programming languages. The best way to introduce dependent types is by example.

4.1.1 Dependent Type Structures

In Coq, like in other programming languages, we can use structures or records to define new types. In most languages, these types are static, meaning that the type of the type itself is immutable. For example, consider the type of matrices. In some cases, we may wish to distinguish between different sizes of matrices at the type level.

In Java for example, there is no convenient way to use a single class to define the type of n by m matrices such that each size constitutes a different type. Instead, all such matrices would have the same type, or we would need to explicitly write other classes to define matrices with specific sizes.

In Coq, we use dependent type structures. Here we briefly explain some basic Coq syntax on the fly before a detailed overview in the next section.

Example 4.1.1. The following code defines the type of n by m Boolean-valued matrices dependent on the values of n and m as a partial map from indices to matrix entry:

```
Structure BMatrix (n m : nat) := {
   matrix_map (i j : nat) : i < n -> j < m -> bool
}.
```

The above can be understood as follows. In the first line, we are telling Coq that we want to define `BMatrices` as a new type defined by the constituent fields of a `Structure`, which is synonymous with `Record`. This type should be dependent on parameters `n` and `m` of type `nat`, which defines the set of natural numbers.

The next line defines the single constituent field of this type, but there could be many which must be separated by `;`. The field `matrix_map` defines the matrix as a partial map from the matrix indices to a Boolean value. It is parametrized on the matrix indices `i` and `j`, which both must be of type `nat`. Then, the type of the field given by `i < n -> j < m -> bool` is a function requiring as input an object of type `i<n` and an object of type `j<m` returning a Boolean value. These function parameters are actually requirements for proofs that the indices are in the range of the matrix. This notion will be discussed at length later.

With this definition we can now use `BMatrix` as a type in programs and proofs. In Section 4.3 we will return to this example and use Coq's type checking system to show where this dependent type fits into the overall type hierarchy.

4.1.2 The Calculus of Constructions

Coq's logic is based on the Calculus of Constructions (CoC), which is deeply integrated into the type system in order to provide a sound foundation for reasoning.

It is important to understand this theory in order to understand Coq. The CoC is a higher-order typed lambda calculus, and it extends the Curry-Howard isomorphism between programs and proofs in intuitionistic propositional logic. As a programming language foundation, this is important because it admits logical propositions (`Prop`) as one of the fundamental types. By extension, mathematical constructions can be correctly defined by their formal definitions, as expressed in terms of propositions. Furthermore, proofs of propositions can be completed by demonstrating type inhabitance.

4.1.2.1 Type Inhabitance

In Coq, the notion of type inhabitance is used to prove propositions by providing a witness to a type which expresses it. In the CoC if a type encoding a proposition has a witness, then the proposition must be true. The richness of Coq's type system allows this in a very natural way. Furthermore, propositions can also be dependently typed. We demonstrate this using a simple example.

Example 4.1.2. Consider the trivial proposition: `forall A, A → A`, where A is any proposition. We can use the natural-deduction style proof language prove it.

```
Lemma P1 : forall (A : Prop) , A -> A.
intro A. intro x. apply x. Qed.
```

This proposition is proven by applying `intro` to assume `A:Prop` and `x:A`, the witness to `A`. We then apply `x` to show that indeed, given any `A:Prop`, `A` implies `A`. More details about proofs and tactics including `intro` will be given in the next chapter. Next, let us define a function with the same signature.

```
Definition PFun : forall (A : Prop), A -> A := fun A x => x.
```

This can be read as the lambda expression $\lambda(A_{:Prop})(x_{:A}).x$ and it provides a witness to the type `forall (A : Prop), A -> A` which can be used directly in proofs as follows:

```
Lemma P2 : forall (A : Prop), A -> A.
apply PFun. Qed.
```

The existence of `PFun` proves that the type `forall (A : Prop), A -> A` is indeed inhabited, which is sufficient to show that the proposition is true. `P1` and `P2` are now equivalent proofs of the same proposition.

4.2 Russel's and Girard's Paradoxes

A system which is suitable for programming and proving should provide its users with an assuredness that the underlying logic is consistent. In a consistent logic it is not possible to prove contradictory formulae or paradoxes. This should be a crucial requirement for the reasoning component of any project employing formal methods. And since this project derives implementation from specification, it is crucial that all constructions are consistent.

Not all proof assistants satisfy this requirement, and some suffer from inconsistency by Girard's Paradox. This can be interpreted in a way which compares it to Russel's Paradox, which famously demonstrates the inconsistency of naive set theory. It uses naive set theory to construct the set of all sets that do not contain themselves as elements, which is inconsistent. The paradox can be summarized symbolically:

$$\textbf{if } R = \{x | x \notin x\} \textbf{ then } R \in R \iff R \notin R.$$

To address this inconsistency, axiomatic set theory was introduced and formalized as Zermelo-Fraenkel set theory (ZFC), which restricts set comprehension by distinguishing between elements which are themselves sets and those which are not.

Russel's paradox is an example of an impredicative construction. An impredicative type system is one in which types are quantified over themselves. In other words, they allow types to contain the constructions they are supposed to define, which often leads to logical inconsistency.

Investigations of consistency in type systems have been inspired by Russel's paradox and the formalization of ZFC. It has been shown that constructive type systems such as U and U^{-}, where both $Type$ and $Prop$ are impredicative, suffer from inconsistency. In other words, in such type systems it is possible to build proof terms of type $\bot$, or the canonical representation of $false$, which is certainly not a desirable property.

Considerable work has been done by Coquand et. al. to understand this and, in practice, is resolved by the inclusion of type predicativity in Coq. Predicative type systems do not allow types to be defined in terms of themselves. In practice, however, this can be obscured from the user and managed by the type system, which Coq handles automatically. For more details, we refer to [15].

4.3 Predicative Types and Universe Variables

Coq uses the notion of distinct universe levels to avoid circular type dependency. It achieves this by assigning universe variables to types which classify other types as part of a theoretically infinite hierarchy of types. If we think of sets, for example, the inconsistent construction described in Russel's paradox would no longer be inconsistent because Coq would enforce that two distinct levels of sets are used in the definition.

Conveniently, all type level inferences are done automatically, so there is no need to manage this manually. This is due to Coq's predicative type system, which is actively enforcing that no quantified type object has its quantifiers instantiated with the object itself. For more details we refer to [7].

In practice this ensures that types derived from other types are classified properly. In addition to avoiding inconsistent constructions, this also ensures that type hierarchies reflect their intended definitions and that implementation is distinct from specification. We demonstrate this with a continued example.

Example 4.3.1. In Section 4.1 we introduced the type `BMatrices`. We will use Coq's type checker to see where this fits in the type hierarchy.

First we need a variable with appropriate label and type:

```
Variable (M1 : BMatrix 3 2).
```

Using the `Check` command, we can check its type:

```
Check M1.
   M1 : BMatrix 3 2
```

Coq's response to this query is that `M1` indeed has type `BMatrix` 3 2, which does not require any further classification. We can proceed by checking the type of `BMatrix` 3 2 itself.

```
Check BMatrix 3 2.
   BMatrix 3 2 : Set
```

Coq classifies `BMatrix` 3 2 as a `Set`, namely the set of all 3 by 2 `BMatrices`. Let us consider two more type queries:

```
Check Set.
   Set : Type (* (Set)+1 *)
Check Type.
   Type (* Top.4 *) : Type (* (Top.4)+1 *)
```

With these two queries, we begin to see Coq adding universe variables to type classifications. The classification of `Set` requires a universe variable which is greater than that of `Set` itself. This is given by `(* (Set)+1 *)`, which tells us that Coq assigned a universe variable one level higher than `Set`'s. This explicit classification of `Set` exemplifies the predicativity of the type system.

Finally the last query shows that `Type` itself must be classified by a higher type in the hierarchy. The universe variables `Top.4` and `Top.4+1` are automatically computed and assigned by the type system.

Chapter 5

The Coq Proof Assistant

In this section we give by example an informal overview of Coq for the purpose of describing the basic features used in the implementation. Some features which we used only infrequently will be discussed in the following chapters as needed. There are many features of Coq which were not used and are therefore not discussed. For more information about Coq, we refer to [8].

5.1 Set, Prop and Type

Coq has three fundamental type classifications called *sorts*. These are `Set, Type` and `Prop`. `Set` is the universe of all possible program types and specifications, while `Prop` is the universe of logical propositions. As mentioned earlier, propositions represent the types of proof terms. A proposition is only proved when the type representing it is inhabited. `Type` encompasses both `Set` and `Prop`.

Every term has exactly one type which can be checked by the user at any time during an interactive session using the command `Check`.

```
Check 3.
   3 : nat
Check nat.
   nat : Set
Check plus.
   plus : nat -> nat -> nat
```

Every program statement must be terminated by a period. As mentioned, every proof specification is a `Prop`,

```
Check forall a b, a + b = b + a.
   forall a b, a + b = b + a : Prop
```

while a proof of a `Prop` is a witness to its type. Remark that here the types of `a` and `b` are inferred by the types of the function they appear in, as they sometimes are. Generally it is necessary to specify the types of variables, i.e. replacing `forall a b` with `forall (a b : nat)`.

We provide a witness to this proof type by identifying the proposition as a `Lemma` and switching to the proof language and providing a completed, saved proof.

5.2 Proofs and Tactics

As mentioned above, propositions in Coq are proved by using tactics and applying other proved facts. In this work, most of the proofs are completed methods commonly used in the literature. We make extensive use of induction and substitution rules.

It should be noted that in many proofs, use of the tactic `induction` does not necessarily mean that we are proving a goal by induction. In Coq, this tactic is in some cases synonymous with `destruct` or `elim`, which all break down an inductively defined object into its distinct cases. Sometimes this is needed to prove by case analysis. In other cases, we really mean to prove by induction, for example where lists are involved.

In many proofs, substitution rules are required to make progress towards proving a goal. This means that associative, distributive, commutative and other laws are explicitly encoded and applied when needed. Let us consider a simple example by proving the commutativity of `+` over natural numbers using Coq's interactive command line mode.

```
Coq < Lemma Plus_Commutative : forall (a b : nat), a + b = b + a.
1 subgoal

  ============================
   forall a b : nat, a + b = b + a
```

The first line is typed by the user. We wish to show that `+` is a commutative operator. Coq generates the proof goal, as shown beneath the horizontal line. We will begin the proof by applying an introduction `intros` rule to create variables for `a` and `b`.

```
Plus_Commutative < intros.
1 subgoal
```

```
  a : nat
  b : nat
  ============================
   a + b = b + a
```

The tactic `intros` creates and names variables, placing them into the local context - the set of assumptions available for proving the current goal. It does this for a variety of formulae, including universally quantified expressions as seen here. It also acts as an implication introduction rule, where a goal's antecedent is added to the local context and a name is given.

We proceed by inductive reasoning on `a`.

```
Plus_Commutative < induction a.
2 subgoals

  b : nat
  ============================
   0 + b = b + 0

subgoal 2 is:
 S a + b = b + S a
```

The tactic `induction a` generates a subgoal for each of the cases in the inductive definition of `nat` - `0` and `S` for the successor function.

```
Plus_Commutative < auto.
1 subgoal

  a : nat
  b : nat
  IHa : a + b = b + a
  ============================
   S a + b = b + S a
```

A call to `auto` combines assumption tactics and introduction/reduction rules to attempt to solve the current goal. It is able to solve the base case of this proof automatically.

```
Plus_Commutative < simpl.
1 subgoal

  a : nat
```

```
  b : nat
  IHa : a + b = b + a
  ============================
   S (a + b) = b + S a
```

The tactic `simpl` combines recursively applied expansion and reduction tactics to produce a simplified expression. With the now simplified expression, we can apply the induction hypothesis `IHa` to the goal.

```
Plus_Commutative < rewrite IHa.
1 subgoal

  a : nat
  b : nat
  IHa : a + b = b + a
  ============================
   S (b + a) = b + S a
```

Notice the use of a `rewrite` rule. We can substitute equal terms of an expression to make progress towards a goal. If an assumed equation matches the current goal we would call `apply` rather than `rewrite`.

A call to `auto` manages to solve the goal, and the keyword `Qed` closes the proof. A summary of the applied tactics is given followed by confirmation that the lemma has been successfully defined.

```
Plus_Commutative < auto.
No more subgoals.

Plus_Commutative < Qed.
 intros.
 induction a.
 auto.
 simpl.
 rewrite IHa.
 auto.
Plus_Commutative is defined
```

An understanding of the tactics discussed so far is sufficient to understand most of the proofs in this work, which are heavily based on applying or substituting proven lemmas.

For example, having already defined a lemma for the commutativity of plus, we may wish to apply it in other proofs.

```
Coq < Lemma Plus_C : forall a b c, a + b + c = c + b + a.
1 subgoal

  ============================
   forall a b c : nat, a + b + c = c + b + a

Plus_C < intros.
   1 subgoal

     a : nat
     b : nat
     c : nat
     ============================
      a + b + c = c + b + a
```

To prove this goal, we use the previous lemma as a rewrite rule. That is, for all `a` and `b`, occurrences matching `a + b` may be substituted by `b + a`. There are generally three different ways to apply rewrite rules - without term matching, with partial term matching and with full term matching.

```
Plus_C < rewrite (Plus_Commutative a _).
1 subgoal

  a : nat
  b : nat
  c : nat
  ============================
   b + a + c = c + b + a
```

Here we are applying partial term matching. `rewrite (Plus_Commutative a _)` will substitute the first occurrence of `a + _` with `_+ a`, where `_` is a wildcard. This is necessary because a rewrite rule may apply in more than one term of an expression, as is the case here.

Next we will tell Coq that only an occurrence of `(b + a) + c` should be substituted by `c + (b + a)`.

```
Plus_C < rewrite (Plus_Commutative (b+a) c).
1 subgoal
```

```
  a : nat
  b : nat
  c : nat
  ============================
   c + (b + a) = c + b + a
```

A call to `auto` is insufficient to complete this proof automatically. It may be obvious that `+` is an associative operator, but we need to convince Coq that this is true. Luckily, this fact is built into the arithmetic library and the proof is easily completed.

```
Plus_C < apply plus_assoc.
No more subgoals.

Plus_C < Qed.
 intros.
 rewrite (Plus_Commutative a _).
 rewrite (Plus_Commutative (b + a) _).
 rewrite plus_assoc.
Plus_C is defined
```

From this point on, we will denote omitted proofs, proof steps or program statements by `(***)`. All implementation programs and proofs can be found unedited in the appendix.

5.3 Structures

Now that we have seen some details about proofs in Coq, we turn our attention to program constructions. As mentioned earlier, we can define new types using structures. We are mainly interested in creating types which are a combination of objects, operations and properties. Consider the type of non-empty lists.

```
Structure NEList A := {
   ls : list A;
   ls_not_empty : [] <> ls
}.
```

This structure simply combines a list containing elements of any fixed type and a proof that the list is not empty. To build an instance of `NEList` we need to provide a value for `ls` and a proof of the proposition `ls_not_empty`.

We can define a list using notation similar to other functional programming languages. We define `xs` to be a singleton list containing a single element `1`.

```
Definition xs := [1].
```

We then use a basic rule about lists from Coq's `List` module to show that `[1]` is not empty,

```
Lemma xs_ne : [] <> xs.
apply nil_cons.
Qed.
```

where `nil_cons` is a proof of the fact that any list which has a head and a tail cannot be empty, and since `[1]` is a shorthand for `1 :: nil`, the rule applies.

Now, in any situation where we must use or reason with non-empty lists, we can use the type `NEList` to make available as an assumption the fact the embedded list is certainly not empty.

The fields of a structure can be accessed via their projection functions. Given a type `A`, and a non-empty list `nelA : NEList A`, we access its fields as follows:

```
Check (ls _ nelA).
   : list A
Check (ls_not_empty _ nelA).
   : [] <> ls A nelA
```

Here the middle argument `_` indicates that we wish the parameter `A` to be inferred, if possible.

5.4 Functions

Functions are, of course, the bedrock of any functional programming language, and Coq allows users to work with them in a familiar way to other languages. In Coq functions are specified in curried form, i.e. a function $f : (x, y) \to z$ is specified by `f : x -> y -> z`, and may be specified and passed as parameters. Consider the following example.

```
Definition Ap : (nat -> nat -> nat) -> nat -> nat -> nat :=
   fun Op a b => (Op a b).
```

The function `Ap` takes a binary operator over `nat` and two more arguments of type `nat`. The keyword `fun` followed by the variable names `Op a b` binds them to the corresponding types in the function body where the operator `Op` is applied to `a` and `b`.

There are other ways to define and label function arguments. We can define `Ap` equivalently by

```
Definition Ap (Op : nat -> nat -> nat) (a : nat) (b : nat) : nat :=
   Op a b.
```

5.4.1 Fixpoint

Coq uses the keyword `Fixpoint` to denote the definition of a recursive function. Recursive functions must be decreasing on some argument and Coq enforces this to ensure termination. Let us consider an example which also demonstrates inductive pattern matching.

```
Fixpoint fac (n : nat) :=
   match n with
      | 0 => 1
      | S x => S x * fac x
   end.
```

The factorial of a natural number is defined recursively. We define the function by matching the argument with one of the two clauses for natural numbers - either 0 or the successor of some number. In the base case, corresponding to the basis definition for natural numbers, 1 is returned. In the inductive case, a recursive call is made to `fac` which is decreasing in its argument. Induction and recursion are used extensively in this work, especially where lists are involved.

5.4.2 Lambda Expressions

Lambda expressions are used to define anonymous functions. This is especially useful for writing bounded expressions where variables are not necessarily assumed in advance.

For example we can use a lambda expression to directly write an anonymous function for another function's parameter.

```
Check Ap (fun a b : nat => fac a + fac b).
   Ap (fun a b : nat => fac a + fac b) : nat -> nat -> nat
```

5.5 Infix Operators, 'Notation'

For expressing properties and completing proofs, it is often very helpful to use operator symbols instead of function names. Infix operators are defined as follows.

```
Infix "+" := plus (at level 50).
```

This defines `+` to be a left-associative operator for `plus`, at a precedence level of 50. In Coq a lower level has higher precedence. Operators can be assigned right associativity by appending "`,right associativity`" after the level declaration.

Postfix operators are defined as follows.

```
Notation "n !" := (fac n) (at level 50).
```

Here `n` is assumed to be a variable of the appropriate type for the function it appears in on the right hand side of the notation declaration. Precedence and associativity are assigned as they are for infix operators.

5.6 Prop vs. bool

`Prop` is the type of logical propositions. They are used to express properties and reason about program constructions. It is generally not the case that a `Prop` necessarily evaluates or computes to true or false.

For example, the following proposition about `Prop` can not be proven without making additional assumptions about `Prop`

```
Lemma P : forall (p q : Prop), p = q \/ p <> q.
```

which states that for all propositions p and q, either $p = q$ or $p \neq q$.

Because `Prop` is not inductively defined, there are no tools for reasoning by case analysis. Furthermore some propositions can neither be proven true nor false. This is due to Coq's implementation of constructive logic, which as a consequence does not assume the law of excluded middle holds. That is, Coq does not assume by default that, for all propositions p, $p \vee \neg p$.

This is in stark contrast to `bool`, the type of Boolean values. The following proposition is easily proven by case analysis on the Boolean values.

```
Lemma B : forall (b c : bool), b = c \/ b <> c. (***)
```

This relationship has to be understood and applied to define new types which are involved in computation. More on this will be discussed in the next chapter.

Chapter 6

$\mathcal{L}$-Fuzzy Relations in Coq

In this chapter we present our implementation of concrete, finite $\mathcal{L}$-fuzzy relations with an emphasis on operations. Then we specify and prove properties about $\mathcal{L}$-fuzzy relations. Because we are using three different levels of categories, we have separated the proofs of required properties accordingly. In the next chapter we will apply these to formalize an algebraic theory of $\mathcal{L}$-fuzzy relations. The complete source code with proofs can be found in the online / digital appendix. We omit the bodies of most proofs in this document.

We begin by discussing type structures which are necessary for defining $\mathcal{L}$-fuzzy relations and operations on them. Though type classes are available in Coq, we opted not to use them as they are still an experimental feature. Instead, we often define structures for combining types and use their projection functions.

6.1 Types

We have made an effort to define objects and operations using the most general type available, but sometimes constraints on types must be made, for example to ensure decidability so that operations are computable. We use the following type constructions as needed.

6.1.1 Decidable Types

Because we need to have computable if-then-else constructions for certain operations, we need to have decidable types and a computable equality function. A type is decidable

if for any pair of elements, they are either equal or not equal. But computable equality is only possible via a Boolean-valued function.

Consider the identity relation as defined Definition 2.8. It may seem possible to implement it using a `Prop` such as

```
(x=y -> IdRel x y = Top ) /\ (x<>y -> IdRel x y = Bot)
```

however for reasons discussed in Section 5.6, this is not generally computable. Instead, we would like to define the identity using an if-then-else construction. To make this possible, we define `DType` for decidable types as follows:

```
Structure DType := {
   DT : Type;
   Deq : forall x y : DT, {x = y} + {x <> y};
   CDeq x y := if Deq x y then true else false
}.
```

where `DT` is the carrier type, `Deq` is a proof of the type being decidable, and `CDeq` transforms this property into a Boolean-valued function. Remark that fields of a structure using the assignment operator `:=` are just pseudonyms for other objects or functions, not additional requirements.

The property `Deq` can be understood as follows. For an arbitrary type, it is not generally true that the equality of two elements is decidable. This extends from the fact discussed in the last chapter, namely that propositions do not necessarily evaluate to true or false.

The notation `x = y + x <> y` is a type construction indicating that its elements must be proven to satisfy one property or the other, that two elements are decidably either equal or not. This is an example of a coproduct in category theory, specifically of the disjoint union of sets.

The function `CDeq` exploits the decidability property to provide a computable equality function, which is exactly what we need to define certain operations, as we will see later.

In addition to the above property, we define an infix operator and prove two lemmas which are useful in later proofs.

```
Infix "===" := (CDeq) (at level 69, right associativity).
Lemma CDeqProp (A : DType) :
  forall (x y : DT A), x = y <-> x === y = true. (***)
Lemma CDeqPropFalse (A : DType) :
  forall (x y : DT A), x <> y <-> x === y = false. (***)
```

`CDeqProp` states that elements are equal if and only if they are deemed equal under the decidable equality property `Deq`. This is because `Deq` states that for all pairs of elements we have a proof either of type `x=y` or `x<>y`. That allows us to do a case distinction on the proof type for a pair of elements. The converse property `CDeqPropFalse` follows from similar reasoning.

6.1.2 Finite Decidable Non-Empty Types

Finite decidable non-empty types (`FType`s) are used to build non-empty finite source and target sets for relations. Without them, we would not be able to define certain operations which consider all the elements of a type, such as composition. We define `FType`s as being decidable types together with an enumeration of elements in the form of a list, a proof that this list is complete, and a proof that the list is not empty.

```
Structure FType := {
   D : DType;
   FT := DT D;
   Elements : list FT;
   Prf_All_Elements : forall (a : FT), In a Elements;
   Prf_Elements_Not_Empty : Elements <> []
}.
```

6.2 Heyting Algebras

In this framework lattices are used for two purposes. We have a lattice of membership values for relations, and we have the lattice of relations themselves as is required to form a Heyting category. To be more precise, the lattices we use are Heyting algebras (`HeytA`). First we define the operations.

6.2.1 Operations

We define a Heyting algebra (Definition 2.5) in two parts. First we have the `HeytA` operations.

```
Structure HeytAOps (LA : Type) := {
   Carrier := LA;
   Meet : LA -> LA -> LA;
   Join : LA -> LA -> LA;
```

```
    PsComp : LA -> LA -> LA;
    Bot : LA;
    Top : LA;
    LUB : list LA -> LA := fun ls => fold_right Join Bot ls;
    GLB : list LA -> LA := fun ls => fold_right Meet Top ls;
    Order : LA -> LA -> Prop := fun x y => Meet x y = x
 }.
```

`HeytAOps` is parametric on `LA`, the underlying type of lattice elements. We need to have a type-parametric definition to be able to work with lattices of relations having variable source and target types. Then for operations we have respectively the binary meet, join and relative pseudo-complement operations, the bottom and top elements, and finally subset LUB and GLB operations and the order relation.

Notice that `LUB` and `GLB` are defined in terms of their binary counterparts using fold functions over lists. Because of this definition, there is no need to prove either completeness property - which state that there must exist a least upper bound and greatest lower bound for every subset of lattice elements.

Notice also that the order relation between elements is defined algebraically in terms of the meet operation, as opposed to order-theoretically.

6.2.2 Axioms

We define infix operators and shorthands for `HeytA` operations which are consistently used throughout the rest of this work before defining the collection of axioms.

```
Infix "&&&" := (Meet Ops) (at level 60, right associativity).
Infix "|||" := (Join Ops) (at level 61, right associativity).
Infix "-->" := (PsComp Ops) (at level 62, right associativity).
Infix "<<<" := (Order (Ops L')) (at level 63, right associativity).
Definition Top' := Top Ops.
Definition Bot' := Bot Ops.

Structure HeytAProps := {
    (* Algebraic Lattice Laws *)
    Join_Assoc : forall (a b c : LA), (a ||| b) ||| c = a ||| (b ||| c);
    Meet_Assoc : forall (a b c : LA), (a &&& b) &&& c = a &&& (b &&& c);
    Join_Absorp: forall (a b : LA), a ||| a &&& b = a;
```

```
    Meet_Absorp: forall (a b : LA), a &&& (a ||| b) = a;
    Join_Distr : forall (a b c : LA), a ||| (b &&& c)
      = (a ||| b) &&& (a ||| c);
    Join_Comm : forall (a b : LA), a ||| b = b ||| a;
    Meet_Comm : forall (a b : LA), a &&& b = b &&& a;
    (* Bounded Lattice Laws *)
    Join_Bot : forall (a : LA), a ||| Bot' = a;
    Meet_Top : forall (a : LA), a &&& Top' = a;
    (* Heyting Algebra Properties *)
    p1 : forall (a : LA), a --> a = Top';
    p2 : forall (a b : LA), a &&& (a --> b) = a &&& b;
    p3 : forall (a b : LA), b &&& (a --> b) = b;
    p4 : forall (a b c : LA), a --> (b &&& c) = (a --> b) &&& (a --> c)
 }.
```

The use of infix operators makes it very convenient to work with `HeytA`s in proofs and in programming.

We combine the operations and axioms into a third structure:

```
Structure HeytA (LA : Type) := {
  Ops : HeytAOps LA;
  Props : HeytAProps Ops
}.
```

In addition to the above axioms, there are many properties about `HeytA` operations which follow from the axioms. We specified and proved almost 40 lemmas about `HeytA` operations which are applied in later proofs about relations. These can be found in the digital appendix in `LatticeProperties.v`.

6.3 $\mathcal{L}$-Powersets

Before defining the type of $\mathcal{L}$-fuzzy relations, we first define the type of $\mathcal{L}$-powersets. These are modelled on the exponential objects of a Cartesian closed category. Recall that in the category of sets, the exponential object of two objects Z and Y is Z^Y, the function space $Y \to Z$.

The motivation for this definition is to prove a mapping property about `HeytA`s and `HeytA`-valued functions. Namely, we want to show that for some type `A` given `L :`

HeytA, the collection of all functions A -> L also forms a HeytA. This fact will allow us to easily construct the HeytA of relations for a Heyting category.

6.3.1 Operations

We define LPow as follows.

```
Definition LPow L A := A -> L.
```

Next we define the operations of an LPow-valued HeytA. Note that we consistently use L' as a variable HeytA and LA as its carrier type in every possible context.

```
Definition Meet' : LPow LA A -> LPow LA A -> LPow LA A :=
  fun f g x => (f x) &&& (g x).
Definition Join' : LPow LA A -> LPow LA A -> LPow LA A :=
  fun f g x => (f x) ||| (g x).
Definition PsComp' : LPow LA A -> LPow LA A -> LPow LA A :=
  fun f g x => (f x) --> (g x).
Definition Top' : LPow LA A :=
  fun x => Top.
Definition Bot' : LPow LA A :=
  fun x => Bot.
```

6.3.2 Properties

With LPow operations defined, we must now prove that they satisfy the axioms of a HeytA. This is a very straightforward task with the help of an extensionality property on LPow functions.

```
Lemma Eq_LPow (f g : LPow LA A) : (f = g) <-> (forall x, f x = g x).
```

This is easily proved by applying a generic rule for functional extensionality. We proceed with the axioms,

```
Lemma Join_Assoc : forall (f g h : LPow LA A),
  (f ||| g) ||| h = f ||| (g ||| h).
Lemma Meet_Assoc : forall (f g h : LPow LA A),
  (f &&& g) &&& h = f &&& (g &&& h).
Lemma Join_Absorp: forall (f g : LPow LA A), f ||| f &&& g = f.
Lemma Meet_Absorp: forall (f g : LPow LA A), f &&& (f ||| g) = f.
Lemma Join_Distr : forall (f g h : LPow LA A),
```

```
  f ||| (g &&& h) = (f ||| g) &&& (f ||| h).
Lemma Join_Comm : forall (f g : LPow LA A), f ||| g = g ||| f.
Lemma Meet_Comm : forall (f g : LPow LA A), f &&& g = g &&& f.
Lemma Join_Bot : forall (f : LPow LA A), f ||| Bot’ = f.
Lemma Meet_Top : forall (f : LPow LA A), f &&& Top’ = f.
Lemma p1 : forall (f : LPow LA A), f --> f = Top’.
Lemma p2 : forall (f g : LPow LA A), f &&& (f --> g) = f &&& g.
Lemma p3 : forall (f g : LPow LA A), g &&& (f --> g) = g.
Lemma p4 : forall (f g h : LPow LA A),
  f --> (g &&& h) = (f --> g) &&& (f --> h).
```

and build the `HeytA` of `LPows`

```
Definition LPowOps :=
  Build_HeytAOps (LA:=LPow LA A) (Meet’ L’)
  (Join’ L’) (PsComp’ L’) Bot’ Top’.

Definition LPowProps :=
  Build_HeytAProps (LA:=LPow LA A) (Ops := LPowOps)
  Join_Assoc Meet_Assoc Join_Absorp Meet_Absorp
  Join_Distr Join_Comm Meet_Comm Join_Bot Meet_Top p1 p2 p3 p4.

Definition LPowHeytA := Build_HeytA LPowProps.
```

6.4 $\mathcal{L}$-Fuzzy Relations

In this section, we first define the type of $\mathcal{L}$-fuzzy relations and their operations. Then we prove the properties about those operations necessary for formalizing them as an algebraic theory in the next chapter.

6.4.1 Operations

First we define the type of $\mathcal{L}$-fuzzy relations as `LRel`.

```
Definition LRel L A B := LPow (LPow L B) A.
```

These are simply functions `A -> B -> L` for arbitrary types. We impose restrictions on the types of `A`, `B` and `L` in the definitions of operations.

Before defining the operations, we need to declare a variable HeytA of membership values. In this implementation we are assuming a fixed-basis, meaning that all relations share a common HeytA of membership values. We declare

```
Variables (DLA : DType) (L' : HeytA (DT DLA)).
Definition LA := (DT DLA).
```

Now we have to show that the collection of LRels between arbitrary source and target sets forms a HeytA. This will be required to instantiate a Heyting category. Since LRels are defined in terms of LPows, we define the HeytA of LRels by directly applying the mapping property proven about LPows in the last section.

```
Definition LRel_Lat A B := LPowHeytA A (LPowHeytA B L').
```

Next, we define the various operations on LRels as found in Section 2.4.

First we define the composition of LRels. As described in earlier examples, this can be understood as taking the LUB of point-wise meets in the matrix representation of a relation in a similar fashion to matrix multiplication.

```
Definition Comp (A B C : FType) :
  LRel LA (FT A) (FT B) -> LRel LA (FT B) (FT C) -> LRel LA (FT A) (FT C) :=
    fun Q R x z => (LUB (map (fun y => (Q x y) &&& (R y z)) (Elements B))).
```

We assign an infix operator to composition.

```
Infix ">.>" := (Comp) (at level 64, right associativity).
```

The identity LRel over a type A is defined as having Top on the diagonal and Bot otherwise.

```
Definition IdRel (A : FType) : LRel LA (FT A) (FT A) :=
  fun x y => if x === y then Top else Bot.
```

The converse operation simply transposes source and target.

```
Definition Conv (A B : FType) :
  LRel LA (FT A) (FT B) -> LRel LA (FT B) (FT A) :=
    fun R y x => (R x y).
```

The left residual is defined similarly to composition, except that we take the GLB of point-wise relative pseudo-complements.

```
Definition LeftRes (A B C : FType) :
  LRel LA (FT A) (FT C) -> LRel LA (FT B) (FT C) -> LRel LA (FT A) (FT B) :=
    fun S R x y => GLB (map (fun z => R y z --> S x z) (Elements C)).
```

The arrow operations on `LRels` are defined component wise.

```
Definition Up' x := if x === Bot then Bot else Top.
Definition Down' x := if x === Top then Top else Bot.

Definition Up (A B : FType) :
  LRel LA (FT A) (FT B) -> LRel LA (FT A) (FT B) :=
    fun R x y => Up' L' (R x y).

Definition Down (A B : FType) :
  LRel LA (FT A) (FT B) -> LRel LA (FT A) (FT B) :=
    fun R x y => Down' L' (R x y).
```

The greatest and least relations map every pair to `Top` and `Bot` respectively.

```
Definition RTop (A B : FType) : LRel LA (FT A) (FT B) :=
  fun x y => Top.

Definition RBot (A B : FType) : LRel LA (FT A) (FT B) :=
  fun x y => Bot.
```

And finally, scalar relations (partial identities) are formed by setting all entries on the diagonal to a single `HeytA` element `a` and setting all other entries to `Bot`.

```
Definition RScalar (A : FType) (a : LA) :  LRel LA (FT A) (FT A) :=
  fun x y => if x === y then a else Bot.
```

6.4.2 Properties

As mentioned earlier, we separate the various properties which must be proven about `LRels` by their corresponding categorical construction. The following lemmas are about concrete `LRels`. Their labels co-incide with the properties of Theorem 3.1 (category and Dedekind properties) and Lemma 3.3 (arrow properties) in [31] and are summarized in Lemma 2.10 of this thesis.

In the next chapter, we will use these lemmas about concrete `LRels` to show that they form an instance of an abstract category, Dedekind category and finally an arrow category.

6.4.2.1 Category Properties

To form a category, we will need to show two basic properties; namely that the identity laws are satisfied and that composition is associative. In terms of `LRels` this means that we need to show the following.

```
Lemma P1_R : forall (A B : FType) (Q : LRel LA (FT A) (FT B)),
  Q >.> (IdRel (A:=B) L') = Q.

Lemma P1_L : forall (A B : FType) (Q : LRel LA (FT A) (FT B)),
  (IdRel (A:=A) L') >.> Q = Q.

Lemma P2 : forall (A B C D: FType) (Q : LRel LA (FT A) (FT B))
  (R : LRel LA (FT B) (FT C)) (S : LRel LA (FT C) (FT D)),
    (Q >.> R) >.> S = Q >.> (R >.> S)
```

6.4.2.2 Dedekind Properties

Heyting categories extend the base category by adding the converse and residual operations as well as an interpretation for scalar relations. The properties of these operations are the same for a Dedekind or a Heyting category.

We define infix operators for the meet, order and left residual operations on relations.

```
Infix "&R&" := (RMeet L') (at level 64, right associativity).
Infix "<R<" := (ROrder L') (at level 68, right associativity).
Infix "/R/" := (LeftRes L') (at level 67, right associativity).
```

We proceed by proving the following properties about `LRels`.

```
Lemma P3 : forall (A B : FType) (Q1 Q2 : LRel LA (FT A) (FT B)),
  Conv (Q1 &R& Q2) = (Conv Q1) &R& (Conv Q2).
Lemma P4 : forall (A B C : FType) (Q : LRel LA (FT A) (FT B))
  (R : LRel LA (FT B) (FT C)),
    Conv (Q >.> R) = (Conv R) >.> (Conv Q).
Lemma P5 : forall (A B : FType) (Q : LRel LA (FT A) (FT B)),
  Conv (Conv Q) = Q.
Lemma P7 : forall (A B C : FType) (Q : LRel LA (FT A) (FT B))
  (R : LRel LA (FT B) (FT C)) (T : LRel LA (FT A) (FT C)),
    Q >.> R &R& T <R< Q >.> (R &R& (Conv Q) >.> T).
Lemma ResP1 : forall (A B C : FType) (R1 R2 : LRel LA (FT A) (FT C))
```

```
  (S : LRel LA (FT B) (FT C)),
    (R1 &R& R2) /R/ S <R< (R1 /R/ S) &R& (R2 /R/ S).
Lemma ResP2 : forall (A B C : FType) (T : LRel LA (FT A) (FT B))
  (S : LRel LA (FT B) (FT C)),
    T <R< (T >.> S) /R/ S.
Lemma ResP3 : forall (A B C : FType) (R : LRel LA (FT A) (FT C))
  (S : LRel LA (FT B) (FT C)),
    (R /R/ S) >.> S <R< R.
Lemma ScalarProp : forall (A : FType) (R : LRel LA (FT A) (FT A)),
  (R <R< IdRel(A:=A) L' /\
  RTop (A:=A)(B:=A) L' >.> R = R >.> RTop (A:=A)(B:=A) L')
    -> (forall x y, R x x = R y y).
```

6.4.2.3 Arrow Properties

Arrow categories extend Dedekind/Heyting categories by adding the arrow operations. We prove the following properties about the arrow operations. These correspond to the requirements of Definition 2.13.

```
Lemma AP1_1 : forall (A B : FType) (Q R : LRel LA (FT A) (FT B)),
   Up L' Q <R< R  ->  Q <R< Down L' R.
Lemma AP1_2 : forall (A B : FType) (Q R : LRel LA (FT A) (FT B)),
  Q <R< Down L' R -> Up L' Q <R< R.
Lemma AP2 : forall (A B C : FType) (R : LRel LA (FT B) (FT A))
  (S : LRel LA (FT B) (FT C)),
    Up L' ((Conv R) >.> (Down L' S)) = (Conv (Up L' R)) >.> (Down L' S).
Lemma AP3 : forall (A B : FType) (Q R : LRel LA (FT A) (FT B)),
   (Up L' (Q &R& (Down L' R))) = ((Up L' Q) &R& (Down L' R)).
Lemma AP4 : forall (A : FType) (R : LRel LA (FT A) (FT A)),
  R <R< IdRel(A:=A) L' /\
  RTop (A:=A)(B:=A) L' >.> R = R >.> RTop (A:=A)(B:=A) L' /\
  R <> RBot (A:=A)(B:=A) L' -> Up L' R = IdRel (A:=A) L'.
```

Chapter 7

Arrow Categories in Coq

In this chapter we first define the types of categories, Heyting categories and arrow categories. We then construct an instance of an arrow category having `FType`s as objects and `LRel`s between `FType`s as the morphisms by applying the proofs about `LRel`s discussed in the previous chapter.

7.1 Categories

7.1.1 `Category`

We begin by defining a type for a base category. The following definition is based on the type `Category` in ConCaT - a freely-available package for constructive category theory in Coq [14]. The main difference here is that we chose not to use setoids for the collection of morphisms. Instead we simply use an arbitrary type. The advantage of using setoids is that they allow any equivalence relation to be used to define equivalent morphisms. In our implementation, morphisms are equivalent only when equal. The following implements Definition 3.1.

```
Structure Category : Type := {
   Ob : Type;
   Hom : Ob -> Ob -> Type;
   CComp : forall a b c : Ob, (Hom a b) -> (Hom b c) -> (Hom a c);
   Id : forall a : Ob, Hom a a;
   Comp_Assoc : forall (a b c d : Ob)
   (f : Hom a b) (g : Hom b c) (h : Hom c d),
      CComp (CComp f g) h = CComp f (CComp g h);
   Idl_law : forall (a b : Ob) (f : Hom a b), CComp (Id _) f = f;
```

```
    Idr_law : forall (a b : Ob) (f : Hom a b), CComp f (Id _) = f
 }.
```

A category consists of a type for the objects, the Hom-set of morphisms, a composition operation, an identity morphism, and proofs of the associativity of composition and of the two identity laws.

Finally, we define an infix operator for morphisms.

```
Infix "-->" := Hom (at level 95, right associativity).
```

7.1.2 Category LREL

Now we want to construct `LREL` - the category with `FType`s as objects and `LRel`s as morphisms. Since we have already proven all the required properties about concrete `LRel`s, this is a simple task.

Since `LRel`s are parametric on a single `HeytA` of membership values, we declare this as a variable.

```
Variables (DLA : DType) (L' : HeytA (DT DLA)).
```

We then build a `Category` by applying the required proofs.

```
Definition LREL := Build_Category (P2 L') (P1_L L') (P1_R L').
```

7.2 Heyting Categories

Our next step is to define the type of Heyting categories and show that `LREL` together with additional operations and properties forms a Heyting category. Since our implementation is geared towards modelling only finite relations, we use the `HeytA` defined earlier for the lattice of relations.

From a theoretical point of view, the following construction loses generality over infinite relations, but since we are interested in computing with relations, this concession does not have any serious impact aside from assuming that lists, and not a more general collection type, are used to model subsets of elements.

7.2.1 Heyting Category

A `HeytingCategory` extends a base `Category` by requiring that the collection of morphisms forms a `HeytA`, adds the converse and residual operations and requires that they satisfy the axioms of a Heyting category. The following implements Definition 3.6.

```
Structure HeytingCategory : Type := {
   FC : Category;
   HomLattice (A B : Ob FC) : HeytA (A-->B);
   OpConv (A B : Ob FC) : (A-->B) -> (B-->A);
   ConvP0 : forall (A B : Ob FC) (Q R : A-->B),
     OpConv ((Meet(Ops (HomLattice A B))) Q R) =
       (Meet(Ops (HomLattice B A))) (OpConv Q) (OpConv R);
   ConvP1 : forall (A B C : Ob FC) (Q : A-->B) (R : B-->C),
     OpConv (Q >.> R) = (OpConv R) >.> (OpConv Q);
   ConvP2 : forall (A B : Ob FC) (Q : A-->B),
     OpConv (OpConv Q) = Q;
   ModularLaw : forall (A B C : Ob FC)
     (Q : A-->B) (R : B-->C) (S : A-->C),
      (Order(Ops (HomLattice A C)))
        (Meet (Ops (HomLattice A C)))
          (Q >.> R) S)
          (Q >.> (Meet(Ops (HomLattice B C)) R ((OpConv Q) >.> S)));
   OpLRes : forall (A B C : Ob FC),
     (A-->C)-> (B-->C) -> (A-->B);
   ResP1 : forall (A B C : Ob FC) (R1 R2 :A -->C) (S : B-->C),
   Order(Ops(HomLattice A B))
     (OpLRes (Meet(Ops (HomLattice A C)) R1 R2) S)
     (Meet(Ops (HomLattice A B)) (OpLRes R1 S) (OpLRes R2 S));
   ResP2 : forall (A B C : Ob FC) (T : A-->B) (S : B-->C),
     Order(Ops(HomLattice A B)) T (OpLRes (T >.> S) S);
   ResP3 : forall (A B C : Ob FC) (R : A-->C) (S : B-->C),
     Order(Ops(HomLattice A C)) ((OpLRes R S) >.> S) R
}.
```

Notice that we use the notation `A-->B` to denote a morphism from `A` to `B`, i.e. `Hom A B`, in the category `C`. Also, since `Meet` and `Order` are parametric on their `HeytA`, we cannot conveniently define infix operators for them in the definition.

7.2.2 HeytingCategory `HLREL`

Now we proceed by constructing `HLREL` by extending `LREL`. Once again, we first declare a variable `HeytA` of membership values and then construct `HLREL` by providing the required objects, operations and proofs from the concrete level.

```
Variables (DLA : DType) (L' : HeytA (DT DLA)).
Definition LA := DT DLA.

Definition HLREL :=
  Build_HeytingCategory
  (FC:=LREL L')
  (HomLattice := fun (A B : FType) => LRel_Lat L' (FT A) (FT B))
  (OpConv := Conv (DLA:=DLA))
  (P3 L') (P4 L') (P5 (DLA:=DLA)) (P7 L')
  (ResP1 L') (ResP2 L') (ResP3 L').
```

7.3 Arrow Categories

We define the type `ArrowCategory`. This definition corresponds almost directly to Definition 3.7, except that the Galois correspondence axiom is split into two implications.

7.3.1 `ArrowCategory`

We implement `ArrowCategory` as an extension of a Heyting category.

```
Structure ArrowCategory : Type := {
  FD : HeytingCategory;
  OpUp (A B : Ob (FC FD)) : (A --> B) -> (A --> B);
  OpDown (A B : Ob (FC FD)) : (A --> B) -> (A --> B);
  arrow_1_1 : forall (A B : Ob (FC FD)) (Q R : A --> B),
    (OpUp Q) <<< R ->  Q <<< (OpDown R);
    arrow_1_2 : forall (A B : Ob (FC FD)) (Q R : A --> B),
    Q <<< (OpDown R) ->  (OpUp Q) <<< R ;
  arrow_2 : forall (A B C : Ob (FC FD)) (R : B --> A) (S : B --> C),
    OpUp ((OpConv R) >.> (OpDown S)) = (OpConv (OpUp R)) >.> (OpDown S);
  arrow_3 : forall (A B : Ob (FC FD)) (Q R : A --> B),
    OpUp (Q &&& OpDown R) = (OpUp Q &&& (OpDown R));
```

```
arrow_4 : forall (A : Ob (FC FD)) (R : A-->A),
  (Order (Ops(HomLattice A A))) R (Id A) /\
  (ATop A A) >.> R = R >.> (ATop A A) /\ ~(R = ABot A A) ->
    (OpUp R) = Id A
  }.
```

7.3.2 Arrow Category ALREL

Finally, we show that HLREL along with the arrow operations and properties defined and proven on LRels forms an arrow category.

```
Variables (DLA : DType) (L' : HeytA (DT DLA)).
Definition LA := DT DLA.

Definition ALREL :=
  Build_ArrowCategory
  (FD:=HLREL L')
  (OpUp := Up L')
  (OpDown := Down L')
  (AP1_1 (L':=L')) (AP1_2 (L':=L'))
  (AP2 L') (AP3 L') (AP4 (L':=L')).
```

Chapter 8

Computing with LRels

In this chapter we show how to build instances of all the components required to work with `LRels` as an instance of an arrow category.

8.1 Heyting Algebra

In this section we demonstrate how to define a Heyting algebra of membership values. For this example, we define a Heyting algebra for three possible membership values, `T`, `M`, and `F`. We start by defining the carrier type `L3`, which is simply an enumeration of the three values.

```
Inductive L3 : Type := F | M | T.
```

Next, we have to show that this is a decidable type. This follows from the fact that it is an enumeration type, and is proven automatically by the tactic `decide equality`. We define `DL3` as follows.

```
Lemma L3_Deq : forall (x y : L3), {x = y} + {x <> y}.
decide equality.
Defined.

Definition DL3 : DType := Build_DType L3_Deq.
```

Notice the use of the keyword `Defined` rather than `Qed` in the above lemma. This ensures that the entire proof term is retained in an executable form. We proceed by defining Heyting algebra operations for `L3`. Here we define them explicitly by case distinction and collect them into `L3_Ops`

```
Definition Meet : L3 -> L3 -> L3 :=
   fun a b => match a with
      | T =>  match b with | T => T | M => M | F => F  end
      | M =>  match b with | T => M | M => M | F => F  end
      | F =>  match b with | T => F | M => F | F => F  end
   end.

Definition Join : L3 -> L3 -> L3 :=
   fun a b => match a with
      | T =>  match b with | T => T | M => T | F => T  end
      | M =>  match b with | T => T | M => M | F => M  end
      | F =>  match b with | T => T | M => M | F => F  end
   end.

Definition PsComp : L3 -> L3 -> L3 :=
   fun a b => match a with
      | T =>  match b with | T => T | M => M | F => F  end
      | M =>  match b with | T => T | M => T | F => F  end
      | F =>  match b with | T => T | M => T | F => T  end
   end.

Definition Bot : L3 := F.
Definition Top : L3 := T.

Definition L3_Ops := Build_HeytAOps Meet Join PsComp Bot Top.
```

Next we prove that these operations indeed form a Heyting algebra. This is an easy task for Coq and can be completed by computation. For instance, the first of the 13 properties we need to prove is the associativity of `Join`, and it is proven as shown below.

```
Definition L3_Props : HeytAProps L3_Ops.
apply Build_HeytAProps.
intros; elim a; elim b; elim c; compute; auto.
(***)
Qed.
```

The rest of the properties are proven similarly, though with the appropriate number of operand eliminations. Lastly we use `L3_Props` to build a Heyting algebra.

```
Definition L3_Lat := Build_HeytA (LA:=DT DL3) L3_Props.
```

8.2 Source and Target Types

With our Heyting algebra of membership values defined, we must now construct FTypes for source and target sets. As an example, we will define an FType containing 4 elements. Since an FType extends a DType we begin by following the same steps as in the previous section.

```
Inductive E4 : Type := A4  | B4 | C4 | D4.

Definition E4_Deq : forall (x y : E4), {x = y} + {x <> y}.
decide equality.
Defined.

Definition DE4 : DType := Build_DType E4_Deq.
```

Next we have to provide a list of E4 elements and prove that this list contains every element of that type. Additionally, we must prove that the list is not empty. These are also very easy proofs which Coq can complete by computation. This will give us FE4, the FType over E4.

```
Definition E4Elements := [A4;B4;C4;D4].

Lemma E4PrfElements : forall (a : DT DE4), In a E4Elements. (***)

Lemma ElemNE : E4Elements <> []. (***)

Definition FE4 : FType := Build_FType E4PrfElements ElemNE.
```

We similarly define other source and target types FE3, a 3 element type, and FBool, the FType over bool.

8.3 Relations and Operations

In this section we are interested in computing with relations as the morphisms in an instance of an arrow category. The point of this is to guarantee that the behaviour of the operations is consistent with the algebraic theory.

We start by declaring an instance of an arrow category using L3_Lat - the Heyting algebra of membership values defined in the last section.

```
Definition L3_ALREL : ArrowCategory := ALREL L3_Lat.
```

Now we can define some examples of relations. Consider the relations `R : FBool --> FE3` and `Q : FE3 --> FE4` expressed first as matrices.

$$\mathtt{R} := \begin{array}{c} \\ \mathtt{true} \\ \mathtt{false} \end{array} \begin{array}{c} \begin{array}{ccc} \mathtt{A3} & \mathtt{B3} & \mathtt{C3} \end{array} \\ \begin{pmatrix} \mathtt{T} & \mathtt{M} & \mathtt{F} \\ \mathtt{M} & \mathtt{M} & \mathtt{F} \end{pmatrix} \end{array} \qquad \mathtt{Q} := \begin{array}{c} \\ \mathtt{A3} \\ \mathtt{B3} \\ \mathtt{C3} \end{array} \begin{array}{c} \begin{array}{cccc} \mathtt{A4} & \mathtt{B4} & \mathtt{C4} & \mathtt{D4} \end{array} \\ \begin{pmatrix} \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \end{pmatrix} \end{array}$$

We can define these in Coq as follows.

```
Definition R : FBool --> FE3 :=
  fun x y => match x with
      | true  =>  match y with | A3 => T | B3 => M | C3 => F  end
      | false =>  match y with | A3 => M | B3 => M | C3 => F  end
  end.


Definition Q : LRel L3 (FT FE3) (FT FE4) :=
  fun x y => match x with
      | A3 =>  match y with | A4 => T | B4 => M | C4 => F | D4 => F  end
      | B3 =>  match y with | A4 => T | B4 => M | C4 => F | D4 => F  end
      | C3 =>  match y with | A4 => T | B4 => M | C4 => F | D4 => F  end
  end.
```

Notice that the type of `R` is written using the notation for morphism while `Q` uses the notation for concrete $\mathcal{L}$-relations. At this level the notations are interchangeable.

We can now apply various operations and compute results. A function `RelApp` is defined to print a relation in row-major order. We use it to look at the relation `R` and the identity relation on `FE3`.

```
Eval compute in RelApp R.
  = [[T; M; F]; [M; M; F]]

Eval compute in RelApp (Id (c:=C') FE3).
  = [[T; F; F]; [F; T; F]; [F; F; T]]
```

Now, for example, we can compute the result of applying the operation up arrow operation `R` followed by composition with the identity relation. We first compute this operation 'by hand', and then show the computation performed in Coq.

$$R^{\uparrow}; \mathbb{I}_{\mathtt{FE3}} = \begin{pmatrix} \mathtt{T} & \mathtt{M} & \mathtt{F} \\ \mathtt{M} & \mathtt{M} & \mathtt{F} \end{pmatrix}^{\uparrow} ; \begin{pmatrix} \mathtt{T} & \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{T} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} & \mathtt{T} \end{pmatrix} = \begin{pmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} \\ \mathtt{T} & \mathtt{T} & \mathtt{F} \end{pmatrix}$$

```
Eval compute in RelApp (OpUp R >.> (Id (c:=C') FE3)).
  = [[T; T; F]; [T; T; F]]
```

Or as final example, we can combine other operations as desired. Consider the following computation first 'by hand' and then by using Coq.

$$Q^{\smile\downarrow};R^{\smile} = \begin{pmatrix} \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \end{pmatrix}^{\smile\downarrow} ; \begin{pmatrix} \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{M} & \mathtt{F} & \mathtt{F} \end{pmatrix} = \begin{pmatrix} \mathtt{T} & \mathtt{T} & \mathtt{T} \\ \mathtt{F} & \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} & \mathtt{F} \end{pmatrix} ; \begin{pmatrix} \mathtt{T} & \mathtt{M} \\ \mathtt{M} & \mathtt{M} \\ \mathtt{F} & \mathtt{F} \end{pmatrix} = \begin{pmatrix} \mathtt{T} & \mathtt{M} \\ \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} \end{pmatrix}$$

```
Eval compute in RelApp ((OpDown (OpConv Q) >.> OpConv R)).
  = [[T; M]; [F; F]; [F; F]; [F; F]]
```

Chapter 9

Conclusion and Future Work

In this thesis we have presented a framework for reasoning and computing with $\mathcal{L}$-fuzzy relations in categorical constructions based on Dedekind and arrow categories. This can be used to specify, reason about, and implement applications based on $\mathcal{L}$-fuzzy relations such as fuzzy controllers or fuzzy databases. For this all to be possible in the same language bridges a gap between specification and implementation. We hope that projects like this will help increase the adoption of functional programming languages and interactive proof assistants for formal software development and verification.

In future work, we will construct detailed examples of programs specified and developed in Coq using our framework. Furthermore, for the sake of formalizing the algebraic theory, we would like to implement the original definition of Dedekind categories and complete Heyting algebras, which would not require that subsets of lattice elements are modelled by finite lists. This would be useful in mathematical applications, but is not required for specifying and developing programs.

Furthermore, it would be very useful to integrate properties such as the Heyting algebra axioms into Coq's automation tactics so that additional properties can be proven more easily. Most of the proofs in our work consist of applying a small set of tactics to axioms and other auxiliary properties, and this seems to present an opportunity for a significant degree of automation.

Chapter 10

Appendix

The source code for the framework implemented in Coq can be downloaded at the following URL:
`http://www.cosc.brocku.ca/~mwinter/LRelationsInCoq.zip`

Bibliography

[1] Agda - Libraries and Other Developments,
Retrieved from http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Libraries.

[2] Agda Wiki, Retrieved from http://wiki.portal.chalmers.se/agda/pmwiki.php.

[3] Asperti, A., Longo, G.: Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist Foundations of Computing Series, The MIT Press (1991).

[4] Barendregt, H.P.: Introduction to generalized type systems. Journal of Functional Programming, 1(2):125-154, April 1991.

[5] Birkhoff G.: Lattice Theory. American Mathematical Society Colloquium Publications Vol. XXV, 3rd edition (1940).

[6] Burris, S., Sankappanavar, H.P.: A Course in Universal Algebra. S. Burris and H.P. Sankappanavar (2012). http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html

[7] Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press (2013).

[8] The Coq Proof Assistant, http://coq.inria.fr/.

[9] Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation, Vol. 76, Issue 2-3, 1988.

[10] Freyd, P.J., Scedrov, A.: Categories, Allegories. Mathematical Library Vol 39, North-Holland (1990).

[11] Furusawa, H., Kahl, W.: A Study on Symmetric Quotients Technical Report Nr. 1998-06, Universität der Bundeswehr München, 1998.

[12] Furusawa H., Kawahara Y., Winter M.: Dedekind Categories with Cutoff Operators. Fuzzy Sets and Systems 173, 1-24 (2011).

[13] Goguen J.A.: L-fuzzy sets. J. Math. Anal. Appl. 18 (1967), 145-157.

[14] Huet, G., Saïbi, A.: Constructive Category Theory. In: Proceedings of the Joint Clics-Types Workshop on Categories and Type Theory, MIT Press (1998).

[15] Hurkens, A. J. C.: A simplification of Girard's paradox. Typed Lambda Calculi and Applications. Lecture Notes in Computer Science Volume 902, 1995, pp 266-278.

[16] Isabelle Community Projects, Retreived from `https://isabelle.in.tum.de/community/Projects`.

[17] Jónsson B., Tarski A.: Boolean algebras with operators, I, II, Amer. J. Math. 73 (1951) 891-939, 74 (1952) 127-162

[18] Kahl W.: Towards Certifiable Implementation of Graph Transformation via Relation Categories. Relational and Algebraic Methods in Computer Science, LNCS 7560, 82-97 (2012).

[19] Katovsky A.: Category Theory. Archive of Formal Proofs, retreived from `http://afp.sf.net/entries/Category2.shtml`.

[20] Kawahara, Y., Furusawa H.: Crispness and Representation Theorems in Dedekind Categories. DOI-TR 143, Kyushu University (1997).

[21] Olivier J.P., Serrato D.: Catégories de Dedekind. Morphismes dans les Catégories de Schröder. C.R. Acad. Sci. Paris 290 (1980) 939-941.

[22] Olivier J.P., Serrato D.: Squares and Rectangles in Relational Categories - Three Cases: Semilattice, Distributive lattice and Boolean Non-unitary. Fuzzy sets and systems 72 (1995), 167-178.

[23] Rutherford, D.E.: Introduction to Lattice Theory. Oliver and Boyd. (1965)

[24] Schmidt G., Hattensperger C., Winter M.: Heterogeneous Relation Algebras. *In:* Brink C., Kahl W., Schmidt G. (eds.), Relational Methods in Computer Science, Advances in Computer Science, Springer Vienna (1997).

[25] Schmidt G., Ströhlein T.: Relationen und Graphen. Springer (1989); English version: Relations and Graphs. Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoret. Comput. Sci., Springer (1993)

[26] Schmidt G.: Relational Mathematics. Encyclopedia of Mathematics and Its Applications 132 (2011).

[27] Winter M.: A new Algebraic Approach to L-Fuzzy Relations Convenient to Study Crispness. INS Information Science 139, 233-252 (2001).

[28] Winter M.: Relational Constructions in Goguen Categories. Relational Methods in Computer Science (RelMiCS), LNCS 2561, 212-227 (2002).

[29] Winter M.: Derived Operations in Goguen Categories. TAC Theory and Applications of Categories 10(11), 220-247 (2002).

[30] Winter M.: Representation Theory of Goguen Categories. Fuzzy Sets and Systems 138, 85-126 (2003).

[31] Winter M.: Goguen Categories - A Categorical Approach to L-fuzzy relations. Trends in Logic 25, Springer (2007).

[32] Winter M.: Arrow Categories. Fuzzy Sets and Systems 160, 2893-2909 (2009).

[33] Winter M.: Membership Values in Arrow Categories. (submitted to Fuzzy Sets and Systems, October 2013).

[34] Winter M.: Higher-Order Arrow Categories. Relational and Algebraic Methods in Computer Science (RAMiCS), LNCS 8428 (2014).

[35] Winter M., Jackson E., Fujiwara Y.: Type-2 Fuzzy Controllers in Arrow Categories. Relational and Algebraic Methods in Computer Science (RAMiCS), LNCS 8428 (2014).

[36] Winter M., Kawahara Y.: Cardinality in Allegories. Relations and Kleene Algebra in Computer Science RelMiCS / AKA 2008. LNCS 4988, 275-290 (2008).

[37] Zadeh L.A.: The concept of a linguistic variable and its application to approximate reasoning - I. Information Sciences 8, 199-249 (1975).