An Extension of RATH to Explore and Apply Goguen Categories

Diplomarbeit von

Thomas Triebsees

UniBwM – ID 27/2002

Aufgabenstellung:

Prof. Dr. Gunther Schmidt

Betreuung:

Dr. Michael Winter

Universität der Bundeswehr München
Fakultät für Informatik
Institut für Softwaretechnologie

Neubiberg, den 02. Dezember 2002

Contents

Chapter 1

Introduction

Over the last years, dealing with unsharp information and uncertainty has become an every day issue in many areas of research and application. The well-known concept of fuzzy sets and fuzzy relations, introduced by Zadeh in 1965 ([4]), constitutes the base to handle this kind of information. The idea is, given an element of a fuzzy set/relation, to assign the degree of membership to this element. In this first approach, the degree of membership is a value taken out of the unit interval $[0, 1]$. Thus, a fuzzy set or fuzzy relation where the degree of membership takes only the value 0 or 1 can be seen as a conventional set/relation. It then is called to be $0 - 1$ crisp or simply crisp.
In 1967, Goguen generalized Zadeh's concept. He realized that it often can be useful not to express the degree of membership by an element of the unit interval. Thus, for example, $[0, 1] \times [0, 1]$ taking a tuple of two values out of $[0, 1]$ can be interesting for a customer to rate a product due to two quality criteria. In his paper [6], Goguen introduced the notions of $\mathcal{L}$-fuzzy sets and $\mathcal{L}$-fuzzy relations which use an arbitrary completely distributive lattice $\mathcal{L}$ to express membership within a fuzzy set/fuzzy relation.

In real world applications, especially the concept of fuzzy relations and $\mathcal{L}$-fuzzy relations is used to develop fuzzy controllers [5][7][8] for steering technical processes. Thus, for example, the air-conditioning of a building or certain safety systems within cars are regulated by such controllers. Especially the second application shows that it would be useful to have an exact theory to reason over $\mathcal{L}$-fuzzy relations and, thus, over fuzzy controllers. One then could prove certain properties of given fuzzy controllers and, hence, avoid an unexpected, safety critical behavior.
A first approach was given in [10] by H. Furusawa. He used the theory of relational categories

[9] and, especially, of Dedekind categories to model $\mathcal{L}$-fuzzy relations. Unfortunately, he got problems while expressing the notion of crispness within this theory. He gave two proposals (l-crispness and s-crispness) which coincide with $0-1$ crispness only under some restrictions on the entry lattice $\mathcal{L}$.

Finally, Michael Winter in [11] showed that the theory of Dedekind categories is too weak to express crispness in general. Hence, he introduced a new kind of relational category, the so-called Goguen category. By providing new operations, he gave a convenient notion of crispness and managed to show that the notions of l-crispness and s-crispness are subsumed by this new definition.

In [12] a proposal was given how to model fuzzy controllers within the theory of Goguen categories. This model constitutes the base to transfer the concepts of fuzzy control into Goguen categories. But, it would also be nice to have a computer-aided framework based on this model to construct fuzzy controllers and, just as important, to handle Goguen categories. This is the main issue of this thesis. We aim at a collection of Haskell [19][20] modules which allows a convenient handling of Goguen categories and, thus, of fuzzy controllers and $\mathcal{L}$-fuzzy relations.
There are already two Haskell libraries covering the treatment of relational categories and relations. The RELVIEW system developed at the Christian-Albrechts-University of Kiel allows to create and manipulate binary relations. The RATH system [17], developed at the University of the Federal Armed Forces Munich, goes a step further by providing a Haskell library to create, explore and test instances of relational categories. Hence, it is more suitable for our purposes than the RELVIEW system. We will focus on an extension of this module collection such that it covers Goguen categories.

In Chapter 2 we provide the mathematical background which is needed to understand this thesis. First, it includes a brief introduction to sets and relations. After that, fuzzy sets and fuzzy relations are introduced. Before we then introduce $\mathcal{L}$-fuzzy sets and $\mathcal{L}$-fuzzy relations, we provide a comprehensive overview over the lattice theoretical concepts. The chapter is concluded by an overview over relational categories and Goguen categories.

Chapter 3 then is dedicated to the extension of RATH. We first provide a module collection for lattices which is the necessary base for all other modules. After that we introduce some modules for Goguen categories matching all development criteria of RATH such that they can easily be integrated. With these modules, we especially provide some test routines to test instances of Goguen categories for correctness. Finally, a module for handling $\mathcal{L}$-fuzzy relations is implemented.

The last part of this thesis deals with fuzzy controllers. Chapter 4 is started by a comprehensive introduction to fuzzy controllers. After that, we introduce the relational model of [12], extend it at some points and introduce operations on fuzzy controllers. All these considerations are done on the abstract level of Goguen categories such that they can be deducted fully mathematical. The introduced operations then are the mathematical base for the module for fuzzy controllers. In this module we present suitable combinators to construct them and test their behavior. At the end of this thesis we develop an example controller which shows how the provided combinators can be used.

Chapter 2

Introduction of the Mathematical Concepts

In this chapter we want to introduce the mathematical concepts used throughout this thesis. We will often assume that the reader is familiar with the basic concepts and, therefore, give only a sketch of the topic.

2.1 Sets and relations

In the following we introduce the basic set theoretic concepts and notation. Although it can lead to contradictions, we will treat sets in a naive way. Due to space considerations, we omit all proofs. They can be found in the various literature.
The set membership of an element x is expressed by the symbol $\in$. We say $x \in M$ if x is an element of M and $x \notin M$ if it is not. The empty set $\emptyset$ has no elements. If a set N contains at least the elements of another set M we write $N \supseteq M$ ("N contains M") or $M \subseteq N$ ("M is a subset of N"). Notice that every set M can uniquely be described by its characteristic function $\chi_M : N \to \{0,1\}$ mapping an element x of an arbitrary set $N \supseteq M$ to one of the boolean values 0 and 1 as follows :

$$\chi_M(x) := \begin{cases} 1 & , if\ x \in M \\ 0 & , if\ x \notin M. \end{cases}$$

For the well known algebraic operations on sets we use the following denotation.

Definition 2.1.1. *Let M, N, and P be sets and $M \subseteq P$. Then we have*

(1) $M \cup N := \{x \mid x \in M \text{ or } x \in N\}$ **(union)**
(2) $M \cap N := \{x \mid x \in M \text{ and } x \in N\}$ **(intersection)**
(3) $\overline{M} := \{x \mid x \notin M \text{ and } x \in P\}$ **(complement)**.

Some interactions between and properties of these operations are summarized in the following lemma.

Lemma 2.1.1. *Let M, N, P and Q be sets such that $M, N \subseteq Q$. Then we have*

(1) $M \cup N = N \cup M$ and $M \cap N = N \cap M$, **(commutativity)**
(2) $(M \cup N) \cup P = M \cup (N \cup P)$ and $(M \cap N) \cap P = M \cap (N \cap P)$, **(associativity)**
(3) $(M \cup N) \cap P = (M \cap P) \cup (N \cap P)$, **(distributivity)**
(4) $\overline{\overline{M}} = M$,
(5) $M \cap N = \overline{\overline{M} \cup \overline{N}}$ and $M \cup N = \overline{\overline{M} \cap \overline{N}}$, **(de Morgan)**
(6) $M \subseteq N \;\Leftrightarrow\; \overline{N} \subseteq \overline{M}$.

The rule of de Morgan shown in (5) is essential because it shows how to obtain the union / intersection operation if one only has the complement and the intersection/union operators. We denote the infinitary variants of $\cap$ resp. $\cup$ by $\bigcap\limits_{i \in I} M_i$ resp. $\bigcup\limits_{i \in I} M_i$ for an index set I and arbitrary sets M_i.

A special set is the set of all subsets of an arbitrary set M. Using set comprehension, it can be expressed by

$\mathbb{P}(M) := \{X \mid X \subseteq M\}$ **(power set)**

Throughout this thesis we will need two basic constructions — direct sums and cartesian products. The direct sum $M + N$ and the cartesian product $M \times N$ of two sets M and N are defined as

$$M + N := \{(0, x) \mid x \in M\} \cup \{(1, y) \mid y \in N\}$$
$$M \times N := \{(x, y) \mid x \in M \text{ and } y \in N\}.$$

The direct sum $M + N$ obviously concatenates M and N. Notice that $M + N$ is isomorphic to $M \cup N$ for the case $M \cap N = \emptyset$. In contrast, the cartesian product gives us all tuples with the first component out of M and the second component out of N. Notice that n-ary cartesian products can be built by iterating the construction for two sets. But, in the following we focus on binary cartesian products.

In most cases not all pairs are of interest. Therefore, a mathematical characterization of the needed tuples is necessary. This leads us directly to the notion of relations. A binary relation $R : M \to N$ between two sets M and N is a subset of $M \times N$, i.e., R is an element of $\mathbb{P}(M \times N)$. We call R homogeneous if $M = N$ and, otherwise, heterogeneous. Throughout this thesis we will write xRy for the fact that $(x, y) \in R$ holds and $\neg xRy$ if it does not. Furthermore, the set $\{x \in M \mid \exists y \in N \ : \ xRy\}$ is called the *domain* ($dom(R)$) and the set $\{y \in N \mid \exists x \in M \ : \ xRy\}$ is called the *range* ($ran(R)$) of R. As relations are only special sets, the operations of Definition 2.1.1 are applicable and the properties shown in Lemma 2.1.1 hold.
We want to stress that a relation $R \subseteq M \times N$ in this manner can be represented as a $(|M| \times |N|)$ matrix indexed by elements of M and N. The entry (x,y) then takes the value 1 if xRy and 0 if $\neg xRy$. Hence, if we are dealing with the matrix representation, we write $R(x, y)$ for xRy resp. $\neg R(x, y)$ for $\neg xRy$. Later on, we will have to study relations that are not representable in the intuitive way shown here.
To emphasize that we are dealing with relations, we introduce a new notation and write $\sqcap/\sqcup$ instead of $\cap/\cup$ and $\sqsubseteq$ instead of $\subseteq$. The operation symbol $\overline{}$ stays unchanged. Furthermore, we use the special relations

$$\begin{array}{lll} \top\!\!\!\top_{M,N} & := M \times N, & \textbf{(full relation)} \\ \bot\!\!\!\bot_{M,N} & := \emptyset, & \textbf{(empty relation)} \\ \mathbb{I}_M & := \{(x, x) \mid x \in M\}. & \textbf{(identical relation)} \end{array}$$

Notice that $\bot\!\!\!\bot_{M,N} \sqsubseteq R$ and $R \sqsubseteq \top\!\!\!\top_{M,N}$ hold for every relation $R : M \to N$. The following definition gives us a formal base to describe certain kinds of relations.

Definition 2.1.2. *A relation $R : M \to N$ is called*

(1) **injective** $:\Leftrightarrow \forall x, y \in M, z \in N \ : \ xRz$ *and* $yRz \Rightarrow x = y$,
(2) **surjective** $:\Leftrightarrow \forall y \in N \exists x \in M \ : \ xRy$,
(3) **bijective** $:\Leftrightarrow$ *R injective and surjective,*
(4) **total** $:\Leftrightarrow \forall x \in M \exists y \in N \ : \ xRy$,
(5) **univalent** $:\Leftrightarrow \forall x \in M, y, z \in N \ : \ xRy$ *and* $xRz \Rightarrow y = z$,

and, if it is homogeneous, i.e., $M = N$,

(6) **reflexive** $:\Leftrightarrow \forall x \in M \ : \ xRx,$
(7) **irreflexive** $:\Leftrightarrow \forall x \in M \ : \ \neg xRx,$
(8) **transitive** $:\Leftrightarrow \forall x, y, z \in M \ : xRy$ *and* $yRz \Rightarrow xRz,$
(9) **symmetric** $:\Leftrightarrow \forall x, y \in M \ : \ xRy \Rightarrow yRx,$
(10) **antisymmetric** $:\Leftrightarrow \forall x, y \in M \ : \ xRy$ *and* $yRx \Rightarrow \ x = y,$
(11) **asymmetric** $:\Leftrightarrow \forall x, y \in M \ : \ xRy \Rightarrow \neg yRx.$

As usual, we call a univalent relation a function, a total function a mapping and a bijective mapping a bijection. Thus, a relation can be interpreted as a function $R : M \times N \rightarrow \{0, 1\}$. Notice that an n-ary cartesian product $M_1 \times ... \times M_n$ for an index set $I = \{1, .., n\}$ can be interpreted as the set of all mappings $f : I \rightarrow M_1 \cup M_2 \cup \cdots \cup M_n$ with $f(i) \in M_i$. As shown here, mappings and functions will be denoted by small letters like f.

Homogeneous relations characterized by (6)-(11) are of special interest in order theory and, thus, in lattice theory. Notice that every asymmetric relation is antisymmetric. The opposite is, in general, not true.

Definition 2.1.3. *A homogeneous relation R over a set M is called a*

(1) **preorder** $:\Leftrightarrow$ *R reflexive and transitive*
(2) **order** $:\Leftrightarrow$ *R reflexive, transitive and antisymmetric*
(3) **linear order** $:\Leftrightarrow$ *R is an order and $\forall x, y \in M \ : \ xRy$ or yRx*
(4) **strict order** $:\Leftrightarrow$ *R irreflexive and transitive*

Notice that we defined a strict order as an irreflexive and transitive homogeneous relation. This could equivalently be described by demanding the relation to be asymmetric and transitive.
Throughout this thesis we will call a tuple $(P, \leq)$ a poset (partially ordered set), iff P is a set with an order relation $\leq$ on it. Often we will identify the tuple $(P, \leq)$ only with P. Obviously, $\subseteq$ is an order relation and, thus, the power set $\mathbb{P}(M)$ of an arbitrary set M together with $\subseteq$ is a poset.

There are two essential operations that can (together with $\sqcup$, $\sqcap$, $\overline{}$) be used for an algebraic treatment of relations — conversion and composition.

Definition 2.1.4. *Let $R : M \rightarrow N$, and $S : N \rightarrow O$ be relations. Then we define*

(1) $R^{\smile} := \{(y, x) \mid (x, y) \in R\}$ **(conversion)**
(2) $R; S := \{(x, z) \mid \exists y \in N \ : \ (x, y) \in R$ *and* $(y, z) \in S\}$ **(composition)**

Obviously, our definition of ; implies that we read the composition $R;S$ from left to right, i.e., $R : M \rightarrow N$ and $S : N \rightarrow O$ implies $R;S : M \rightarrow O$. Notice that, in general, $dom(R) \neq dom(R;S)$ and $ran(S) \neq ran(R;S)$.
The following lemma summarizes some basic properties of conversion and composition.

Lemma 2.1.2. *Let $Q, R : M \rightarrow N$, and $S : N \rightarrow O$ be relations. Then we have*

(1) $(R^\smile)^\smile = R$,

(2) $(Q \sqcap R)^\smile = Q^\smile \sqcap R^\smile$ *and* $(Q \sqcup R)^\smile = Q^\smile \sqcup R^\smile$,

(3) $(R;S)^\smile = S^\smile;R^\smile$.

If $f : M \rightarrow N$ is a function or mapping we will denote the converse relation $f^\smile$ by f^{-1}. It is clear that $f;f^{-1} = \mathbb{I}_M$ if f is a bijection.
Now we are ready to express the characterizations of definition 2.1.2 in an algebraic manner.

Lemma 2.1.3. *Let $R : M \rightarrow N$ be a relation. Then we have*

$$\begin{array}{lllll}
(1) & R \text{ injective} & \Leftrightarrow R;R^\smile \sqsubseteq \mathbb{I}_M & \Leftrightarrow R^\smile \text{ univalent,} & \\
(2) & R \text{ surjective} & \Leftrightarrow \mathbb{I}_N \sqsubseteq R^\smile;R & \Leftrightarrow R^\smile \text{ total} & \Leftrightarrow \top\!\!\top_{M,N} \sqsubseteq \top\!\!\top_{M,M};R, \\
(3) & R \text{ total} & \Leftrightarrow \mathbb{I}_M \sqsubseteq R;R^\smile & \Leftrightarrow R^\smile \text{ surjective} & \Leftrightarrow \top\!\!\top_{M,N} \sqsubseteq R;\top\!\!\top_{N,N}, \\
(4) & \text{univalent} & \Leftrightarrow R^\smile;R \sqsubseteq \mathbb{I}_N & \Leftrightarrow R^\smile &
\end{array}$$

and if R is homogeneous,

$$\begin{array}{lllll}
(5) & R \text{ reflexive} & \Leftrightarrow \mathbb{I}_M \sqsubseteq R & \Leftrightarrow \mathbb{I}_M \sqsubseteq R^\smile, & \\
(6) & \text{irreflexive} & \Leftrightarrow R \sqsubseteq \overline{\mathbb{I}_M} & \Leftrightarrow \mathbb{I}_M \sqcap R \sqsubseteq \bot\!\!\bot_M & \Leftrightarrow \mathbb{I}_M \sqsubseteq \overline{R}, \\
(7) & \text{transitive} & \Leftrightarrow R^2 := R;R \sqsubseteq R & \Leftrightarrow R^\smile;\overline{R} \sqsubseteq \overline{R}, & \\
(8) & \text{symmetric} & \Leftrightarrow R \sqsubseteq R^\smile & \Leftrightarrow R = R^\smile & \Leftrightarrow \overline{R} \text{ symmetric,} \\
(9) & \text{antisymmetric} & \Leftrightarrow R \sqcap R^\smile \sqsubseteq \mathbb{I}_M & \Leftrightarrow \overline{\mathbb{I}_M} \sqsubseteq \overline{R} \sqcup \overline{R^\smile}, & \\
(10) & \text{asymmetric} & \Leftrightarrow R \sqcap R^\smile \sqsubseteq \bot\!\!\bot_{M,M} & \Leftrightarrow \top\!\!\top_{M,M} \sqsubseteq \overline{R} \sqcup \overline{R^\smile}. &
\end{array}$$

We will see some of these results again. They obviously constitute a component-free description of the underlying properties and, hence, are the basic characterizations within relational categories. This more common approach will be introduced later on.

Lemma 2.1.3 already gives a short impression of component less computation and characterization using relations. For a comprehensive introduction to relation algebras we refer to [15].

2.2 Fuzzy sets and fuzzy relations

In this section we want to give a short introduction to fuzzy sets. Fuzzy set theory is a wide ranged area of research. Therefore, we will only introduce the intent and the basic notions. Again we omit all proofs and exemplary refer to [5][4] and [7] for a comprehensive overview and mathematical background.

In standard set theory one can only express whether an element x is a member of a set M or it is not. But what if you have to express vague phrases like "warm", "cold", "big", "small" and so on ? Consider, for instance, the set of warm temperatures. With the approach shown in the last section, one would have to set an upper and a lower bound (e.g., $19°C$ and $27°C$) such that every temperature within these bounds is a member of the mentioned set. But, when $27°C$ is warm, why is $26,9°C$ not ? This motivates the following definition.

Definition 2.2.1. *We call a structure $F := (U, \gamma_F)$ with an arbitrary set U (universe) and a mapping $\gamma : U \to [0, 1]$ a* **fuzzy set over** *U. The application of γ_F to an element $x \in U$ is called* **fuzzification***.*

At this, $[0, 1]$ is the unit interval of all real numbers between 0 and 1. The mapping γ_F can, therefore, be interpreted as the degree of membership of a certain element $x \in U$ in F and, hence, we call it the membership function of F. We want to stress that there is an own theory for finding adequate membership functions. As this is not important for the general comprehension, we will not go into detail with it. Notice that $\gamma_F = \chi_F$, iff $ran(\gamma_F) = \{0, 1\}$. Of course, there are other possible ranges for the degree of membership than the unit interval. This is only a standard definition. Goguen, for example, in [6] introduced $\mathcal{L}$-fuzzy sets with γ taking values from a certain lattice $\mathcal{L}$. We will come back to this later.
Often we will identify the tuple (U,γ_F) only with F or with γ_F. Furthermore, we often use the induced set $\{(x, \gamma_F(x)) \mid x \in U\}$ whereas tuples $(x, 0)$ are not listed explicitly.

In order to describe properties of fuzzy sets and certain interactions to usual sets, we introduce some basic notions.

Definition 2.2.2. *Let F be a fuzzy set over U. Then we define*

(1) the **support** $supp(F) := \{(x, 1) | \ x \in U, \gamma_F(x) \neq 0\}$

(2) the α**-cut** $cut_\alpha(F) := \{(x, 1) | \ x \in U, \gamma_F(x) \geq \alpha\}$

(3) the **kernel** $ker(F) := \{(x, 1) | \ x \in U, \gamma_F(x) = 1\}$

(4) the **cokernel** $coker(F) := \{(x,1)|\ x \in U, \gamma_F(x) = 0\}$

(5) F *is* **crisp** $:\Leftrightarrow F = cut_1(F)$.

Obviously, F is a conventional set, iff it is crisp and, hence, γ_F is equal to χ_F. The α-cut will play an important role within Goguen categories, later on. But, already here it should be noticed that the α-cut for a given fuzzy set F induces an antitone function $\alpha \mapsto cut_\alpha(F)$, i.e., $\alpha \leq \alpha'$ implies $cut_{\alpha'}(F) \subseteq cut_\alpha(F)$ for all $\alpha, \alpha' \in [0,1]$ and all fuzzy sets F.

It is clear that $\cup$,$\cap$,$\overline{}$ and $\subseteq$ cannot be transferred from conventional sets to fuzzy sets without any adaptation. In fact, finding adequate definitions of $\cup$, $\cap$ and $\overline{}$ has costed remarkable efforts up to now. Therefore, we first define fuzzy set inclusion and then separately have a look at the union, intersection and negation operators for fuzzy sets.

Definition 2.2.3. *Let F and G be two fuzzy sets over U. Then*

$$F \subseteq G :\Leftrightarrow \forall x \in U : \gamma_F(x) \leq \gamma_G(x).$$

Notice that we use the same inclusion symbol $\subseteq$ for conventional and for fuzzy sets, respectively. It will be clear from the context which one is meant. Obviously, $\subseteq$ again is an order relation, i.e.,

$$\begin{array}{ll} F \subseteq G \text{ and } G \subseteq H \Rightarrow F \subseteq H, & \text{(transitive)} \\ F \subseteq G \text{ and } G \subseteq F \Rightarrow F = G, & \text{(antisymmetric)} \\ F \subseteq F. & \text{(reflexive)} \end{array}$$

Now, we want to focus on fuzzy set union and fuzzy set intersection. If we recall the definition of $M \cup N$ and $M \cap N$ for conventional sets, we see that they can equivalently be described using the characteristic functions χ_M resp. χ_N of M resp. N, and the boolean functions $and, or : \{0,1\} \times \{0,1\} \to \{0,1\}$ as follows :

$$\begin{array}{l} \chi_{M\cup N}(x) := \chi_M(x)\ or\ \chi_N(x) \\ \chi_{M\cap N}(x) := \chi_M(x)\ and\ \chi_N(x). \end{array}$$

Definition 2.2.2 marks crisp fuzzy sets as a special case of fuzzy sets. As they are equivalent to conventional sets, the operations $\cup$ and $\cap$ for crisp fuzzy sets have to be equivalent to union and intersection of conventional sets. In fuzzy set theory, we have the notion of t-norms resp. t-conorms (s-norms) as a synonym for fuzzy set intersection resp. fuzzy set union. The following definition gives us an axiomatic base.

Definition 2.2.4. *Let $\tau, \sigma : [0,1] \times [0,1] \to [0,1]$ be two mappings. Then we call τ a* **t-norm***, iff the following properties hold for all $x, y, z, x', y' \in [0,1]$:*

(1) $\tau(0,x) = 0$ *and* $\tau(1,x) = x$

(2) $x \leq x'$ *and* $y \leq y' \Rightarrow \tau(x,y) \leq \tau(x',y')$ *(monotonic)*

(3) $\tau(x,y) = \tau(y,x)$ *(commutative)*

(4) $\tau(x, \tau(y,z)) = \tau(\tau(x,y), z)$ *(associative)*

Furthermore we call σ a **t-conorm (s-norm)***, iff it satisfies the following for all x, y, z, x', y' $\in [0,1]$:*

(1) $\sigma(0,x) = x$ *and* $\sigma(1,x) = 1$

(2) $x \leq x'$ *and* $y \leq y' \Rightarrow \sigma(x,y) \leq \sigma(x',y')$ *(monotonic)*

(3) $\sigma(x,y) = \sigma(y,x)$ *(commutative)*

(4) $\sigma(x, \sigma(y,z)) = \sigma(\sigma(x,y), z)$ *(associative)*

Obviously, t-norms resp. t-conorms induce the algebraic structure of commutative semigroups with neutral element 1 resp. 0. Furthermore, we have

$$\begin{aligned} &\tau(1,0) = \tau(0,1) = \tau(0,0) = 0; \qquad &&\tau(1,1) = 1 \\ &\sigma(1,0) = \sigma(0,1) = \sigma(1,1) = 1; \qquad &&\sigma(0,0) = 0. \end{aligned}$$

Thus, τ/σ reduced to $\{0,1\}$ is equivalent to the boolean function *and*/*or*.

Now, we want to switch to possible negation operators in fuzzy theory. Again, we can express $\overline{M}$ for a conventional set M, using the boolean function *not* : $\{0,1\} \to \{0,1\}$ and the characteristic function χ_M, by

$$\chi_{\overline{M}}(x) := \textit{not}\ \chi_M.$$

Thus, we give the following definition.

Definition 2.2.5. *Let $\nu : [0,1] \to [0,1]$ be a mapping. Then we call ν a* **fuzzy negation***, iff the following properties are satisfied for all $x, y \in [0,1]$:*

(1) $\nu(0) \quad = 1$ *and* $\nu(1) = 0$

(2) $x \leq y \Rightarrow \nu(y) \leq \nu(x)$ *(antitonic)*

Restriction (1) explicitly demands that ν reduced to $\{0, 1\}$ is equivalent to the boolean function *not*. The antitonic behavior demanded in (2) is the typical characterization of a negation operator and, therefore, obviously necessary.

One can imagine that Definitions 2.2.4 and 2.2.5 enclose a big class of possible operations. In the following we want to give some widely used examples for fuzzy set union, intersection and negation. Notice that this is not a complete collection at all and, therefore, we refer to the already cited literature for a more comprehensive overview.

Lemma 2.2.1.
The following operators are t-norms

(1) $\tau_m(x, y) := min(x, y)$ **(minimum conjunction)**
(2) $\tau_b(x, y) := max(0, x + y - 1)$ **(bold conjunction)**
(3) $\tau_p(x, y) := x \cdot y$ **(product conjunction)**
(4) $\tau_d(x, y) := \begin{cases} 1 \text{ , } if\ x = y = 1 \\ 0 \text{ , } otherwise \end{cases}$ **(drastic conjunction)**.

The following operators are t-conorms

(1) $\sigma_m(x, y) := max(x, y)$ **(maximum disjunction)**
(2) $\sigma_b(x, y) := min(1, x + y)$ **(bold disjunction)**
(3) $\sigma_p(x, y) := x + y - x \cdot y$ **(product disjunction)**
(4) $\sigma_d(x, y) := \begin{cases} 0 \text{ , } if\ x = y = 0 \\ 1 \text{ , } otherwise \end{cases}$ **(drastic conjunction)**.

The following operators are negations

(1) $\nu_l(x) := 1 - x$ **(Łukasiewicz negation)**
(4) $\nu_d(x) := \begin{cases} 1 \text{ , } if\ x = 0 \\ 0 \text{ , } otherwise \end{cases}$ **(drastic negation)**.

The operations τ_m, σ_m and ν_l are the standard pendant to *and*, *or*, and *not*. We want to mention that τ_m is the greatest t-norm, i.e., $\tau_m(x, y) \geq \tau'(x, y)$ for all other t-norms τ' and all $x, y \in U$. Furthermore, σ_m is the least t-conorm, i.e., $\sigma_m(x, y) \leq \sigma'(x, y)$ for all other t-conorms σ' and all $x, y \in U$. We want to admit that the question which of the possible t-norms/conorms resp. negations to choose strongly depends on the application.

According to Definition 2.2.4, the union and intersection of two fuzzy sets F and G over U

resp. the negation of F can be computed as follows using a given t-norm τ, t-conorm σ and negation ν :

$$\begin{array}{llll} F \cap G & := (U, \gamma_{F\cap G}) & , \gamma_{F\cap G}(x) & := \tau(\gamma_F(x), \gamma_G(x)) \\ F \cup G & := (U, \gamma_{F\cup G}) & , \gamma_{F\cup G}(x) & := \sigma(\gamma_F(x), \gamma_G(x)) \\ \overline{F} & := (U, \gamma_{\overline{F}}) & , \gamma_{\overline{F}}(x) & := \nu(\gamma_F(x)). \end{array}$$

Last but not least, we want to present an important result. To do so, we introduce an abbreviated notation. If we have a crisp fuzzy set M over U, we may obtain a new fuzzy set $\alpha \cdot M$ by

$$(\alpha \cdot M)(x) := (x, \alpha \cdot \chi_M(x)) \qquad .$$

Since the α-cut of a given fuzzy set F is crisp, we may represent F as shown in the following theorem.

Theorem 2.2.1 (**α-cut Theorem for Fuzzy Sets**). *Let F be a fuzzy set over U. Furthermore, let $\cup$ be the fuzzy union operator induced by σ_m (max disjunction).Then we have*

$$F = \bigcup_{\alpha \in [0,1]} \alpha \cdot cut_\alpha(F).$$

We will see this result again in different variants since it is essential for the representation of Goguen categories introduced in Section 2.6.

The extension of fuzzy sets to fuzzy relations is done quite intuitively. A fuzzy relation R from U to V is a fuzzy set over $U \times V$. This implies that γ_R is a mapping $\gamma_R : U \times V \to [0,1]$. In the following we will identify γ_R with R. Again, this definition induces a matrix representation whereas an entry $R(x,y)$ can be interpreted as the degree to which the tuple (x,y) is in R. We do not yet want to go into detail with fuzzy relations. Later on, we introduce $\mathcal{L}$-fuzzy relations and examine certain properties.

2.3 Lattices

In this section we want to give a short introduction to the lattice theoretical concepts used throughout this thesis. Since we will have to provide a Haskell module for lattices, this introduction is held pretty comprehensive. Nevertheless, we omit all proofs due to space considerations. For further reading we refer to [1][3][2] and [12], for example. We start by defining lattices and outlining some basic results.

2.3.1 Definition and some first properties

Lattices are structures that allow us to compute in a more abstract way than, for example, within the real numbers ($\mathbb{R}$) or the natural numbers ($\mathbb{N}$). The main difference is that we are not necessarily dealing with linear orders. We first introduce the following operations.

Definition 2.3.1. *Let P be a poset and $x, y \in P$. Then we define*

(1) $x \wedge y := inf(x,y) :=$ *z with $z \leq x$ and $z \leq y$ and for all z' with $z' \leq x$ and $z' \leq y$ we have $z' \leq z$* **(meet)**

(2) $x \vee y := sup(x,y) :=$ *z with $z \geq x$ and $z \geq y$ and for all z' with $z' \geq x$ and $z' \geq y$ we have $z' \geq z$* **(join)**

The extension of (1)/(2) to sets of elements we denote by $\bigwedge$ / $\bigvee$. Notice that $\wedge$, $\vee$, $\bigwedge$ and $\bigvee$ are, in general, partial operations, i.e., $inf(X)$ and $sup(X)$ do not necessarily exist in P.

With these preparations, we are now ready to define lattices.

Definition 2.3.2. *A structure $\mathcal{L}$ is called*

(1) a **lower semilattice** $:\Leftrightarrow$ *$\mathcal{L}$ is a poset and closed under meet*

(2) an **upper semilattice** $:\Leftrightarrow$ *$\mathcal{L}$ is a poset and closed under join*

(3) a **complete** *lower/upper semilattice* $:\Leftrightarrow$ *$\bigwedge M / \bigvee M$ exist in $\mathcal{L}$ for all subsets $M \neq \emptyset$ of $\mathcal{L}$*

(4) a **lattice** $:\Leftrightarrow$ *$\mathcal{L}$ is an upper and a lower semilattice.*

For a lattice $\mathcal{L}$ we will denote the corresponding upper/lower semilattice by $\mathcal{L}^u/\mathcal{L}_l$. If we speak of either upper or lower semilattices, we will call them semilattices. If the underlying order is linear, we call the (semi)lattice linear. Notice that every finite (semi)lattice is complete because every subset M of $\mathcal{L}$ is finite and therefore $\bigwedge M$, respectively $\bigvee M$, can be computed by iterating the definition of meet and join. Furthermore, complete upper/lower semilattices (and therefore all finite upper/lower semilattices) have the nice property shown in the next lemma.

Lemma 2.3.1. *Let $\mathcal{L}$ be a complete lower/upper semilattice. Then $\mathcal{L}$ has a least/greatest element $0/1$. Furthermore, we have*

(1) $\mathcal{L}$ is an upper semilattice $\Rightarrow 1 = \bigvee \mathcal{L}$,

(2) $\mathcal{L}$ is a lower semilattice $\Rightarrow 0 = \bigwedge \mathcal{L}$.

In the following we want to show that it suffices to demand that either the lower or the upper semilattice of a lattice $\mathcal{L}$ is complete to derive the completeness of $\mathcal{L}$. We start with the following lemma.

Lemma 2.3.2. *Let $\mathcal{L}$ be a lower/upper semilattice. Then the set of upper/lower bounds $\widehat{M}$ to a set $M \subseteq \mathcal{L}$ is again a complete lower/upper semilattice with least/greatest element $\bigvee M / \bigwedge M$.*

Now, we can connect completeness of upper and lower semilattices.

Theorem 2.3.1. *Let $\mathcal{L}$ be a lattice. Then we have*

$$\mathcal{L}_l \text{ is complete} \Leftrightarrow \mathcal{L}^u \text{ is complete.}$$

Obviously, Theorem 2.3.1 is a direct conclusion from Lemma 2.3.2. Hence, we have motivated the definition of complete lattices.

Definition 2.3.3.
A lattice $\mathcal{L}$ is called **complete** *iff $\mathcal{L}_l$ or $\mathcal{L}^u$ is complete.*

Notice that the least resp. greatest element in a complete lattice $\mathcal{L}$ can be computed by $0 = \bigwedge \mathcal{L} = \bigvee \emptyset$ and $1 = \bigwedge \emptyset = \bigvee \mathcal{L}$.
Lemma 2.3.2 indicates that in an upper/lower semilattice $\mathcal{L}$ the set of lower/upper bounds to a set $M \subseteq \mathcal{L}$ is somehow embedded into $\mathcal{L}$. This leads us to the notion of a sub(semi)lattice.

Definition 2.3.4. *Let $\mathcal{L}$ be a lattice. We call a subset $\mathcal{L}' \subseteq \mathcal{L}$*

(1) an upper subsemilattice of $\mathcal{L}$ $:\Leftrightarrow$ $\mathcal{L}'$ is closed under join,

(2) a lower subsemilattice of $\mathcal{L}$ $:\Leftrightarrow$ $\mathcal{L}'$ is closed under meet,

(3) a sublattice of $\mathcal{L}$ $:\Leftrightarrow$ $\mathcal{L}'$ is closed under join and meet.

Again, if we refer to either upper or lower subsemilattices, we will call them just subsemilattices.

With Definitions 2.3.2, 2.3.3 and 2.3.4 we have the basic notions of lattice theory. Two standard lattices are the unit interval $[0, 1] \subseteq \mathbb{R}$ together with $\leq$ and the powerset $\mathbb{P}(M)$ of an arbitrary set M together with $\subseteq$. We want to stress that there is another (algebraic) approach to lattices. Consider the following theorem.

Theorem 2.3.2. *A structure $\mathcal{L} := (L, \wedge, \vee)$ with a* **carrier set** *L and two operations $\wedge, \vee : L \times L \to L$ is a lattice, iff the following properties hold for all $x, y, z \in L$*

(1) $(x \vee y) \vee z = x \vee (y \vee z)$ *and* $(x \wedge y) \wedge z = x \wedge (y \wedge z)$, *(associativity)*
(2) $x \vee y = y \vee x$ *and* $x \wedge y = y \wedge x$, *(commutativity)*
(3) $(x \vee y) \wedge x = x$ *and* $(x \wedge y) \vee x = x$. *(absorption)*

Notice that L and $\vee$ resp. L and $\wedge$ induce the related upper resp. lower, semilattice. The equivalence of the last theorem makes clear that, from a mathematical point of view, it plays no role whether lattices are defined algebraically or via posets and sup/inf. But, we want to provide a Haskell module for lattices later on. There we will have to consider these two possibilities.

2.3.2 Order and lattice morphisms, Galois connections

In this section we want to focus on homomorphisms between structured sets (e.g., orders and lattices). Hence, we have a mathematical concept to construct and compare them.

Definition 2.3.5. *Let P_1 and P_2 be posets and $f : P_1 \to P_2$ be a mapping. Furthermore let $\mathcal{L}_1$ and $\mathcal{L}_2$ be lattices and $g : \mathcal{L}_1 \to \mathcal{L}_2$ be a mapping. Then we call f*

(1) **monotonic** $:\Leftrightarrow x \leq y \Rightarrow f(x) \leq f(y)$
(2) **antitionic** $:\Leftrightarrow x \leq y \Rightarrow f(y) \leq f(x)$
(3) an **order homomorphism** $:\Leftrightarrow$ *f is monotonic*
(4) an **order isomorphism** $:\Leftrightarrow$ *f is bijective and f and f^{-1} are order homomorphisms*

and g

(5) a **lower semilattice homomorphism** $:\Leftrightarrow g(x \wedge y) = g(x) \wedge g(y)$
(6) a **lower co-semilattice homomorphism** $:\Leftrightarrow g(x \wedge y) = g(x) \vee g(y)$
(7) an **upper semilattice homomorphism** $:\Leftrightarrow g(x \vee y) = g(x) \vee g(y)$
(8) an **upper co-semilattice homomorphism** $:\Leftrightarrow g(x \vee y) = g(x) \wedge g(y)$
(9) a **lattice homomorphism** $:\Leftrightarrow$ *g is a lower and an upper semilattice homomorphism*
(10) a **co-lattice homomorphism** $:\Leftrightarrow$ *g is a lower and an upper co-semilattice homomorphism*
(11) a **(semi)lattice isomorphism** $:\Leftrightarrow$ *g is bijective and g and g^{-1} are (semi)lattice homomorphisms.*

Furthermore, g is called **complete**, *iff the underlying property can be extended to arbitrary subsets $M \subseteq \mathcal{L}_1$.*

From (1) and (3) one can see that the order homomorphisms between two posets P_1 and P_2 are just the monotone mappings, which we will denote by $P_1 \overset{mon}{\to} P_2$. Furthermore, we denote all antitone mappings between P_1 and P_2 by $P_1 \overset{anti}{\to} P_2$. If we refer to the monotone as well as to the antitone mappings we write $P_1 \overset{*}{\to} P_2$.
If we speak of either lower or upper semilattice homomorphisms we will call them semilattice homomorphisms. Notice that every (semi)lattice homomorphism is monotonic, i.e., an order homomorphism. The other direction is, in general, not true. If there is an order/lattice isomorphism between two orders/lattices, we call them isomorphic. Notice that we have $g(1) = g(\bigwedge \emptyset) = \bigwedge \emptyset = 1$ resp. $g(0) = g(\bigvee \emptyset) = \bigvee \emptyset = 0$ for complete lower resp. upper semilattice homomorphisms. Analogously, we have $g(1) = g(\bigwedge \emptyset) = \bigvee \emptyset = 0$ resp. $g(0) = g(\bigvee \emptyset) = \bigwedge \emptyset = 1$ for complete lower resp. upper co-semilattice homomorphisms. Later on, complete upper co-semilattice homomorphisms will play an important role. Therefore, we call them just *antimorphisms* in analogy to [11].

Unfortunately, not every bijective order homomorphism is an order isomorphism. Consider the example in Figure 2.1 with two posets $P_1 = \{a, b, c, d\}$ and $P_2 = \{a', b', c', d'\}$ and a bijective order homomorphism $f : P_1 \to P_2$ between them.

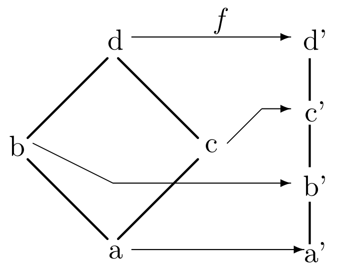

Figure 2.1: Bijective order homomorphism, but no isomorphism

Obviously, f^{-1} is no order homomorphism because $b' \leq c'$ but $f^{-1}(b') \not\leq f^{-1}(c')$. Thus, f is no order isomorphism. It is clear that this behavior cannot occur if P_1 and P_2 are linear. Hence, every bijective order homomorphism between two linear posets is an order isomorphism.

Now the question arises in which case a bijective (semi)lattice homomorphism is a (semi)lattice isomorphism.

Theorem 2.3.3. *Every bijective (semi) lattice homorphism $f : \mathcal{L}_1 \to \mathcal{L}_2$ is a (semi) lattice isomorphism.*

The next theorem handles composition of (upper/lower semi)lattice homomorphisms.

Theorem 2.3.4. *Let both $f : \mathcal{L}_1 \to \mathcal{L}_2$ and $g : \mathcal{L}_2 \to \mathcal{L}_3$ be (upper/lower semi)lattice homomorphisms. Then $h := f; g$ is an (upper/lower semi)lattice homomorphism between $\mathcal{L}_1$ and $\mathcal{L}_3$.*

Through the notion of (semi)lattice homomorphisms we now have a new (more algebraic) possibility to describe sub(semi)lattices (cf. Definition 2.3.4).

Lemma 2.3.3. *Let $\mathcal{L}$ be a (semi)lattice. A subset $\mathcal{L}'$ of $\mathcal{L}$ is a sub(semi)lattice, if and only if there is an injective (semi)lattice homomorphism $f : \mathcal{L}' \to \mathcal{L}$.*

Finally, we want to examine a special construction dealing with pairs of functions between posets. It was initialized by Galois and shows its whole strength if the underlying posets are even lattices.

Definition 2.3.6. *Let P_1 and P_2 be posets. Then a pair (f, g) of functions $f : P_1 \to P_2$ and $g : P_2 \to P_1$ is called a*

(1) **monotone Galois connection**, *iff $f(x) \leq y \;\Leftrightarrow\; x \leq g(y)$*
(2) **antitone Galois connection**, *iff $y \leq f(x) \;\Leftrightarrow\; x \leq g(y)$*

holds for all $x \in P_1$ and $y \in P_2$

If we refer to either monotone or antitone Galois connections, we just speak of Galois connections. The notions of monotone and antitone are motivated by the following lemma. Furthermore, some basic properties are listed.

Lemma 2.3.4. *Let P_1 and P_2 be posets and $f : P_1 \to P_2$ and $g : P_2 \to P_1$ functions. Then we have :*

(1) If (f, g) constitutes a monotone Galois connection, then f and g are monotonic.
(2) If (f, g) constitutes an antitone Galois connection, then f and g are antitonic.
Furthermore, if (f, g) is a Galois connection, then
(3) $x \leq g(f(x))$ and $y \leq f(g(y))$,
(4) $f(x) = f(g(f(x)))$ and $g(y) = g(f(g(y)))$

hold for all $x \in P_1$ and $y \in P_2$.

We do not want to go into detail with this construction since we do not need more than the provided results so far.

2.3.3 Special elements

Now, we want to introduce the definitions of special elements within certain lattices. As we will see, they often play an important role since they may reflect the structure of the underlying lattice. We start with irreducible elements.

Definition 2.3.7. *Let $\mathcal{L}$ be a lattice and $x \in \mathcal{L}$. Then we define :*

(1) If $\mathcal{L}$ has a least element 0 then $x \neq 0$ is **join-irreducible** $:\Leftrightarrow$
$y \vee z = x$ implies $x = y$ or $x = z$ for all $y, z \in \mathcal{L}$.

(2) If $\mathcal{L}$ has a greatest element 1 then $x \neq 1$ is **meet-irreducible** $:\Leftrightarrow$
$y \wedge z = x$ implies $x = y$ or $x = z$ for all $y, z \in \mathcal{L}$.

Obviously, every element of a linear order is join- and meet-irreducible. The join irreducible elements of the divisibility lattice with *gcd* and *lcm* are the prime numbers. The first example shows that irreducible elements can be in order relationship to each other. The second one makes clear that they can be used to compute all other elements of the underlying lattice under some circumstances.
In the following we provide a convenient theorem covering these circumstances for lattices in which all ascending/descending chains are finite. Ascending/descending chains are tuples of elements $x_1, ..., x_n \in \mathcal{L}$ such that $x_1 < x_2 < ... < x_n$ resp. $x_1 > x_2 > ... > x_n$ holds.

Theorem 2.3.5. *Let $\mathcal{L}$ be a lattice. We denote the set of join resp. meet irreducible elements by M_j resp. M_m. Then we have :*

(1) If $\mathcal{L}$ has a 0 and all descending chains are finite, then $x = \bigvee M'_j$ for all $x \in \mathcal{L}$ and a certain $M'_j \subseteq M_j$.

(2) If $\mathcal{L}$ has a 1 and all ascending chains are finite, then $x = \bigwedge M'_m$ for all $x \in \mathcal{L}$ and a certain $M'_m \subseteq M_m$.

Theorem 2.3.5 obviously gives sufficient conditions such that all elements of $\mathcal{L}$ can be partitioned via $\wedge$ resp. $\vee$. We immediately conclude that this result is applicable for every finite lattice. But, if we again consider linear orders, we see that this partition need not necessarily be unique. Later on, we will see when this property is true.

A slightly stronger notion is introduced with the following definition.

Definition 2.3.8. *Let $\mathcal{L}$ be a lattice with least element 0. Then we define :*

An element $a \neq 0$ is an **atom** $:\Leftrightarrow$ *$a \wedge x = a$ or $a \wedge x = 0$ holds for all $x \in \mathcal{L}$.*

Furthermore, $\mathcal{L}$ is called **atomic** *iff all elements $0 \neq x \in \mathcal{L}$ can be computed by*

$$\bigvee\{a \in \mathcal{L} \mid a \text{ atom},\ a \leq x\}.$$

It is easy to see that every atom is join-irreducible. The opposite is, in general, not true. This can already be seen from the fact that two different atoms cannot be in an order relationship, i.e., $a_1 \leq a_2$ implies $a_1 = a_2$. Atoms play an important role within Boolean algebras as we will see later on.

Finally, we want to introduce linear elements.

Definition 2.3.9. *Let $\mathcal{L}$ be a lattice. Then we define :*

An element $x \in \mathcal{L}$ is called **linear** $:\Leftrightarrow$ *$x \wedge y = 0$ implies $y = 0$ for all $y \in \mathcal{L}$.*

Notice that the fact that all elements of $\mathcal{L}$ are linear does not imply that $\mathcal{L}$ is linear, i.e., $x \leq y$ or $y \leq x$ holds for all $x, y \in \mathcal{L}$. Linear elements will play a role for the algebraic treatment of crisp $\mathcal{L}$-fuzzy relations, later on.

2.3.4 Modular lattices

One can imagine that the lattice definitions given in Section 2.3.1 describe a very huge class of structures which, in general, does not even guarantee distributivity. Therefore, we now introduce some restricted lattices that provide a more comfortable algebraic environment.

Obviously, the weak modular laws

$$a \leq c \Rightarrow a \vee (b \wedge c) \leq (a \vee b) \wedge c,$$
$$a \leq c \Rightarrow c \wedge (b \vee a) \geq (c \wedge b) \vee a$$

hold in every lattice. We even have the following.

Lemma 2.3.5. *Let $\mathcal{L}$ be a lattice and $a, b, c \in \mathcal{L}$. Then we have*

$$a \leq c \;\Rightarrow\; a \vee (b \wedge c) = (a \vee b) \wedge c \quad \textit{iff} \quad a \leq c \;\Rightarrow\; c \wedge (b \vee a) = (c \wedge b) \vee a.$$

This motivates the following definition.

Definition 2.3.10. *A lattice $\mathcal{L}$ is called* **modular***, if one of the modular laws holds for all $a, b, c \in \mathcal{L}$.*

To see the effect of this restriction, we give an example of a non modular lattice in Figure 2.2.

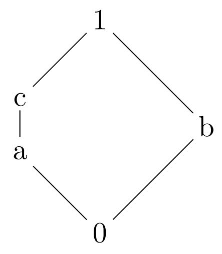

Figure 2.2: Non modular lattice $\mathcal{L}_{NonMod}$

Obviously, we have $a = a \vee (b \wedge c) < (a \vee b) \wedge c = c$. It can be shown that there is an injective lattice homomorphism f from $\mathcal{L}_{NonMod}$ to every non modular lattice $\mathcal{L}$. Thus, $\mathcal{L}_{NonMod}$ can be seen as *the* non modular lattice. The characteristic mark of $\mathcal{L}_{NonMod}$ is that it has no uniform stratification. We, for example, have two pathes of different length from 0 to 1. This observation is formally described in the next theorem.

Theorem 2.3.6 (Transposition principle of R. Dedekind)**.** *Let $\mathcal{L}$ be a modular lattice and $a, b \in \mathcal{L}$. Then the mappings*

$$\begin{aligned} f &: \{x \in \mathcal{L} | b \leq x \leq a \vee b\} \to \{x \in \mathcal{L} | a \wedge b \leq x \leq a\}, \; f(x) = x \wedge a \\ g &: \{x \in \mathcal{L} | a \wedge b \leq x \leq a\} \to \{x \in \mathcal{L} | b \leq x \leq a \vee b\}, \; g(y) = y \vee b \end{aligned}$$

are lattice isomorphisms such that $g = f^{-1}$ and $f = g^{-1}$.

Using this theorem, we are now ready to describe equality of two elements by means of $\vee$ and $\wedge$.

Corollary 2.3.1. *Let $\mathcal{L}$ be a modular lattice and $a, b, c \in \mathcal{L}$. Then we have*

$$a \leq c, a \vee b = c \vee b, a \wedge b = c \wedge b \;\Rightarrow\; a = c.$$

Obviously, the resulting equality of a and c comes due to the fact that f and g in Theorem 2.3.6 are bijective.

At the end of this section we want to admit that the modularity property is inherited, i.e., every sublattice of a modular lattice is modular.
The standard lattices $[0, 1]$ and $\mathbb{P}(M)$ are modular. Furthermore, every linear lattice is modular.

2.3.5 Distributive lattices

In Corollary 2.3.1 we saw that in a modular lattice not every element can be uniquely determined by $\vee$ and $\wedge$ with respect to an element $b \in \mathcal{L}$. We needed the restriction $a \leq c$ to conclude $a = c$. Consider the lattice in Figure 2.3.

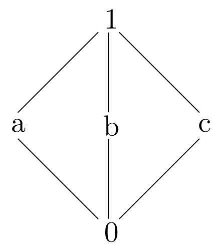

Figure 2.3: Modular, but non distributive lattice $\mathcal{L}_{NonDis}$

Obviously, $\mathcal{L}_{NonDis}$ is modular. But, $1 = a \vee b = c \vee b$ and $0 = a \wedge b = c \wedge b$ do not imply $a = c$. This shows that the restriction $a \leq c$ is essential in modular lattices. Later on, we need lattices in which certain elements are unique with respect to another element. These elements are defined by means of $\vee$ and $\wedge$. Thus, we aim to define lattices in which the restriction $a \leq c$ is no longer necessary. This can be achieved by demanding distributivity.

It is easy to see that the weak distributive laws

$$a \vee (b \wedge c) \leq (a \vee b) \wedge (a \vee c)$$
$$a \wedge (b \vee c) \geq (a \wedge b) \vee (a \wedge c)$$

are valid in every lattice. But, analogous to modular lattices, we even have the following result.

Lemma 2.3.6. *Let $\mathcal{L}$ be a lattice and $a, b, c \in \mathcal{L}$. Then we have*

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \quad \textit{iff} \quad a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c).$$

This motivates the following definition.

Definition 2.3.11. *A lattice $\mathcal{L}$ is called* **distributive***, if one of the distributive laws holds for all $a, b, c \in \mathcal{L}$.*

Hence, we have a class of lattices where an element can uniquely be determined by means of $\vee$ and $\wedge$.

Theorem 2.3.7. *Let $\mathcal{L}$ be a distributive lattice and $a, b, c \in \mathcal{L}$. Then we have*

$$a \vee b = c \vee b, a \wedge b = c \wedge b \;\Rightarrow\; a = c.$$

Furthermore, we have the following.

Lemma 2.3.7. *Let $\mathcal{L}$ be a distributive lattice with least/greatest element $0/1$. Then the partition of an element $x \in \mathcal{L}$ via the join/meet irreducible elements is unique if it exists.*

The existential part of the last theorem is (according to Theorem 2.3.5) fulfilled if all descending/ascending chains are finite.
Notice that the lattice shown in Figure 2.3 can be seen as *the* modular but non distributive lattice since there is an injective lattice homomorphism $f : \mathcal{L}_{NonDis} \to \mathcal{L}$ from $\mathcal{L}_{NonDis}$ to any modular and non distributive lattice $\mathcal{L}$.
Examples for distributive lattices are again the unit interval $[0, 1]$ and the powerset $\mathbb{P}(M)$ of an arbitrary set M. As a non standard distributive lattice we mention the set of all fuzzy sets over a universe U (i.e., the set of all mappings $\gamma : U \to [0, 1]$) together with one of the possible t-resp. t-conorms as meet resp. join.

2.3.6 Brouwerian lattices

Brouwerian lattices play an important role in the category theoretical approaches to relational structures. Before we can define them, some preparations have to be done.
First of all, distributivity can be extended to arbitrary subsets M of complete lattices. This leads us to the notion of completely upwards/downwards distributive lattices which will play an important role later on.

Definition 2.3.12. *Let $\mathcal{L}$ be a complete distributive lattice. Then it is called*

(1) **completely upwards distributive** $:\Leftrightarrow x \wedge \bigvee M = \bigvee_{y \in M} (x \wedge y)$ *for all $x \in \mathcal{L}$ and $M \subseteq \mathcal{L}$*

(2) **completely downwards distributive** $:\Leftrightarrow x \vee \bigwedge M = \bigwedge_{y \in M} (x \vee y)$

for all $x \in \mathcal{L}$ and $M \subseteq \mathcal{L}$

(3) **completely distributive** $:\Leftrightarrow$ *it is completely upwards and downwards distributive*

Unfortunately, complete upwards distributivity does, in general, not imply complete downwards distributivity and vice versa. Thus, complete distributivity has to be demanded explicitly if necessary. But, we have the following result for finite lattices.

Lemma 2.3.8. *Let $\mathcal{L}$ be a finite lattice. Then $\mathcal{L}$ is completely upwards distributive iff it is completely downwards distributive.*

This lemma is important since we are only able to deal with finite lattices within our Haskell module. The last lemma together with Lemma 2.3.6 implies that testing upwards distributivity suffices to conclude complete upwards distributivity and, hence, complete downwards distributivity within finite lattices.

Now, we need the notion of relative pseudo complements.

Definition 2.3.13. *Let $\mathcal{L}$ be a lattice and $x, y \in \mathcal{L}$. An element $x : y \in \mathcal{L}$ is called a* **relative pseudo complement** *of x in y, if and only if the equivalence*

$$a \leq x : y \;\Leftrightarrow\; x \wedge a \leq y$$

is valid for all $a \in \mathcal{L}$.

Obviously, the equivalence within the last definition implies that relative pseudo complements are unique if they exist. Hence, we can speak of *the* relative pseudo complement.
If we again have a look at the lattice $\mathcal{L}_{NonDis}$ in Figure 2.3, we see that $a : 0$ does not exist in $\mathcal{L}_{NonDis}$. Seemingly, the non distributivity is to blame. This motivates the following theorem in which we get an even stronger result.

Theorem 2.3.8. *Let $\mathcal{L}$ be a complete lattice. Then the relative pseudo complement $x : y$ exists for all $x, y \in \mathcal{L}$ iff $\mathcal{L}$ is completely upwards distributive.*

Thus, we can define Brouwerian lattices as follows.

Definition 2.3.14. *A lattice $\mathcal{L}$ is called a* **Brouwerian lattice***, if the pseudo complement $x : y$ exists for all $x, y \in \mathcal{L}$. Furthermore, $\mathcal{L}$ is a complete Brouwerian lattice iff it is completely upwards distributive.*

A homomorphism between Brouwerian lattices is a lattice homomorphism preserving relative pseudo complements. Obviously, all finite and distributive lattices are Brouwerian lattices.

2.3.7 Boolean lattices

The last section was based on the notion of relative pseudo complements. Now, we want to introduce an even stronger variant which leads to Boolean lattices.

Definition 2.3.15. *Let $\mathcal{L}$ be a lattice with least element* 0 *and greatest element* 1*. Furthermore, let $x \in \mathcal{L}$ be an element. Then we call $\overline{x}$ the complement of x, iff*

(1) $x \vee \overline{x} = 1$ *and*

(2) $x \wedge \overline{x} = 0$

holds.

Notice that complements are not necessarily unique. This can already be seen in Figure 2.3 where both b and c are complements to a. But fortunately, we have the following.

Lemma 2.3.9. *Let $\mathcal{L}$ be a distributive lattice with least element* 0 *and greatest element* 1*. Furthermore, let $x, \overline{x}^1, \overline{x}^2$ elements of $\mathcal{L}$ such that both $\overline{x}^1$ and $\overline{x}^2$ are complements to x. Then $\overline{x}^1 = \overline{x}^2$.*

Together with Definition 2.3.15, we see that the last lemma is a direct conclusion of Theorem 2.3.7. Hence, we have motivated the following definition.

Definition 2.3.16. *Let $\mathcal{L}$ be a distributive lattice with least element* 0 *and greatest element* 1*. Then we call $\mathcal{L}$ a* **Boolean lattice (or Boolean algebra)***, iff the complement exists for all $x \in \mathcal{L}$.*

A homomorphism between Boolean lattices is a homomorphism between distributive lattices preserving complements. Standard examples for Boolean lattices are the truth values with *or*, *and* and negation and the powerset $\mathbb{P}(M)$ of an arbitrary set M together with $\cap$, $\cup$ and set complement. Notice that Boolean lattices need not necessarily be complete.
In the following we have summarized some properties of the complement operator.

Lemma 2.3.10. *Let $\mathcal{L}$ be a Boolean lattice. Then we have*

(1) $\overline{\overline{x}} = x$,

(2) $\overline{1} = 0$ *and* $\overline{0} = 1$,

(3) $\overline{x \vee y} = \overline{x} \wedge \overline{y}$ *and* $\overline{x \wedge y} = \overline{x} \vee \overline{y}$ *(de Morgan)*

for all $x, y \in \mathcal{L}$.

Furthermore, there is an interesting interaction between Boolean and Brouwerian lattices.

Theorem 2.3.9. *A Brouwerian lattice $\mathcal{L}$ with least element 0 is a Boolean lattice iff $x \vee x : 0 = 1$ holds for all $x \in \mathcal{L}$.*

As mentioned before, atoms play an important role within Boolean lattices. They are the base from which every element of the underlying lattice can be constructed. This fact is summarized in the following theorem.

Theorem 2.3.10. *Let $\mathcal{L}$ be a Boolean lattice. Then we have the following :*

(1) $\mathcal{L}$ is atomic.

(2) $\mathcal{L}$ is isomorphic to $\mathbb{P}(at(\mathcal{L}))$.

Especially property (2) is essential since the set of atoms fixes a Boolean algebra up to isomorphism.
In the last section we saw that complete Brouwerian lattices are completely upwards distributive. Furthermore, we mentioned that complete upwards distributivity does, in general, not imply complete downwards distributivity and vice versa. Hence, one can ask for complete distributivity within complete Boolean lattices.

Theorem 2.3.11. *Let $\mathcal{L}$ be a complete Boolean lattice. Then we have that*

(1) the generalized distributive laws

$$x \wedge \bigvee M = \bigvee_{y \in M} (x \wedge y), \qquad x \vee \bigwedge M = \bigwedge_{y \in M} (x \vee y)$$

and

(2) the generalized rules of de Morgan

$$\overline{\bigvee_{y \in M} y} = \bigwedge_{y \in M} \overline{y}, \qquad \overline{\bigwedge_{y \in M} y} = \bigvee_{y \in M} \overline{y}$$

hold for all $x \in \mathcal{L}$ and all subsets M of $\mathcal{L}$.

Thus, complete Boolean lattices constitute a strong and comfortable algebraic structure.

2.3.8 Fixpoints

In Mathematics and Computer Science many operations can be described by fixpoints of certain functions. Since we want to present a Haskell module for lattices, we want to provide the basic results concerning fixpoints and lattices in this section. We start with the following definition.

Definition 2.3.17. *Let f be a monotone endofunction on a given lattice $\mathcal{L}$. Then an element $x \in \mathcal{L}$ is called*

(1) a **prefixpoint** *of f $:\Leftrightarrow f(x) \leq x$,*

(2) a **fixpoint** *of f $:\Leftrightarrow f(x) = x$,*

(3) the **least fixpoint of f greater or equal to** *a $:\Leftrightarrow$ x is a fixpoint, $a \leq x$ and for all fixpoints y with $a \leq y$ we have $x \leq y$.*

In the following we want to provide two central fixpoint theorems which examine the existence of fixpoints resp. the induced structure of the set of all fixpoints. restriction we

Theorem 2.3.12 (Knaster and Tarski, Part 1). *Let $\mathcal{L}$ be a complete lattice and f a monotone endofunction on $\mathcal{L}$. Then*

(1) f has a least fixpoint $\mu_f = \bigwedge\{x \in \mathcal{L} \mid f(x) \leq x\}$,

(2) f has a greatest fixpoint $\nu_f = \bigvee\{x \in \mathcal{L} \mid x \leq f(x)\}$.

Notice that we need *complete* lattices to apply the last theorem. Property (1) obviously shows that the least fixpoint can be computed as the infimum of all prefixpoints. Theorem 2.3.12 directly implies

(1) $f(x) \leq x \Rightarrow \mu_f \leq x$ and

(2) $x \leq f(x) \Rightarrow x \leq \nu_f$.

The next theorem is even more important.

Theorem 2.3.13 (Knaster and Tarski, Part 2). *Let $\mathcal{L}$ be a complete lattice, f be a monotone endofunction on $\mathcal{L}$ and $Fix(f)$ be the set of all fixpoints of f. Then $Fix(f)$ is a complete sublattice of $\mathcal{L}$.*

From Theorem 2.3.12 we know that $Fix(f) \neq \emptyset$. Since $0_{\mathcal{L}}$ and $1_{\mathcal{L}}$ are not necessarily fixpoints, we can only conclude $0_{\mathcal{L}} \leq 0_{Fix(f)}$ and $1_{\mathcal{L}} \geq 1_{Fix(f)}$.

We do not want to undergo further investigations on fixpoint theory since it is not necessary for this thesis. But, the last two theorems will be applied within our Haskell module for lattices.

2.4 $\mathcal{L}$-fuzzy sets and $\mathcal{L}$-fuzzy relations

J. A. Goguen in [6] generalized the concept of fuzzy sets and fuzzy relations. Instead of the unit interval $[0,1]$ he used elements of arbitrary completely distributive lattices as entries. The necessity of this extended approach to fuzzy theory can already be seen with every day applications. If a customer, for example, wants to buy a new car, he could make the decision due to certain quality criteria like price, maximum speed and age of the car. Furthermore, he could use the phrases $BAD, MEDIUM$ and $GOOD$ to express the degree of fulfillment of one of these parameters. Now, let C be the set of all cars. Suppose, the customer rates the offered cars using the quality criteria above and the three phrases. This induces a fuzzy set representing the degree of fulfillment of the customer's wishes. According to our terminology, C is the universe of the resulting fuzzy set. This fuzzy set consists of tuples $(c,(v_1,v_2,v_3))$ whereas $c \in C$ and $v_1,v_2,v_3 \in \{BAD, MEDIUM, GOOD\}$. Obviously, the induced entry lattice is no linear order and definitely not the unit interval.

After this motivating example, we want to provide the basic results for $\mathcal{L}$-fuzzy relations. We do not explicitly introduce $\mathcal{L}$-fuzzy sets since the extension of fuzzy sets to $\mathcal{L}$-fuzzy sets is obvious. We want to go into detail with $\mathcal{L}$-fuzzy relations as announced at the end of Section 2.2. Again, all proofs are omitted due to space considerations. We refer to [6] and especially [11],[13] for the interested reader.

Definition 2.4.1. *Let $\mathcal{L}$ be a lattice and A,B be sets. Then an* **$\mathcal{L}$-fuzzy relation** *R over $\mathcal{L}$ from A to B is a mapping $R : A \times B \to \mathcal{L}$.*

We say R has *source* A and *target* B. Furthermore, we identify the entry at position (x,y) by $R(x,y)$. The induced ordering is denoted by $\sqsubseteq$. Again, we use $R^{\smile}$ to identify the converse relation of R. Although the last definition covers $\mathcal{L}$-fuzzy relations over arbitrary lattices $\mathcal{L}$, we implicitly, unless otherwise stated, want to prerequire $\mathcal{L}$ to be a complete Brouwerian lattice throughout this section. Thus, we again have an identity relation ($\mathbb{I}_A$) on every set A as well as a top resp. bottom relation ($\top\!\!\!\top_{AB}/\bot\!\!\!\bot_{AB}$) between two sets A and B.

Because of the fact that $\mathcal{L}$ is a lattice, $\wedge$ and $\vee$ automatically induce a set of componentwise defined operations on $\mathcal{L}$-fuzzy relations (join, meet, composition). But, we want to examine derived operations analogous to t- and t-conorms with fuzzy sets and fuzzy relations. This leads us to the following definition.

Definition 2.4.2. *Let $\mathcal{L}$ be a distributive lattice with least element* 0 *and greatest element* 1. *Furthermore, let $* : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ be a binary operation on $\mathcal{L}$ and e,z be elements of $\mathcal{L}$.*

Then we call the tuple ($\mathcal{L}$,$*$,e,z) a **lattice ordered operator set** *(loos), iff*

(1) $*$ is monotonic in both arguments,
(2) $x * e = e * x = x$ for all $x \in \mathcal{L}$,
(3) $x * z = z * x = z$ for all $x \in \mathcal{L}$

are satisfied, and a **lattice ordered semi group** *(losg), iff additionally*

(4) $*$ is associative

holds. Furthermore, a loos/losg ($\mathcal{L}$,$*$,e,z) fulfilling

$$x * (\bigvee M) = \bigvee_{y \in M} (x * y) \text{ and } (\bigvee M) * x = \bigvee_{y \in M} (y * x)$$

for nonempty subsets M of $\mathcal{L}$ is called a **complete** *loos/losg (cloos/closg).*

Obviously, e resp. z play the role of the unique neutral resp. zero element. Notice that we do not explicitly demand $*$ to be commutative. Hence, the structures defined above are called *commutative* iff $*$ is.
The connection between lattice ordered semi groups and t- resp. t-conorms is obvious. If $\mathcal{L} = [0, 1]$, $e = 1$, $z = 0$ and $*$ is commutative, the tuple ($\mathcal{L}$,$*$,e,z) is a t-norm. Analogously, ($\mathcal{L}$,$*$,e,z) is a t-conorm, if $e = 0$, $z = 1$. In the following we often speak of loos-/losg-/cloos-/closg-based operators $*$ and omit the tuple if it is not necessary.
The next lemma summarizes some basic properties used later on.

Lemma 2.4.1. *Let $\mathcal{L}$ be a distributive lattice with least element* 0 *and greatest element* 1 *and ($\mathcal{L}$,$*$,e,z) be a loos. If* e = 1 *we have :*

(1) $z = 0$,
(2) $x * y \leq x \wedge y$ for all $x, y \in \mathcal{L}$.

Analogously, the following is true if e = 0 *:*

(1) $z = 1$,
(2) $x \vee y \leq x * y$ for all $x, y \in \mathcal{L}$.

Obviously, this lemma marks $\wedge$ as the greatest loos-based operator with neutral element 1 and $\vee$ as the least loos-based operator with neutral element 0. We already mentioned this fact in Section 2.2. It is a very important result and will be used to show certain properties of derived operations in Goguen categories, later on.

Hence, we are ready to define the following operations on $\mathcal{L}$-fuzzy relations.

Definition 2.4.3. *Let $\mathcal{L}$ be a distributive lattice with 0 and 1, and let ($\mathcal{L}$,$*$,e,z) be a loos. Then we define for $\mathcal{L}$-fuzzy relations $Q, Q' : A \to B$ and $R : B \to C$:*

(1) $(Q \sqcap_* Q')(x,y) := Q(x,y) * Q'(x,y)$

(2) $(Q;_* R)(x,z) \;:= \bigvee_{y \in B} (Q(x,y) * R(y,z)).$

If $* = \wedge$ we write $\sqcap$ instead of $\sqcap_\wedge$ resp. $;$ instead of $;_\wedge$. If $* = \vee$ we use $\sqcup$ for $\sqcap_\vee$. We then obviously obtain the standard meet, join and composition operators on $\mathcal{L}$-fuzzy relations. In the following we often write $\sqcup_*$ instead of $\sqcap_*$ to indicate that $*$ has neutral element 0. From Lemma 2.4.1 we immediately conclude that $Q \sqcap_* Q' \sqsubseteq Q \sqcap Q'$ holds for all relations $Q, Q' : A \to B$, if 1 is the left and right neutral element of $*$. Analogously, $Q \sqcup Q' \sqsubseteq Q \sqcup_* Q'$, if 0 is the neutral element of $*$.

The following lemma lists some intuitive properties of the standard operations.

Lemma 2.4.2. *Let $\mathcal{L}$ be a complete Brouwerian lattice. Furthermore, let I be an index set and $Q, Q', Q_i : A \to B$, $R, R_i : B \to C$, $S : C \to D$ and $T : A \to C$ be $\mathcal{L}$-fuzzy relations. Then we have*

(1) $Q;\mathbb{I}_B = Q$ *and* $\mathbb{I}_B;R = R$,

(2) $Q;\perp\!\!\!\perp_{BE} = \perp\!\!\!\perp_{AE}$ *and* $\perp\!\!\!\perp_{EB};R = \perp\!\!\!\perp_{EC}$ *for all nonempty sets E,*

(3) $Q;(R;S) = (Q;R);S$,

(4) $(Q \sqcap Q')^\smile = Q^\smile \sqcap Q'^\smile$,

(5) $(Q;R)^\smile = R^\smile;Q^\smile$,

(6) $(Q^\smile)^\smile = Q$,

(7) $Q;(\bigsqcap_{i \in I} R_i) \sqsubseteq \bigsqcap_{i \in I}(Q;R_i)$ *and*
$(\bigsqcap_{i \in I} Q_i);R \sqsubseteq \bigsqcap_{i \in I}(Q_i;R)$, *(meet subdistributivity of ;)*

(8) $Q;(\bigsqcup_{i \in I} R_i) = \bigsqcup_{i \in I}(Q;R_i)$ *and*
$(\bigsqcup_{i \in I} Q_i);R = \bigsqcup_{i \in I}(Q_i;R)$, *(join distributivity of ;)*

(9) $Q;R \sqcap T \sqsubseteq Q;(R \sqcap Q^\smile;T)$. *(modular law)*

Notice, that the prerequisite that $\mathcal{L}$ is a complete Brouwerian lattice (and, hence, completely upwards distributive) is essential for properties (3),(7) and (8). The last lemma includes many properties (e.g., (7)-(9)) that will be used to axiomatize relational categories, later on.

The possibility to derive operations based on lattice-ordered semi-groups is a characteristic mark of $\mathcal{L}$-fuzzy relations. Since Goguen categories aim at a convenient algebraic surrounding for such relations, derived operations have to be modeled. Since only purely component free methods are available within relational categories, one has to examine whether there is a component free counterpart of Definition 2.4.3.

First, we have to find relations which reflect the underlying lattice $\mathcal{L}$. The approach in [11] uses scalars introduced in [10].

Definition 2.4.4. *Let A be a nonempty set, and let $\mathcal{L}$ be a lattice with least element 0 and greatest element 1. For an element $u \in \mathcal{L}$ we define the corresponding* **scalar** $\alpha_A^u : A \to A$ **on** *A by*

$$\alpha_A^u(x,y) := \begin{cases} u, & if\ x = y \\ 0, & otherwise \end{cases}.$$

Obviously, $\perp\!\!\!\perp_{AA} = \alpha_A^0$ and $\mathbb{I}_A = \alpha_A^1$ which marks the bottom relation as the least and the identity relation as the greatest scalar on A. The relation $\alpha_A^u;\top\!\!\!\top_{AB}$ containing only entries u will be denoted by $\top\!\!\!\top_{AB}^u$ in the following. It is easy to see that the set of scalars is isomorphic to $\mathcal{L}$.

Now, the question arises how a relation R can be computed using the scalars. To do so, we need the following.

Definition 2.4.5. *Let $\mathcal{L}$ be a lattice, $R : A \to B$ an $\mathcal{L}$-fuzzy relation and $u \in \mathcal{L}$. Then we define the u-***cut** *of R by*

$$R^u(x,y) := \begin{cases} 1, & if\ R(x,y) \geq u \\ 0, & otherwise \end{cases}.$$

The crisp relation R^u obviously marks all entries greater or equal to u by a 1. It is clear intuitively that we can compute $R : A \to B$ by

$$R = \bigsqcup_{u \in \mathcal{L}} (\alpha_A^u; R^u)$$

within complete Brouwerian lattices. But, this is still no component free representation since R^u is defined componentwise. A survey is to compute R^u via residued operations.

Definition 2.4.6. *Let $\mathcal{L}$ be a lattice and $Q : A \to B$, $R : B \to C$ and $S : A \to C$ be $\mathcal{L}$-fuzzy relations. Then we define*

(1) $Q;Y \sqsubseteq S \Leftrightarrow Y \sqsubseteq Q\backslash S,$ **(right residual)**

(2) $Y;R \sqsubseteq S \Leftrightarrow Y \sqsubseteq S/R.$ **(left residual)**

Notice that we need $\mathcal{L}$ to be a complete Brouwerian lattice to guarantee the existence of the residuals. This becomes clear with the following lemma, which shows how the resiudals can be computed.

Lemma 2.4.3. *Let $\mathcal{L}$ be a complete Brouwerian lattice and $Q : A \to B$, $R : B \to C$ and $S : A \to C$ be $\mathcal{L}$-fuzzy relations. Then we have*

$$\begin{aligned} (1)\ (Q\backslash S)(y,z) &= \bigwedge_{x\in A} (S(x,z) : Q(x,y)) \\ (2)\ (S/R)(x,y) &= \bigwedge_{z\in C} (S(x,z) : R(y,z)). \end{aligned}$$

Obviously, complete Brouwerian lattices are essential since we need to compute relative pseudo complements and the infimum of certain subsets of $\mathcal{L}$.
Now, consider the relation $\alpha^u_A\backslash R$ for a given scalar on A and an $\mathcal{L}$-fuzzy relation $R : A \to B$. Obviously, this relation has a 1 at position (x, y), iff $R(x, y) \geq u$ holds. All other entries are lower then 1 or even 0. Hence, R^u is the greatest crisp relation, which $\alpha^u_A\backslash R$ includes. This motivates the following definition.

Definition 2.4.7. *Let $\mathcal{L}$ be a lattice with least element* 0 *and greatest element* 1*. Furthermore, let $R : A \to B$ be an $\mathcal{L}$-fuzzy relation. Then we define two operations as follows:*

(1) $R^\downarrow$ is the greatest crisp relation which R includes.

(2) $R^\uparrow$ is the least crisp relation R is included in.

It is easy to see that $R^\downarrow$ and $R^\uparrow$ for a given $\mathcal{L}$-fuzzy relation R can be computed componentwise by

$$\begin{aligned} R^\downarrow(x,y) &= \begin{cases} 1, & if\ R(x,y) = 1 \\ 0, & otherwise \end{cases} \\ R^\uparrow(x,y) &= \begin{cases} 1, & if\ R(x,y) \neq 0 \\ 0, & otherwise \end{cases} . \end{aligned}$$

Hence, we are able to provide the following theorem.

Theorem 2.4.1 (**α-cut Theorem**)**.** *Let $\mathcal{L}$ be a complete Brouwerian lattice and $R : A \to B$ an $\mathcal{L}$-fuzzy relation. Then we have*

$$R = \bigsqcup_{u\in\mathcal{L}} (\alpha^u_A; (\alpha^u_A \backslash R)^{\downarrow}).$$

From the underlying definitions we see that we have a purely component-free representation of R. With these remarks, the following characterization of derived operations is obvious.

Lemma 2.4.4. *Let $\mathcal{L}$ be a complete Brouwerian lattice and $Q, Q' : A \to B$ and $R : B \to C$ be $\mathcal{L}$-fuzzy relations. Furthermore, let ($\mathcal{L}$,$*$,e,z) be a loos. Then we have :*

$$\begin{aligned} Q \sqcap_* Q' &= \bigsqcup_{u,v\in\mathcal{L}} (\alpha^{u*v}_A; ((\alpha^u_A\backslash Q)^{\downarrow} \sqcap (\alpha^v_A\backslash Q')^{\downarrow})) \\ Q;_* R &= \bigsqcup_{u,v\in\mathcal{L}} (\alpha^{u*v}_A; (\alpha^u_A\backslash Q)^{\downarrow}; (\alpha^u_B\backslash R)^{\downarrow}) \end{aligned}$$

Finally, we want to list three essential properties of $^{\downarrow}$ and $^{\uparrow}$ which will be used later on.

Lemma 2.4.5. *Let $\mathcal{L}$ be a complete Brouwerian lattice. Furthermore, let $Q : A \to B$ be an $\mathcal{L}$-fuzzy relation and $u \in \mathcal{L}$. Then we have :*

(1) $(\alpha^u_A)^{\uparrow} = \mathbb{I}_A$ iff $u \neq 0$.
(2) Q is crisp iff $Q = Q^{\uparrow}$ iff $Q = Q^{\downarrow}$.
(3) ($^{\uparrow}$,$^{\downarrow}$) is a monotone Galois connection.

2.5 Category theoretical approach to relations

In this section we introduce a component free algebraic approach to relational structures by means of category theory. Since category theory is dealing with objects and morphisms between them, it is pretty well applicable for this purpose. Hence, the developed relational categories provide a convenient surrounding to reason over relations. As we aim to extend the RATH system, we combine the category theoretical introduction in this section with a slight introduction to RATH. Thus, we directly show the practical realization of the mentioned relational structures. As the RATH system is relatively complex, we only give an overview of the most important data structures and test routines. For a comprehensive description we refer to [17]. Furthermore, we assume that the reader is familiar with the basic concepts of the pure functional programming language Haskell. For an easy to read and comprehensive introduction we recommend [19] and [20]. As before, we omit all proofs of the category theoretical results introduced here and refer, for example, to [11],[17] and [9] for the interested reader.

2.5.1 Categories

Categories constitute the weakest structure within the hierarchy of relational categories. Their formal definition is given as follows.

Definition 2.5.1. *A category $\mathcal{C}$ is a tuple $(Obj_{\mathcal{C}}, Mor_{\mathcal{C}}, \_ : \_ \to \_, ;, \mathbb{I})$ with*

(1) a class of objects $Obj_{\mathcal{C}}$,

(2) a class of morphisms $Mor_{\mathcal{C}}$,

(3) a ternary relation $f : A \to B$ mapping each morphism $f \in Mor_{\mathcal{C}}$ univalently to its source A and its destination B,

(4) the binary associative composition operator ; which maps each pair of morphisms $f : A \to B$, $g : B \to C$ to the morphism $f;g : A \to C$,

(5) the unary identity operator $\mathbb{I}$ which maps every object $A \in Mor_{\mathcal{C}}$ to a morphism $\mathbb{I}_A$ which is right and left neutral with respect to ;, i.e., $\mathbb{I}_A;f = f$ and $h;\mathbb{I}_A = h$ for all morphisms f in $\mathcal{C}[A,B]$ and h in $\mathcal{C}[C,A]$ and all objects B, C,

whereas $\mathcal{C}[A,B]$ is the class of all morphisms from A to B.

Notice that we speak of classes of objects and morphisms. This does not imply that they are representable as sets. Furthermore, we use the same symbol ; for the composition operator as in the previous sections. This should not irritate, and it will be clear from the context which one is meant. For clearness, the morphisms are often called relations in the following. Typical categories are, for example, orders together with order homomorphisms, nonempty sets together with all $\mathcal{L}$-fuzzy relations between them and usual sets with all functions between them. The underlying identity and composition operators then are obvious.

To express interactions between given categories, we have the notion of a *functor*.

Definition 2.5.2. *A* **functor** *$\mathcal{F}$ between categories $\mathcal{C}_1$ and $\mathcal{C}_2$ is a tuple of mappings*

$$(F_{Obj} : Obj_{\mathcal{C}_1} \to Obj_{\mathcal{C}_2}, F_{Mor} : Mor_{\mathcal{C}_1} \to Mor_{\mathcal{C}_2})$$

satisfying the following conditions :

(1) $f : A \to B$ implies $F_{Mor}(f) : F_{Obj}(A) \to F_{Obj}(B)$ for all objects A, B of $\mathcal{C}_1$ and

all morhpisms f in $\mathcal{C}_1[A,B]$.

(2) F_{Mor} preserves composition, i.e., $F_{Mor}(f;g) = F_{Mor}(f);F_{Mor}(g)$ for all objects A,B,C of $\mathcal{C}_1$ and all morphisms f in $\mathcal{C}_1[A,B]$, g in $\mathcal{C}_1[B,C]$.

(3) F_{Mor} preserves identity, i.e., $F_{Mor}(\mathbb{I}_A) = \mathbb{I}_{F_{Obj}(A)}$ for all objects A of $\mathcal{C}_1$.

A functor is called **faithful** *iff F_{Mor} is injective. Furthermore, it is called* **full** *iff F_{Mor} is surjective on the image of F_{Obj}.*

A functor $\mathcal{F}$ such that F_{Obj} and F_{Mor} are bijective and $\mathcal{F}^{-1} :=(F_{Obj}^{-1},F_{Mor}^{-1})$ is again a functor, is called an *isomorphism.*
We do not want to go into detail with the theory of functors since we do not need it for this thesis. Our attention goes to the implementation of these structures within RATH. Haskell provides a type class system, which provides inheritance and, hence, makes a kind of object-oriented programming possible. The implementation using type classes is straightforward within RATH.

```
class Category cat obj mor | cat -> obj, cat -> mor where
  isObj   :: cat -> obj -> Bool
  isMor   :: cat -> obj -> obj -> mor -> Bool
  objects :: cat -> [obj]
  homset  :: cat -> obj -> obj -> [mor]
  source  :: cat -> mor -> obj
  target  :: cat -> mor -> obj
  idmor   :: cat -> obj -> mor
  comp    :: cat -> mor -> mor -> mor
```

The function `isObj` delivers, given an object, whether this object is a member of the underlying category or not. Analogously does `isMor` with morphisms. The objects of the category are stored in the object list `objects`. The function `homset` delivers $\mathcal{C}[A,B]$ for two given objects A and B. Source resp. target of a given morphism f can be computed using `source` resp. `target`. Finally, the identity morphism of a given object and the composition operator are provided. This data structure is a direct realization of Definition 2.5.1, although it includes some redundant parameters (e.g., `isObj` and `isMor`). But, this gives the user the chance to provide more efficient functions than the standard implementations.
Unfortunately, the type class representation above has a great demerit. Obviously, it takes three type parameters `cat`, `obj` and `mor`. Type classes of this form are called multi parameter type classes. Since type inference of such constructs is not yet proven to be mathematically correct, they are no Haskell 98 standard. Therefore, RATH provides a second approach using record data structures.

```
data Cat obj mor = Cat { cat_isObj   :: obj -> Bool
                        ,cat_isMor   :: obj -> obj -> mor -> Bool
                        ,cat_objects :: [obj]
                        ,cat_homset  :: obj -> obj -> [mor]
                        ,cat_source  :: mor -> obj
                        ,cat_target  :: mor -> obj
                        ,cat_idmor   :: obj -> mor
                        ,cat_comp    :: mor -> mor -> mor }
```

This variant provides the same functionality and is Haskell 98 standard. But, it has the demerit that it does not allow to create instances and, hence, functionality cannot be inherited. This will become clear when we introduce the next data structures in the following section.

These two variants imply that RATH deals with two different views — a type class view (object oriented) and a view based on record data structures. They can be connected by instantiating the multi parameter type classes with the corresponding record data structures as follows.

```
instance Category (Cat obj mor) obj mor where
  isObj   = cat_isObj
  isMor   = cat_isMor
  ...
  comp    = cat_comp
```

Finally, the representation of a functor is straightforward.

```
data Fun obj2 mor2 obj1 mor1 = Fun { fun_obj :: obj1 -> obj2
                                    ,fun_mor :: mor1 -> mor2 }
```

The reversed order of the type signature comes due to comfortable typing while interacting with other data structures and should not irritate.

The data structures introduced here are accompanied by comprehensive test routines to check whether a certain implementation constitutes such a structure (e.g., a category). We do not want to go into detail with it here and come back to it later.

2.5.2 Allegories

Allegories provide two additional operations — meet and conversion. The axiomatic description is given as follows.

Definition 2.5.3. *An allegory $\mathcal{A}$ is a tuple $(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smile}, \sqcap)$ such that*

(1) $(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I})$ *is the underlying category,*

(2) $\mathcal{A}[A, B]$ *together with* $\sqcap$ *is a lower semilattice for all objects* A *and* B *in* $Obj_{\mathcal{A}}$*,*

(3) ${}^{\smile}$ *is the total and monotonic conversion operator which suffices the following rules for all relations* $Q, R : A \to B$ *and* $S : B \to C$*:*

(a) $(R^{\smile})^{\smile} = R$, $R^{\smile} : B \to A$*,*

(b) $(R;S)^{\smile} = S^{\smile};R^{\smile}$*,*

(c) $(Q \sqcap R)^{\smile} = Q^{\smile} \sqcap R^{\smile}$*,*

(4) the meet subdistributivity law $Q;(R \sqcap S) \sqsubseteq Q;R \sqcap Q;S$ *is valid for all relations* $Q : A \to B$ *and* $R, S : B \to C$*,*

(5) the modular law $Q;R \sqcap S \sqsubseteq Q;(R \sqcap Q^{\smile};S)$ *holds for all relations* $Q : A \to B$*,* $R : B \to C$ *and* $S : A \to C$*.*

Thus, allegories with conversion and meet can be used to express interactions between given relations using exclusivly componentfree methods. The induced inclusion operator is denoted by $\sqsubseteq$. Again, we use the same symbols for conversion, meet and inclusion as for $\mathcal{L}$-fuzzy relations. The context will make clear which one is meant. Furthermore, it is based on the fact that nonempty sets with all $\mathcal{L}$-fuzzy relations between them together with ${}^{\smile}$ and $\sqcap$ (defined on $\mathcal{L}$-fuzzy relations) constitute an allegory.

A functor between allegories is a functor preserving meet and conversion. This immediately implies that a functor $\mathcal{F}$ on allegories $\mathcal{A}_1$ and $\mathcal{A}_2$ is a lower semilattice homomorphism between the classes $\mathcal{A}_1[A, B]$ and $\mathcal{A}_2[F_{Obj}(A), F_{Obj}(B)]$.

As before, the RATH system provides a type class for allegories.

```
class Category all obj mor => Allegory all obj mor | all -> obj, all -> mor %%@
where
  converse :: all -> mor -> mor
  meet     :: all -> mor -> mor -> mor
  incl     :: all -> mor -> mor -> Bool
```

The context `Category all obj mor` makes clear that the functionality of the type class

`Category` is inherited. The additional operations should be clear. Again, the function `incl` is redundant from a mathematical point of view, but allows a more efficient, user defined operation.

The counterpart using record data structures is given as follows.

```
data All obj mor = All { all_cat      :: Cat obj mor
                        ,all_converse :: mor -> mor
                        ,all_meet     :: mor -> mor -> mor
                        ,all_incl     :: mor -> mor -> Bool }
```

Here we see that the underlying category has to be carried explicitly in the parameter `all_cat`. Again, the advantage of type classes becomes clear with the following instantiations.

```
instance Category (All obj mor) obj mor where
  isObj   = cat_isObj . all_cat
  isMor   = cat_isMor . all_cat
  ...
  comp    = cat_comp  . all_cat
instance Allegory (All obj mor) obj mor where
  converse = all_converse
  meet     = all_meet
  incl     = all_incl
```

Obviously, the structure `All` constitutes both — a category and an allegory. Together with `Cat` we already have two instances of `Category`.

In the following lemma we want to summarize some first results within allegories.

Lemma 2.5.1. *Let $\mathcal{A}$ be an allegory, A, B and C objects of $\mathcal{A}$ and $Q, R : A \to B$, $S : B \to C$ relations. Then we have*

(1) $\mathbb{I}_A^{\smile} = \mathbb{I}_A$,
(2) $(Q \sqcap R); S \sqsubseteq Q; S \sqcap R; S$,
(3) ; *is monotonic in both arguments.*

The provided operations within allegories allow to characterize special relations. This is done by the following definition.

Definition 2.5.4. *Let $\mathcal{A}$ be an allegory, A and B objects of $\mathcal{A}$ and $Q : A \to B$ a relation. Then we define :*

(1) Q is **univalent** *iff $Q^{\smile};Q \sqsubseteq \mathbb{I}_B$.*

(2) Q is **total** *iff $\mathbb{I}_A \sqsubseteq Q;Q^{\smile}$.*

(3) Q is a **mapping** *iff Q is total and univalent.*

(4) Q is **injective** *iff $Q^{\smile}$ is univalent.*

(5) Q is **surjective** *iff $Q^{\smile}$ is total.*

(6) Q is **bijective** *iff Q and $Q^{\smile}$ are mappings.*

Confer these characterizations to Lemma 2.1.3. They obviously directly correspond to the set theoretic counterparts.

Finally, we want to provide some important results which show when meet-subdistributivity turns into meet-distributivity.

Lemma 2.5.2. *Let $\mathcal{A}$ be an allegory, A, B, C and D objects of $\mathcal{A}$ and $Q : A \to B$, $R, S : B \to C$, $T : C \to D$, $U : A \to C$ relations such that T is injective. Then we have :*

(1) If Q is univalent, then $Q;(R \sqcap S) = Q;R \sqcap Q;S$.

(2) $(R \sqcap S);T = R;T \sqcap S;T$.

(3) If Q is injective, then $Q;R \sqcap U = Q;(R \sqcap Q^{\smile};U)$.

2.5.3 Distributive allegories

The classes of morphisms within allegories form a lower semilattice. The next step is to extend this structure to distributive lattices.

Definition 2.5.5. *A distributive allegory $\mathcal{A}$ is a tuple*

$$(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smile}, \sqcap, \sqcup, \perp\!\!\!\perp)$$

such that

(1) $(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smile}, \sqcap)$ is the underlying allegory,

(2) for all objects A and B in $Obj_{\mathcal{A}}$ $\mathcal{A}[A, B]$ together with $\sqcap$ and $\sqcup$ is a distributive lattice with least element,

(3) $\perp\!\!\!\perp$ maps $\mathcal{A}[A, B]$ to its least element $\perp\!\!\!\perp_{A,B}$,

(4) the join distributivity rule $Q;(R \sqcup S) = Q;R \sqcup Q;S$ is valid for all relations $Q : A \to B$ and $R, S : B \to C$,

(5) the zero law $Q;\perp\!\!\!\perp_{B,C} = \perp\!\!\!\perp_{A,C}$ holds for all relations $Q : A \to B$ and all objects A, B and C.

As before, a functor between distributive allegories is a functor between allegories preserving $\sqcup$ and the bottom elements.

The realization of this data structure within RATH is straightforward by providing additional operations for $\sqcup$ and the determination of the bottom elements. Hence, we omit the listing due to space considerations.

From Section 2.3 we know that distributive lattices constitute a relatively comfortable algebraic surrounding. Of course, all results presented there for distributive lattices are valid in all morphism classes of distributive allegories. Again, the nonempty sets together with all $\mathcal{L}$-fuzzy relations between them constitute a distributive allegory.

The following lemma summarizes the interaction of $\sqcup$ and $\perp\!\!\!\perp$ together with $\smallsmile$.

Lemma 2.5.3. *Let $\mathcal{A}$ be a distributive allegory, A, B objects of $\mathcal{A}$ and $Q, R : A \to B$ relations. Then we have*

(1) $\perp\!\!\!\perp_{AB}^{\smallsmile} = \perp\!\!\!\perp_{AB}$,
(2) $(Q \sqcup R)^{\smallsmile} = Q^{\smallsmile} \sqcup R^{\smallsmile}$.

From (1) and axiom (5) we can immediately conclude that $\perp\!\!\!\perp_{AB}$ is also a left zero for all relations $Q : B \to C$ and all objects A, C.

2.5.4 Division allegories

Residuals are provided with the next relational category.

Definition 2.5.6. *A division allegory $\mathcal{A}$ is a tuple*

$$(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smallsmile}, \sqcap, \sqcup, \perp\!\!\!\perp, /)$$

such that

(1) $(Obj_{\mathcal{A}}, Mor_{\mathcal{A}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smallsmile}, \sqcap, \sqcup, \perp\!\!\!\perp)$ *is the underlying distributive allegory,*
(2) *the left residual / fulfilling*

$$Q; R \sqsubseteq S \;\Leftrightarrow\; Q \sqsubseteq S/R \text{ for all relations } Q : A \to B$$

exists in $\mathcal{A}[A, B]$ for all relations $R : B \to C$ and $S : A \to C$.

Again, the residuals defined for $\mathcal{L}$-fuzzy relations make the nonempty sets together with all $\mathcal{L}$-fuzzy relations between them constitute a division allegory.

We again need an extended functor notion. As expected, a functor between division allegories is a functor between distributive allegories preserving $/$.

Via conversion and the computation

$$\begin{aligned} Q \sqsubseteq S/R &\Leftrightarrow Q;R \sqsubseteq S \\ &\Leftrightarrow R^{\smile};Q^{\smile} \sqsubseteq S^{\smile} \\ &\Leftrightarrow Q^{\smile} \sqsubseteq R^{\smile}\backslash S^{\smile} \end{aligned}$$

we may conclude that there is also a right residual within division allegories and that $S/R = (R^{\smile}\backslash S^{\smile})^{\smile}$ holds.
Now, suppose we have two relations $Q : A \to B$ and $R : A \to C$ and a greatest solution $X : B \to C$ fulfilling

$$Q;X \sqsubseteq R \text{ and } X;R^{\smile} \sqsubseteq Q^{\smile}.$$

In the model provided by nonempty sets and $\mathcal{L}$-fuzzy relations between them, X relates columns of Q and R by their degree of equality. If the underlying entry lattice is the Boolean lattice of truth values, only equal columns are related. We will often need this property of X. The relation X is called the *symmetric quotient* of Q and R and is defined by

$$syQ(Q, R) := (Q\backslash R) \sqcap (Q^{\smile}/R^{\smile}).$$

Within RATH, division allegories are again provided pretty intuitively. The only thing to mention is that the data structures carry separate functions for the left and right residual as well as for the symmetric quotient.

In the following we want to recall some basic results concerning residuals and symmetric quotients. For a comprehensive study we refer to [16].

Lemma 2.5.4. *Let $\mathcal{A}$ be a division allegory and A, B, C, D, E objects of $\mathcal{A}$. Furthermore, let $Q, Q_1, Q_2 : A \to B$, $R, R_1, R_2 : B \to C$, $S, S_1, S_2 : A \to C$, $T : A \to D$, $F : D \to A$ and $G : C \to E$ be relations. Then we have*

(1) $(S_1 \sqcap S_2)/R = (S_1/R) \sqcap (S_2/R)$ and $Q\backslash(S_1 \sqcap S_2) = (Q\backslash S_1) \sqcap (Q\backslash S_2)$,
(2) $S/(R_1 \sqcup R_2) = (S/R_1) \sqcap (S/R_2)$ and $(Q_1 \sqcup Q_2)\backslash S = (Q_1\backslash S) \sqcap (Q_2\backslash S)$,
(3) $S/\mathbb{I}_C = \mathbb{I}_A\backslash S = S$,
(4) $F;(S/R) \sqsubseteq (F;S)/R$ and $(Q\backslash S);G \sqsubseteq Q\backslash(S;G)$,
(5) equality holds in (4), if F and $G^{\smile}$ are mappings,
(6) $S/R \sqsubseteq (S;G)/(R;G)$ and $Q\backslash S \sqsubseteq (F;Q)\backslash(F;S)$,

(7) equality holds in (6), if $F^{\smile}$ and G are total and injective,

(8) $F; syQ(Q,S) = syQ(Q;F^{\smile},S)$, if F is a function,

(9) $syQ(Q,S)^{\smile} = syQ(S,Q)$,

(10) $syQ(Q,S); syQ(S,T) \sqsubseteq syQ(Q,T)$.

Properties (1) and (2) of the last lemma include that / is monotonic in the first and antitonic in the second argument. The opposite is true for \. Furthermore, we immediately can conclude that *syQ* is neither necessarily monotonic nor antitonic in any of its two arguments. Especially properties (4)-(8) and (10) are important since they express the interaction of the three operations and composition.

2.5.5 Dedekind categories

Now, we switch to a structure providing an additional greatest element within the morphism classes.

Definition 2.5.7. *A Dedekind category $\mathcal{D}$ is a tuple*

$$(Obj_{\mathcal{D}}, Mor_{\mathcal{D}}, \_ : \_ \to \_, ;, \mathbb{I}, {}^{\smile}, \sqcap, \sqcup, \perp\!\!\!\perp, /, \top\!\!\!\top)$$

such that

(1) $(Obj_{\mathcal{D}}, Mor_{\mathcal{D}}, \_ : \_ \leftrightarrow \_, ;, \mathbb{I}, {}^{\smile}, \sqcap, \sqcup, \perp\!\!\!\perp, /)$ is the underlying division allegory,

(2) $\mathcal{D}[A,B]$ is a complete Brouwerian lattice for all objects A, B in $Obj_{\mathcal{D}}$,

(3) $\top\!\!\!\top$ maps $\mathcal{D}[A,B]$ to its greatest element $\top\!\!\!\top_{A,B}$.

Dedekind categories have (because of their completeness) the big advantage that every class of morphisms can be represented by sets. This type of category is called *locally small.* The nonempty sets together with all $\mathcal{L}$-fuzzy relations between them constitute a Dedekind category. Furthermore, every complete Brouwerian lattice $\mathcal{L}$ can be represented by a one-objected Dedekind category. The elements of $\mathcal{L}$ are the morphisms. Furthermore, we have to take $\wedge$ as composition and meet, and $\vee$ as join. Finally, 0 and 1 are the bottom and top element, respectively, and the relative pseudo complement is taken as residual. Composition is trivial since we only have one object. We denote the induced Dedekind category of $\mathcal{L}$ by $\mathcal{D}_{\mathcal{L}}$.

As usual, a functor $\mathcal{F}$ between two Dedekind categories $\mathcal{D}_1$ and $\mathcal{D}_2$ is a functor between division allegories preserving the top elements and completeness. This immediately implies that F_{Mor} is a complete Brouwerian lattice homomorphism between $\mathcal{D}_1[A,B]$ and

$\mathcal{D}_2[F_{Obj}(A), F_{Obj}(B)]$.

We again omit the listing of the RATH data structures. They are implemented straightforward by providing an additional function to determine the top element belonging to a given morphism class.

In the following we have summarized some results needed later on.

Lemma 2.5.5. *Let $\mathcal{D}$ be a Dedekind category, A, B, C objects and $Q, Q_i : A \to B$, $R, R_i : B \to C$, $S, S_i : A \to C$ relations with $i \in I$ for an index set I. Then we have*

(1) $\mathbb{T}_{AB}^{\smile} = \mathbb{T}_{BA}$,
(2) $\mathbb{T}_{AA};\mathbb{T}_{AB} = \mathbb{T}_{AB};\mathbb{T}_{BB} = \mathbb{T}_{AB}$,
(3) $Q;(\bigsqcup_{i\in I} R_i) = \bigsqcup_{i\in I}(Q;R_i)$ *and* $(\bigsqcup_{i\in I} Q_i);R = \bigsqcup_{i\in I}(Q_i;R)$,
(4) $Q;(\bigsqcap_{i\in I} R_i) \sqsubseteq \bigsqcap_{i\in I}(Q;R_i)$ *and* $(\bigsqcap_{i\in I} Q_i);R \sqsubseteq \bigsqcap_{i\in I}(Q_i;R)$,
(5) $(\bigsqcap_{i\in I} S_i)/R = \bigsqcap_{i\in I}(S_i/R)$ *and* $Q\backslash(\bigsqcap_{i\in I} S_i) = \bigsqcap_{i\in I}(Q\backslash S_i)$,
(6) $S/(\bigsqcup_{i\in I} R_i) = \bigsqcap_{i\in I}(S/R_i)$ *and* $(\bigsqcup_{i\in I} Q_i)\backslash S = \bigsqcap_{i\in I}(Q_i\backslash S)$.

Properties (5) and (6) are the infinitary extensions for the results provided in Lemma 2.5.4. They are (together with (3) and (4)) a direct consequence of the fact that Dedekind categories are complete structures.
Unfortunately, the restrictions with property (2) of the last lemma have to be made since it cannot be shown in general that $\mathbb{T}_{AB};\mathbb{T}_{BC} = \mathbb{T}_{AC}$ holds in a Dedekind category for all objects A, B, C. Dedekind categories in which this law is valid are called *uniform*.

In the following we want to introduce the notions of l-crispness and s-crispness. They are introduced in [10] by H. Furusawa and constitute the first approach to characterize crisp relations within the theory of Dedekind categories. Later on we then can examine under which circumstances these notions are covered within the theory of Goguen categories [11].

The approach in [10] uses scalars to characterize the underlying entry lattice. Hence, we first have to define these elements within Dedekind categories.

Definition 2.5.8. *Let $\mathcal{D}$ be a Dedekind category and A an object of $\mathcal{D}$. A relation α_A is called* **a scalar on** A A *iff* $\alpha_A \sqsubseteq \mathbb{I}_A$ *and* $\mathbb{T}_{AA};\alpha_A = \alpha_A;\mathbb{T}_{AA}$ *holds.*

In the following we denote the set of all scalars on A by $Sc_{\mathcal{D}}(A)$ whereas $\mathcal{D}$ is the underlying Dedekind category.

The following lemma indicates that this definition delivers the intended elements.

Lemma 2.5.6. *Let $\mathcal{D}$ be a Dedekind category and A an object of $\mathcal{D}$. Then $Sc_{\mathcal{D}}(A)$ is a complete Brouwerian lattice.*

The notions of l-/s-crispness are introduced as follows.

Definition 2.5.9. *Let $\mathcal{D}$ be a Dedekind category, A, B objects of $\mathcal{D}$ and $R : A \to B$ a relation. Then R is l-crisp/s-crisp iff $\alpha_A; Q \sqsubseteq R$ implies $Q \sqsubseteq R$ for all linear/nonzero scalars α_A and all relations $Q : A \to B$.*

Since $Q \sqsubseteq R$ implies $\alpha_A; Q \sqsubseteq R$ for all relations $Q, R : A \to B$ and all scalars $\alpha_A \in Sc_{\mathcal{D}}(A)$, the definition above induces an equivalence.
It is easy to see that $\top\!\!\top_{AB}$ is both l- and s-crisp for all objects A, B. But unfortunately, $\perp\!\!\perp_{AB}$ is not necessarily s-crisp. Let $\mathcal{L}$ be a complete Brouwerian lattice and consider the one-objected Dedekind category $\mathcal{D}_{\mathcal{L}}$. For the object A, $\perp\!\!\perp_{AA}$ obviously corresponds to the least element 0 of $\mathcal{L}$. Furthermore, the scalars are given by the elements of $\mathcal{L}$ since $1 \wedge x = x \wedge 1$ for all $x \in \mathcal{L}$. But, we do not necessarily have that $x \wedge y \sqsubseteq 0$ implies $y \sqsubseteq 0$ for all nonzero elements $x, y \in \mathcal{L}$. This can only be guaranteed if all elements of $\mathcal{L}$ are linear. This motivates the following lemma.

Lemma 2.5.7. *Let $\mathcal{D}$ be a Dedekind category and A, B objects of $\mathcal{D}$. Then $\perp\!\!\perp_{AB}$ is s-crisp iff all nonzero scalars are linear.*

In the next section we will see that these notions do not (and even cannot) cover crispness within Dedekind categories.

Finally, we want to introduce three essential constructions within Dedekind categories that we need for the mathematical treatment of fuzzy controllers later on. In Section 2.1 we already mentioned the possibility to create the direct sum and cartesian product of given sets. These constructions now shall be characterized with the language of Dedekind categories. We start with the direct sum.

Definition 2.5.10. *Let $\mathcal{D}$ be a Dedekind category and A_1, A_2 be objects. Then a pair of relations (called* **injections***) $\iota_1 : A_1 \to A_1 + A_2$, $\iota_2 : A_2 \to A_1 + A_2$ together with the object $A_1 + A_2$ is called a* **relational sum** *of A_1 and A_2 iff the following conditions are satisfied :*

(1) ι_i is total and injective, i.e., $\iota_i; \iota_i^{\smile} = \mathbb{I}_{A_i}$, $1 \le i \le 2$.
(2) $\iota_1; \iota_2^{\smile} = \perp\!\!\perp_{A_1A_2}$ and $\iota_2; \iota_1^{\smile} = \perp\!\!\perp_{A_2A_1}$.
(3) $\iota_1^{\smile}; \iota_1 \sqcup \iota_2^{\smile}; \iota_2 = \mathbb{I}_{A_1+A_2}$.

The last definition can easily be extended to relational sums of sets of objects. If the relational sum exists in $\mathcal{D}$ for all sets of objects, we say $\mathcal{D}$ *has relational sums.* Definition 2.5.10 determines relational sums up to isomorphism. Hence, we can speak of *the* relational sum for a given set of objects.

Cartesian products are provided as follows.

Definition 2.5.11. *Let $\mathcal{D}$ be a Dedekind category and A_1, A_2 objects of $\mathcal{D}$. The a pair of relations (called* **projections***) $\pi_1 : A_1 \times A_2 \to A_1$, $\pi_2 : A_1 \times A_2 \to A_2$ together with the object $A_1 \times A_2$ is called a* **relational product** *iff the following conditions are satisfied :*

(1) $\pi_i^{\smile}$ is total and injective, i.e., $\pi_i^{\smile};\pi_i = \mathbb{I}_{A_i}$, $1 \leq i \leq 2$.
(2) $\pi_1^{\smile};\pi_2 = \top\!\!\top_{A_1A_2}$ and $\pi_2^{\smile};\pi_1 = \top\!\!\top_{A_2A_1}$.
(3) $\pi_1;\pi_1^{\smile} \sqcap \pi_2;\pi_2^{\smile} = \mathbb{I}_{A_1\times A_2}$.

If the relational product exists for any two objects of a given Dedekind category $\mathcal{D}$, we say $\mathcal{D}$ *has relational products.* Notice that this is a slightly weaker prerequisite than for relational sums.
Again, Definition 2.5.11 determines relational products up to isomorphism so that we can speak of *the* relational product for two given objects A_1 and A_2.

Last but not least, we want to introduce an abstract counterpart of singleton sets which is used to model sets within the theory of relational categories.

Definition 2.5.12. *Let $\mathcal{D}$ be a Dedekind category. Then an object I of $\mathcal{D}$ is called a* **unit** *if $\top\!\!\top_{II} = \mathbb{I}_I$ and $\top\!\!\top_{AI}$ is total for all objects A of $\mathcal{D}$.*

Units are also unique up to isomorphism with this definition. One even can show that $I \times I$ is isomorphic to I. The characteristic mark of units delivers us directly that all relations $Q : A \to I$ are univalent and all relations $R : I \to B$ are injective for all objects A, B. Hence, sets over a given universe B can be modeled by relations $R : I \to B$. This will be used for the relational model of fuzzy controllers later on.

2.6 Goguen categories

M. Winter in [11] showed that Dedekind categories have an axiomatization which is too weak to express crispness. To motivate the reader, we want to give the underlying counterexample used in the proof. Let $\mathcal{L}$ together with $\wedge$ and $\vee$ be the lattice shown in Figure 2.4.

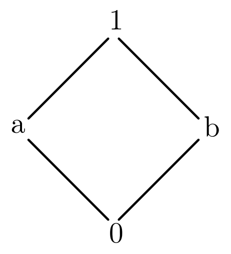

Figure 2.4: Boolean lattice with four elements

Now, let $X := \{x\}$ and $Y := \{x, y\}$ be two sets and $f : Y \to Y$, $R : X \to Y$ be two $\mathcal{L}$-fuzzy given as

$$R := \begin{pmatrix} 1 & 0 \end{pmatrix}, \qquad f := \begin{pmatrix} a & b \\ b & a \end{pmatrix}.$$

Obviously, f is bijective since $f; f^{\smile} = \mathbb{I}_Y$. Furthermore, R is crisp. Now suppose there is a formula ϕ which is valid for an $\mathcal{L}$-fuzzy relation R if and only if R is crisp. Then it can be shown within the theory of Dedekind categories that ϕ holds for every relation $S := g; R; f$ for given bijections g and f. Hence, it would have to be valid for $\mathbb{I}_X; R; f$. But, the computation

$$\begin{aligned} R; f(x, x) &= (R(x, x) \wedge f(x, x)) \vee (R(x, y) \wedge f(y, x)) = a \\ R; f(x, y) &= (R(x, x) \wedge f(y, x)) \vee (R(x, y) \wedge f(y, y)) = b \end{aligned}$$

shows that $R; f$ is not crisp. Thus, we have a contradiction.

This implies that an extended approach is necessary. In the following we want reproduce the basic definitions and properties of Goguen categories needed for this thesis. The proofs are again omitted.

2.6.1 Definition and properties

Goguen categories are based on a very intuitive approach. They use the two operations $^{\uparrow}$ resp. $^{\downarrow}$ which map an $\mathcal{L}$-fuzzy relation R to the smallest crisp relation it is included resp. the greatest crisp relation it includes (cf. Definition 2.4.7). By means of these operations one then can easily define crispness as shown in Lemma 2.4.5(2). The definition is given as follows.

Definition 2.6.1. *A Goguen Category $\mathcal{G}$ is a tuple*

$$\mathcal{G} := (Obj_{\mathcal{G}}, Mor_{\mathcal{G}}, \_ : \_ \rightarrow \_, ;, \mathbb{I}, \smile, \sqcap, \sqcup, \perp\!\!\!\perp, \top\!\!\!\top, \uparrow, \downarrow)$$

where

(1) $(Obj_{\mathcal{G}}, Mor_{\mathcal{G}}, \_ : \_ \leftrightarrow \_, ;, \mathbb{I}, \smile, \sqcap, \sqcup, \perp\!\!\!\perp, \top\!\!\!\top)$ *is the underlying Dedekind category with* $\perp\!\!\!\perp_{AB} \neq \top\!\!\!\top_{AB}$ *for all objects A and B.*

(2) $\uparrow$ *and* $\downarrow$ *satisfy the following laws :*

(2a) $R^{\uparrow}, R^{\downarrow} : A \rightarrow B$ *for all relations* $R : A \rightarrow B$.

(2b) $(\uparrow, \downarrow)$ *is a monotonic Galois correspondence.*

(2c) $(R^{\smile}; S^{\downarrow})^{\uparrow} = R^{\uparrow\smile}; S^{\downarrow}$ *for all* $R : B \rightarrow A$, $S : B \rightarrow C$.

(2d) If $\alpha_A \neq \perp\!\!\!\perp_{A,A}$ *is a nonzero scalar then* $\alpha_A^{\uparrow} = \mathbb{I}_A$.

(2e) For all antimorphisms f in $Sc_{\mathcal{G}}(A) \overset{anti}{\rightarrow} \mathcal{G}[A, B]$ *with* $f(\alpha_A)^{\uparrow} = f(\alpha_A)$, *for all* $\alpha_A \in Sc_{\mathcal{G}}(A)$ *and all* $R : A \rightarrow B$ *the equivalence*

$$R \sqsubseteq \bigsqcup_{\alpha_A \in Sc_{\mathcal{G}}(A)} (\alpha_A; f(\alpha_A)) \Leftrightarrow (\alpha_A \setminus R)^{\downarrow} \sqsubseteq f(\alpha_A) \text{ for all } \alpha_A \in Sc_{\mathcal{G}}(A)$$

holds. $\mathcal{G}$ is called **linear** *if all nonzero scalars are linear.*

Part (2e) constitutes the key property of this definition. It assures that every morphism of the respective Goguen category can be represented by its α-cut representation (cf. Theorem 2.6.2). Obviously, this property uses second order predicate logic by quantifying over all antimorphisms. But, the example at the beginning of this section shows that first order rules do not suffice.

As before, we need a suitable notion of homomorphisms between Goguen categories. Hence, a homomorphism $\mathcal{F}$ between Dedekind categories is a homomorphism between Goguen categories if it preserves $^{\uparrow}$ and $^{\downarrow}$.

We have already introduced $^{\uparrow}$ and $^{\downarrow}$ for $\mathcal{L}$-fuzzy relations (cf. Section 2.4). This automatically provides the standard model.

Theorem 2.6.1. *Let $\mathcal{L}$ be a complete Brouwerian lattice. Then $\mathcal{L}$-Rel together with $^{\uparrow}$ and $^{\downarrow}$ is a Goguen category.*

Hence, crispness is defined as expected.

Definition 2.6.2. *A relation $R : A \rightarrow B$ is* **crisp** *iff* $R^{\uparrow} = R$.

In the following we sometimes need the class of all crisp relations between two given objects A and B. Hence, we denote it by $Crisp_{\mathcal{G}}[A,B]$.
The next lemma summarizes some intuitive results.

Lemma 2.6.1. *Let a Goguen category $\mathcal{G}$ and three objects A, B and C be given. Furthermore, let $R, R_i : A \to B$, $(i \in I)$, be relations. Then we have*

(1) R is crisp, iff $R^{\downarrow} = R$ iff $R^{\downarrow} = R^{\uparrow}$,
(2) $\mathbb{I}_A$, $\perp\!\!\!\perp_{A,B}$ and $\top\!\!\!\top_{A,B}$ are crisp,
(3) $(\bigsqcup\limits_{i\in I} R_i)^{\uparrow} = \bigsqcup\limits_{i\in I} R_i^{\uparrow}$ and $(\bigsqcap\limits_{i\in I} R_i)^{\downarrow} = \bigsqcap\limits_{i\in I} R_i^{\downarrow}$

Especially part (3) that $^{\uparrow}$ preserves join and $^{\downarrow}$ preserves meet is important. Furthermore, the fact that $\perp\!\!\!\perp_{AB}$ is crisp for all objects A and B is to mention since this is, in general, not true for s-crispness.
Now, we reach the first central theorem.

Theorem 2.6.2 (**α-cut Theorem**)**.** *Let $\mathcal{G}$ be a Goguen category and $R : A \to B$ be a relation. Then R can be computed by*

$$R = \bigsqcup_{\alpha_A \in Sc_{\mathcal{G}}(A)} (\alpha_A; (\alpha_A \backslash R)^{\downarrow}).$$

Since $(\_\backslash R)^{\downarrow} : Sc_{\mathcal{G}}(A) \to Crisp_{\mathcal{G}}[A,B]$ is an antimorphism for all objects A, B and relations $R : A \to B$, this result is nearly directly delivered by Definition 2.6.1(2e).
The next lemma summarizes some conclusions of the α-cut Theorem.

Lemma 2.6.2. *Let $\mathcal{G}$ be a Goguen category, and let $R : A \to B$ be a relation.*

(1) $R^{\uparrow} = \bigsqcup\limits_{\substack{\alpha_A \in Sc_{\mathcal{G}}(A)\\ \alpha_A \neq \perp\!\!\!\perp_{AA}}} (\alpha_A \backslash R)^{\downarrow}$,
(2) $\mathcal{G}$ is uniform, i.e. $\top\!\!\!\top_{AB};\top\!\!\!\top_{BC} = \top\!\!\!\top_{AC}$ for all objects A, B, C,
(3) $\top\!\!\!\top_{CA}; R^{\uparrow}; \top\!\!\!\top_{BD} = \top\!\!\!\top_{CD}$ for all objects C, D if $R \neq \perp\!\!\!\perp_{AB}$,
(4) $R^{l} \sqsubseteq R^{\uparrow} \sqsubseteq R^{s}$.

With this, (2) is essential since this cannot be shown in general for Dedekind categories. Furthermore, (4) confirms the respective result for Dedekind categories.
The equivalence of the three notions of crispness, l-crispness and s-crispness is covered by the following theorem.

Theorem 2.6.3. *Let $\mathcal{G}$ be a Goguen category. Then the following three statements are equivalent:*

(1) $\mathcal{G}$ is linear.

(2) All crisp relations are s-crisp.

(3) All l-crisp relations are crisp.

Hence, these notions are equivalent in linear Goguen categories.
The following lemma collects some intuitive properties of crisp relations.

Lemma 2.6.3. *Let $\mathcal{G}$ be a Goguen category and $Q_i, Q : A \to B$, $i \in I$, $R : A \to C$ and $S : B \to C$ be crisp relations. Then*

(1) $\bigsqcup\limits_{i\in I} Q_i$ is crisp,

(2) $\bigsqcap\limits_{i\in I} Q_i$ is crisp,

(3) $Q^{\smile}$ is crisp,

(4) $Q;S$ is crisp,

(5) R/S and $Q\backslash R$ are crisp.

The last lemma together with Lemma 2.6.1 are absolutely basic properties and will be used without explicitly mentioning them.
We already admitted before that the set of scalars of a given object A is isomorphic to the entry lattice $\mathcal{L}$ of the underlying $\mathcal{L}$-fuzzy relations. This is shown by the next theorem for the abstract theory of Goguen categories.01

Theorem 2.6.4. *Let $\mathcal{G}$ be a Goguen category and A and B objects. Then the complete Brouwerian lattices $Sc_{\mathcal{G}}(A)$ and $Sc_{\mathcal{G}}(B)$ are isomorphic.*

This theorem is essential for theory as well as for practice. As we will see, it plays a key role for the definition and handling of derived operations within Goguen categories.

2.6.2 Derived operations

In this section we introduce a concept to model derived operations from lattice ordered semi groups. Since the underlying lattice is represented by the set of scalars of a given object A of the Goguen category, we can easily lift the definition of losg's to a component free niveau. Throughout this section let $\mathcal{G}$ be a Goguen category and ($Sc[\mathcal{G}]$,$*$,e,z) be a loos whereas $Sc[\mathcal{G}]$ is the set of all scalars on all objects of $\mathcal{G}$. Again, we often only speak of a loos- based operation $*$ and omit the induced tuple. Notice that we have a t-conorm like operation if $e = \perp\!\!\!\perp$ and a t-norm like operation if $e = \mathbb{I}$.

The first thing we need is a basic operation $\Diamond$ from which we can create the derived operation $\Diamond_*$. Two candidates for R$\Diamond$S are ; and $\sqcap$. More generally, $\Diamond$ shall satisfy the properties introduced in the following definition.

Definition 2.6.3. *Let $\mathcal{G}$ be a Goguen category, and let $\Diamond$ an operation. Then $\Diamond$ is* **the base** *for a derived operation if the following is satisfied :*

(1) For all objects A $\Diamond$ is defined for all pairs of relations from $\mathcal{G}[A,A]$. Furthermore, its value is within $\mathcal{G}[B,B]$ for a suitable object B.

(2) If $Q\Diamond R$ is defined for $Q : A \to B$ and $R : C \to D$ then $\Diamond$ is defined for all pairs of relations from $\mathcal{G}[A,B]$ and $\mathcal{G}[C,D]$.

(3) If $Q\Diamond R$ is defined for $Q : A \to B$ and $R : C \to D$ and is within $\mathcal{G}[E,F]$ then $Q\Diamond\perp\!\!\!\perp_{CD} = \perp\!\!\!\perp_{AB}\Diamond R = \perp\!\!\!\perp_{EF}$.

(4) If $\top\!\!\!\top_{AB}\Diamond\top\!\!\!\top_{CD}$ is defined and within $\mathcal{G}[E,F]$ then $\top\!\!\!\top_{AB}\Diamond\top\!\!\!\top_{CD} = \top\!\!\!\top_{EF}$.

(5) For an index set I and relations Q, Q_i, R, R_i, $i \in I$ we have
$$Q\Diamond(\bigsqcup_{i\in I} R_i) = \bigsqcup_{i\in I}(Q\Diamond R_i) \text{ and } (\bigsqcup_{i\in I} Q_i)\Diamond R = \bigsqcup_{i\in I}(Q_i\Diamond R)$$
whenever the application of $\Diamond$ is defined.

(6) If $Q\Diamond R$ is defined for $Q : A \to B$ and $R : C \to D$ and within $\mathcal{G}[E,F]$, then we have for all $\alpha, \beta \in Sc[\mathcal{G}]$
$$(\alpha_E \sqcap \beta_E);(Q\Diamond R) = (\alpha_A;Q)\Diamond(\beta_C;R).$$

(7) For all crisp relations Q, R such that $\Diamond$ is defined, $Q\Diamond R$ is crisp.

The prerequisites of the definition above are quite intuitive. We want to mention that (1) implies that $\Diamond$ has to be defined on the scalars. Furthermore (2) assures that $\Diamond$ is either not defined for a given class of relations or it is defined for the whole class. Property (6) regulates the interaction of the application of $Q\Diamond R$ and the scalars of the respective objects. Notice that this prerequisite is only senseful since we have that the complete Brouwerian lattices of the scalars of different objects A and B are isomorphic (cf. Theorem 2.6.4). Notice furthermore, that ; and $\sqcap$ fulfill Definition 2.6.3.

Now, we want to give the definition of a derived operation.

Definition 2.6.4. *Let $\Diamond$ be the base for a derived operation. Furthermore, let $Q : A \to B$ and $R : C \to D$ be relations such that $Q\Diamond R$ is defined and within $\mathcal{G}[E,F]$. Then the derived operation $\Diamond_*$ is defined by*
$$Q\Diamond_* R := \bigsqcup_{\alpha,\beta\in Sc[\mathcal{G}]} (\alpha * \beta)_E;((\alpha_A\backslash Q)^{\downarrow}\Diamond(\beta_C\backslash R)^{\downarrow}).$$

This definition obviously directly corresponds to the result shown in Lemma 2.4.4. Since $\backslash$ in the second argument and $\Diamond$ in both arguments are monotonic, the derived operation $\Diamond_*$ is also monotonic. As one can see, this definition is rather inefficient with respect to computational purposes. This will have to be considered with the parametrization for our module for Goguen categories, later on.
Together with property (6) of Definition 2.6.3, the last definition obviously implies that $; =;_\sqcap$, which is an intuitive result. But, we have an even stronger result.

Lemma 2.6.4. *Let $\Diamond$ be the base of a derived operation such that it is defined on $\mathcal{G}[A,B]$ and $\mathcal{G}[C,D]$. Then we have*

(1) $Q\Diamond_* R = Q\Diamond R$ for all $Q : A \to B$ and $R : C \to D$ iff $* = \sqcap$,

and, if $\Diamond = \sqcap$,

(2) $Q\Diamond_* Q' = Q \sqcup Q'$ for all $Q, Q' : A \to B$ iff $* = \sqcup$.

In the following we want to summarize some basic results for the cases that $*$ is a t-norm like resp. t-conorm like operation (i.e., $e = \mathbb{I}$ resp. $e = \perp\!\!\!\perp$). We start with $e = \mathbb{I}$.

Lemma 2.6.5. *Let $e = \mathbb{I}$ and $Q : A \to B$, $R : C \to D$ be relations such that $Q\Diamond R$ is defined and within $\mathcal{G}[E,F]$. Then we have :*

(1) If Q or R is crisp then $Q\Diamond_* R = Q\Diamond R$.
(2) $Q\Diamond_*;\perp\!\!\!\perp_{CD} = \perp\!\!\!\perp_{AB}\Diamond_* R = \perp\!\!\!\perp_{EF}$.
(3) $Q\sqcap_*;\top\!\!\!\top_{AB} = Q$ and $\top\!\!\!\top_{CD} \sqcap_* R = R$.
(4) $Q\Diamond_* R \sqsubseteq Q\Diamond_\sqcap R$.

Property (4) again marks $\sqcap$ as the greatest t-norm like operation.
Now, the analogous results for $e = \perp\!\!\!\perp$ follow.

Lemma 2.6.6. *Let $e = \perp\!\!\!\perp$ and $Q : A \to B$ and $R : C \to D$ be relations such that $Q\Diamond R$ is defined and within $\mathcal{G}[E,F]$. Then we have :*

(1) If Q or R is crisp then $Q\Diamond_* R = (Q\Diamond\top\!\!\!\top_{CD}) \sqcup (\top\!\!\!\top_{AB}\Diamond R)$.
(2) $Q\Diamond_*;\perp\!\!\!\perp_{CD} = \perp\!\!\!\perp_{AB}\Diamond_* R = \perp\!\!\!\perp_{EF}$.
(3) $Q\sqcap_*;\perp\!\!\!\perp_{AB} = Q$ and $\perp\!\!\!\perp_{CD} \sqcap_* R = R$.
(4) $Q\Diamond_\sqcup R \sqsubseteq Q\Diamond_* R$.

Hence, in the theory of Goguen categories we also have that $\sqcup$ is the least t-conorm like operation. In the following we often use the denotation $\sqcup_*$ instead of $\sqcap_*$ for a meet-based derived operation to emphasize that $*$ has neutral element $\perp\!\!\!\perp$.

The last two lemmas will often be used later on. They are the basic properties and show that derived operations in the abstract theory of Goguen categories can be handled quite intuitively. This is again encouraged by the following theorems.

Theorem 2.6.5. *Let $\Diamond$ be the base for a derived operation. Then we have :*

(1) If $\Diamond$ is commutative then $\Diamond_*$ is commutative iff $*$ is.

(2) If $\Diamond$ is associative and $*$ is complete then $\Diamond_*$ is associative iff $*$ is.

(3) If $*$ is cloos-based then

$$(\bigsqcup_{i\in I} Q_i)\Diamond_* R = \bigsqcup_{i\in I}(Q_i \Diamond_* R) \text{ and } Q\Diamond_*(\bigsqcup_{i\in I} R_i) = \bigsqcup_{i\in I}(Q\Diamond_* R_i)$$

for an index set I and relations Q, Q_i, R, R_i whenever the application of $\Diamond_*$ is defined.

Finally, we have the following important property concerning conversion.

Theorem 2.6.6. *Let ($Sc[\mathcal{G}]$,$*$,e,z) be a closg. Then we have $(Q;_* R)^\smile = R^\smile;_* Q^\smile$.*

At the end we want to mention that the proofs of the results provided in this section demand considerably more effort than shown here. For the interested reader we refer to [11] and [13] for a full mathematical deduction.

Chapter 3

Extending RATH

In this chapter we want to extend the RATH library by providing a suitable structure for Goguen categories. Since Goguen categories are made to handle $\mathcal{L}$-fuzzy relations, we additionally have to provide modules for lattices and $\mathcal{L}$-fuzzy relations.
We first want to introduce suitable data structures and test routines for handling lattices. After that, we provide a module collection for Goguen categories. This is strongly oriented at the already existing modules of RATH. Finally, we introduce a Haskell module for $\mathcal{L}$-fuzzy relations and, hence, are able to deliver the standard model of Goguen categories.

3.1 A module collection for lattices

As we are going to treat $\mathcal{L}$-fuzzy relations within the algebraic structure of Goguen categories, we need a module collection for lattices. The representation of an $\mathcal{L}$-fuzzy relation R by a mapping $R : A \times B \to \mathcal{L}$ makes this even clearer. But, there is another connection. From the definitions of the relational categories of Section 2.5 we see that the morphism classes between two objects of a given relational category (except the pure category) form at least a lower semilattice. Thus, the morphisms of a Dedekind category, for example, constitute a complete Brouwerian lattice. We do not aim at a comprehensive package for lattices, but provide the most necessary operations like the determination of the irreducible elements and atoms, respectively. Another important point will be the consistency tests for lattices and lattice morphisms. Always consider that we see this module collection as an “add on” to RATH. Thus, many design decisions are made with respect to conformance. Nevertheless, we do not use any functionality of RATH within these modules so that they can be used as a stand-alone package.

We give two different approaches. The first one uses multi parameter type classes which allows an elegant parametrization and typing but has the drawback that it is no Haskell 98 standard. The second one uses record data structures (Haskell 98 standard) as a compensation of Haskell's relatively weak module system. Thus, we have consistency in using our lattice modules together with RATH.

First, we want to have a look at the type classes for lattices. In accordance to Definition 2.3.2 and Theorem 2.3.2, two approaches are possible. We choose to provide a hierarchy starting with partially ordered sets and then add lattice operations successively. Of course, the induced ordering and join/meet can be mutually computed by standard operations, but we let the user the chance to define more efficient operations.

```
module LatticeClass where

class POrderedSet set el where
    isElem    :: set -> el -> Bool
    elements  :: set -> [el]
    lEq       :: set -> el -> el -> Bool
```

As we are going to use our lattice module together with RATH later on, we have to take care of the name spaces to avoid longish qualified names. Thus, we speak of elements rather than of objects. Furthermore, the ordering is denoted by "lower or equal" (`lEq`) instead of `incl`.

The next step is to provide the sup and inf operation, respectively. This results in upper resp. lower semilattices.

```
class POrderedSet lat el => UpSemiLattice lat el where
    sup     :: lat -> el -> el -> el
    topEl   :: lat -> el

class POrderedSet lat el => LoSemiLattice lat el where
    inf      :: lat -> el -> el -> el
    botEl    :: lat -> el
    atomS    :: lat -> [el]
```

We aim to compute finite (thus, complete) lattices. According to Lemma 2.3.1 we therefore have a least and a greatest element, respectivly, denoted by `botEl` and `topEl`, respectively.

Furthermore, every non-trivial lower semilattice with least element contains at least one atom. As the atoms reflect the whole structure in some kinds of lattices (e.g. Boolean, cf. Section 2.3.7), they play an important role with the consistency tests later on.

The next step is a type class for (standard) lattices.

```
class (UpSemiLattice lat el, LoSemiLattice lat el) => Lattice lat el where
    jIrredS  :: lat -> [el]
    mIrredS  :: lat -> [el]
```

We do not have to provide any new operations. But, in a finite lattice, all elements can be computed by the disjunction/conjunction of a finite number of join-/meet-irreducible elements (cf. Theorem 2.3.5, Definition 2.3.7). Thus, every non-trivial lattice with least/greatest element contains at least one join-/meet-irreducible element. Again, irreducible elements will play an important role with the consistency tests and several applications later on.

As we saw in the section about relational categories, Brouwerian lattices are essential. Within this kind of lattice, we have a new operation for the computation of the relative pseudo complements. Therefore, we define the following type class.

```
class (LoSemiLattice lat el) => RelCompLattice lat el where
    relComplem :: lat -> el -> el -> el
```

Notice, that we deliberately do not call the type class "BrouwerianLattice" or similar because the operation `relComplem` can be interesting even if the relative pseudo complements do not exist for *every* pair of elements. We even do not demand that `lat` and `el` form a lattice because this is not necessary for the definition.

The even stronger notion of a complement (cf. Definition 2.3.15) is taken into account by the next type class.

```
class Lattice compLat el => CompLattice compLat el where
    complem :: compLat -> el -> el
```

Again, we avoid designations like "BooleanLattice".

Hence, we have the type class hierarchy shown in Figure 3.1.

The next step is to provide data structures which respect the Haskell 98 standard. We then use them to parametrize our type classes so that the user is free to choose the variant he prefers. Furthermore, several consistency tests and standard operations shall be available within this module.

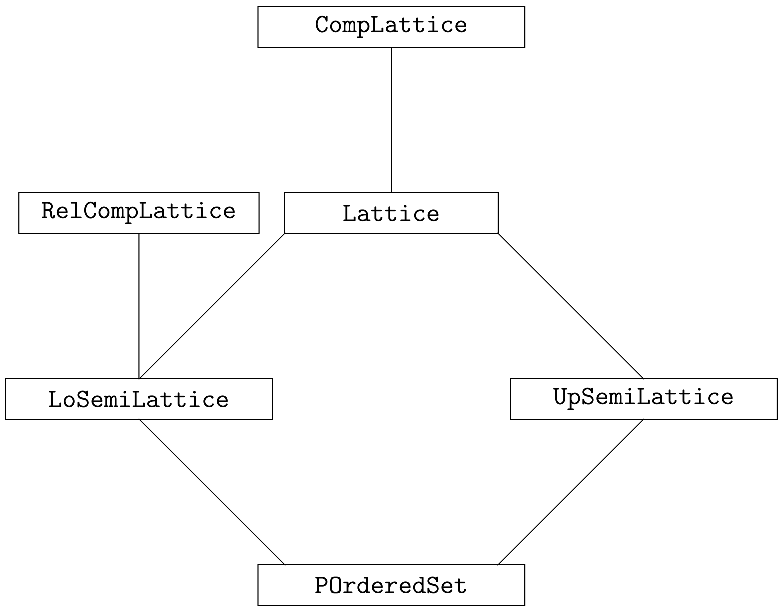

Figure 3.1: The hierarchy of the lattice type classes

```
module Lattice where
import List ((\\),nub,intersect,union)
```

We start with the explanation of the provided data structures in accordance to the type classes. Again, the weakest structures are posets.

```
data PoSet    el = PoSet { poSet_isElem   :: el -> Bool
                         ,poSet_elements :: [el]
                         ,poSet_lEq      :: el -> el -> Bool }
```

As shown here, we parametrize all record data types by the type of the elements. The functions are always analogous to the corresponding type class.

A particularity of this approach can be seen with the next data structure.

```
data USemiLat el = USemiLat { uSemiLat_poSet    :: PoSet el
                            ,uSemiLat_sup      :: el -> el -> el
                            ,uSemiLat_topEl    :: el }
```

We do not have any inheritance as with the type classes. Thus, we have to carry the functions of the underlying poset by an explicit function (`uSemiLat_poSet`). To avoid longish names for accessing the poset functions, we (in analogy to RATH) introduce abbreviated notations.

```
uSemiLat_isElem   = poSet_isElem   . uSemiLat_poSet
```

```
uSemiLat_elements = poSet_elements . uSemiLat_poSet
uSemiLat_lEq      = poSet_lEq      . uSemiLat_poSet
```

The following data structures are constructed analogously.

```
data LSemiLat el = LSemiLat { lSemiLat_poSet    :: PoSet el
                             ,lSemiLat_inf      :: el -> el -> el
                             ,lSemiLat_botEl    :: el
                             ,lSemiLat_atomS    :: [el] }

lSemiLat_isElem   = poSet_isElem   . lSemiLat_poSet
lSemiLat_elements = poSet_elements . lSemiLat_poSet
lSemiLat_lEq      = poSet_lEq      . lSemiLat_poSet

data RelCompLat el = RelCompLat { relCompLat_lSemiLat   :: LSemiLat el
                                 ,relCompLat_relComplem :: el -> el -> el  }

relCompLat_poSet    = lSemiLat_poSet    . relCompLat_lSemiLat
relCompLat_isElem   = lSemiLat_isElem   . relCompLat_lSemiLat
relCompLat_elements = lSemiLat_elements . relCompLat_lSemiLat
relCompLat_lEq      = lSemiLat_lEq      . relCompLat_lSemiLat
relCompLat_inf      = lSemiLat_inf      . relCompLat_lSemiLat
relCompLat_botEl    = lSemiLat_botEl    . relCompLat_lSemiLat
relCompLat_atomS    = lSemiLat_atomS    . relCompLat_lSemiLat

data CompLat el = CompLat { compLat_lat     :: Lat el
                           ,compLat_complem :: el -> el }

compLat_isElem   = lat_isElem   . compLat_lat
compLat_elements = lat_elements . compLat_lat
compLat_lEq      = lat_lEq      . compLat_lat
compLat_sup      = lat_sup      . compLat_lat
compLat_inf      = lat_inf      . compLat_lat
compLat_topEl    = lat_topEl    . compLat_lat
compLat_botEl    = lat_botEl    . compLat_lat
compLat_atomS    = lat_atomS    . compLat_lat
compLat_jIrredS  = lat_jIrredS  . compLat_lat
compLat_mIrredS  = lat_mIrredS  . compLat_lat
compLat_poSet    = lat_poSet    . compLat_lat
```

The construction of standard lattices using upper and lower semi lattices, respectively, cannot be transferred that easily from the type classes. Obviously, we could get consistency problems because the data structures `LSemiLat` and `USemiLat` each carry their own poset. Therefore, we choose to redefine all functions explicitly.

```
data Lat el = Lat{ lat_poSet   :: PoSet el
                  ,lat_sup     :: el -> el -> el
                  ,lat_inf     :: el -> el -> el
                  ,lat_topEl   :: el
                  ,lat_botEl   :: el
                  ,lat_atomS   :: [el]
                  ,lat_jIrredS :: [el]
                  ,lat_mIrredS :: [el] }
```

Thus, the consistency problem is solved, but we have the new problem that the corresponding upper and lower semilattice, respectively, cannot be extracted that comfortably. Hence, we provide the functions

```
lat_uSemiLat :: Lat el -> USemiLat el
lat_uSemiLat l = USemiLat { uSemiLat_poSet = lat_poSet l
                           ,uSemiLat_sup   = lat_sup   l
                           ,uSemiLat_topEl = lat_topEl l }

lat_lSemiLat :: Lat el -> LSemiLat el
lat_lSemiLat l = LSemiLat { lSemiLat_poSet = lat_poSet l
                           ,lSemiLat_inf   = lat_inf   l
                           ,lSemiLat_botEl = lat_botEl l
                           ,lSemiLat_atomS = lat_atomS l }
```

for the user. Notice, that the structure `Lat` carries the set of join and meet-irreducible elements in addition to the functions for upper and lower semilattices, respectively. With the abbreviations

```
lat_isElem   = poSet_isElem   . lat_poSet
lat_elements = poSet_elements . lat_poSet
lat_lEq      = poSet_lEq      . lat_poSet
```

the definition of our data structures is complete. Obviously, they directly correspond to the type class hierarchy of Figure 3.1 so that we are able to provide suitable instances later on.

The definition of the record data types above automatically provides the projections from stronger structures to weaker ones (e.g., from `Lat` to `LSemiLat`). But, there are also embeddings from the weak structures into stronger ones. In the following we provide standard implementations of these constructions. We first introduce some auxiliary functions.

```
partialExt f x []     = x
partialExt f x (y:ys) = if f x y then partialExt f y ys
                        else partialExt f x ys

partialSupInf f els = \x y -> let bds = [e|e<-els,f x e,f y e]
                              in if null bds then x
                                 else partialExt (flip f) (head bds) (tail bds)
```

First of all, `partialExt` determines the unique maximum/minimum of a given poset. With this, the parameter `f` is the ordering on the elements given within the list `y:ys`. If `f` is given as $\leq$, the maximum is computed. If the reversed ordering $\geq$ is given, `partialExt` determines the minimum.
The function `partialSupInf` works analogously by computing the supremum/infimum of all elements of `els` (if it exists) for the given ordering $\leq/\geq$. Thus, `partialSupInf` already delivers two standard constructions and can be used by the following routines.

```
poSet_uSemiLat :: PoSet el -> USemiLat el
poSet_uSemiLat p@(PoSet isElem els lEq) =
  USemiLat { uSemiLat_poSet = p
            ,uSemiLat_sup   = partialSupInf lEq els
            ,uSemiLat_topEl = partialExt lEq (head els) (tail els) }

poSet_lSemiLat :: PoSet el -> LSemiLat el
poSet_lSemiLat p@(PoSet isElem els lEq) =
  let lSL = LSemiLat {
              lSemiLat_poSet = p
             ,lSemiLat_inf   = partialSupInf (flip lEq) els
             ,lSemiLat_botEl = partialExt (flip lEq) (head els) (tail els)
             ,lSemiLat_atomS = atomSet lSL }
  in lSL

uSemiLat_lSemiLat = poSet_lSemiLat . uSemiLat_poSet
lSemiLat_uSemiLat = poSet_uSemiLat . lSemiLat_poSet

lSemiLat_relCompLat :: LSemiLat el -> RelCompLat el
lSemiLat_relCompLat l@(LSemiLat p@(PoSet _ els lEq) inf _ _) =
  RelCompLat { relCompLat_lSemiLat   = l
              ,relCompLat_relComplem =
                  \x y -> let ok = [e|e<-els,lEq (inf x e) y]
                          in if null ok then y
                             else partialExt lEq (head ok) (tail ok) }

poSet_relCompLat :: PoSet el -> RelCompLat el
poSet_relCompLat = lSemiLat_relCompLat . poSet_lSemiLat

uSemiLat_lat :: Eq el => USemiLat el -> Lat el
uSemiLat_lat (USemiLat p@(PoSet _ els lEq) sup top) =
  let lat = Lat { lat_poSet   = p
                 ,lat_sup     = sup
                 ,lat_inf     = partialSupInf (flip lEq) els
                 ,lat_topEl   = top
                 ,lat_botEl   = partialExt (flip lEq) (head els) (tail els)
                 ,lat_atomS   = atomSet $ lat_lSemiLat lat
                 ,lat_jIrredS = jIrredSet lat
                 ,lat_mIrredS = mIrredSet lat }
  in lat

lSemiLat_lat :: Eq el => LSemiLat el -> Lat el
lSemiLat_lat (LSemiLat p@(PoSet _ els lEq) inf bot atomS) =
  let lat = Lat { lat_poSet   = p
                 ,lat_sup     = partialSupInf lEq els
                 ,lat_inf     = inf
                 ,lat_topEl   = partialExt lEq (head els) (tail els)
                 ,lat_botEl   = bot
                 ,lat_atomS   = atomS
                 ,lat_jIrredS = jIrredSet lat
                 ,lat_mIrredS = mIrredSet lat }
  in lat

poSet_lat :: Eq el => PoSet el -> Lat el
poSet_lat = uSemiLat_lat . poSet_uSemiLat

relCompLat_lat :: Eq el => RelCompLat el -> Lat el
relCompLat_lat = lSemiLat_lat . relCompLat_lSemiLat
```

The implementation is straightforward. The only new construction is provided within `lSemiLat_relCompLat`. It is a direct realization of Definition 2.3.13.
Notice that these constructions strongly make use of the fact that we are dealing with finite, and, hence, complete structures. The correctness of the resulting data structures is no question that is to answer here. It has to be tested using the test routines introduced below. Finally, we remain to provide embeddings into complementary lattices.

```
lat_compLat :: Lat el -> CompLat el
lat_compLat l =
  CompLat { compLat_lat     = l
           ,compLat_complem = flip (relCompLat_relComplem
                                    (lSemiLat_relCompLat $ lat_lSemiLat l)) $
                                    lat_botEl l }

poSet_compLat :: Eq el => PoSet el -> CompLat el
poSet_compLat = lat_compLat . poSet_lat

lSemiLat_compLat :: Eq el => LSemiLat el -> CompLat el
lSemiLat_compLat = lat_compLat . lSemiLat_lat

uSemiLat_compLat :: Eq el => USemiLat el -> CompLat el
uSemiLat_compLat = lat_compLat . uSemiLat_lat

relCompLat_compLat :: Eq el => RelCompLat el -> CompLat el
relCompLat_compLat l@(RelCompLat lSL rc) =
  CompLat { compLat_lat     = relCompLat_lat l
           ,compLat_complem = flip rc $ lSemiLat_botEl lSL }
```

Again, the implementation is straightforward. The embedding from `RelCompLat` into complementary lattices is trivial since we already have a function for determining relative pseudo complements. With all other functions, we have to use the standard construction from above.

Thus, we are ready to switch to operations on lattices and the test functions. As we are only dealing with complete lattices, fixpoint operations become important. From Theorems 2.3.12 and 2.3.13 we know that the fixpoints of a monotone function $f : \mathcal{L} \to \mathcal{L}$ over a complete lattice $\mathcal{L}$ form a sublattice of $\mathcal{L}$. Furthermore, we know that f has at least one fixed point. The applications using this fact are manyfold. Therefore, we provide the following functions.

```
leastFPA ::  Eq a => a -> (Lat a) -> (a -> a) -> a
leastFPA a l f = if lat_lEq l a (f a) then leastFPA' a
                 else leastFPA' $ foldl1 (lat_inf l) [x|x<-lat_elements l,
                                                        lat_lEq l a (f x)]
  where
    leastFPA' b = if (f b) == b then b
                  else leastFPA' (f b)

leastFP ::  Eq a => (Lat a) -> (a -> a) -> a
leastFP l = leastFPA (lat_botEl l) l

latFPs :: Eq a => (Lat a) -> (a -> a) -> (Lat a)
latFPs l f = let els = [x | x <- lat_elements l, f x == x ]
             in Lat { lat_poSet = PoSet { poSet_isElem   = flip elem $ els
                                         ,poSet_elements = els
                                         ,poSet_lEq      = lat_lEq l }
                     ,lat_sup     = lat_sup l
                     ,lat_inf     = lat_inf l
                     ,lat_botEl   = foldl (lat_inf l) (lat_topEl l) els
                     ,lat_topEl   = foldl (lat_sup l) (lat_botEl l) els
                     ,lat_atomS   = atomSetBy (lat_poSet l) (lat_botEl l)
                                               (lat_inf l) []
                     ,lat_jIrredS = irredSetBy (lat_poSet l) (lat_botEl l)
                                                (lat_sup l) []
                     ,lat_mIrredS = irredSetBy (lat_poSet l) (lat_topEl l)
                                                (lat_inf l) [] }
```

First of all, `leastFPA` determines the least fixpoint of the monotone function `f`, which is greater or equal to a given element `a` from the underlying lattice `l`. Furthermore, `leastFP` delivers the standard variant by computing the least fixpoint of `f`. The lattice of all fixpoints of `f` is generated by `latFPs`. Notice that we cannot rely on the standard `Prelude` functions `minimum` resp. `maximum` to determine the least resp. greatest element within the list of fixed points since these elements do not necessarily exist. Obviously, the correct termination of all these functions above strongly depends on whether `f` is monotonic. This property can be tested by the function `testMonoFunc` introduced later on. The routines for determining the atoms and irreducible elements, respectively, are also explained later.

Now, our attention goes to the consistency tests. As we are dealing with potentially big lattices, some special thoughts go to efficiency considerations. As mentioned above, in a finite lattice every element can be computed (or partitioned) by a set of irreducible elements. In a distributive lattice this representation is even unique. Atoms play an analogous role within Boolean lattices. The next lemma shows that all tests in a standard/Boolean lattice can be reduced to the irreducible elements/atoms.

Lemma 3.1.1. *Let $\mathcal{L}$ be a set. and $\vee : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ and its extension $\bigvee$ be an operation on $\mathcal{L} \times \mathcal{L}$. If there is an inclusionminimal subset S of $\mathcal{L}$ such that every element $x \in \mathcal{L}$ can be computed by $\bigvee S'$ for a subset S' of S, we have :*

(1) If $\vee$ reduced to S is commutative/associative/idempotent then it is commutative/associative/idempotent in $\mathcal{L}$.
(2) If $\vee$ reduced to S is commutative and idempotent then it is idempotent in $\mathcal{L}$.

The proof is obvious and, therefore, omitted.

Since atoms are join-irreducible elements and in an atomic lattice these two notions are even equivalent, it would suffice to provide a function that computes the irreducible elements. But, atoms are defined via the meet operation and join-irreducible elements via the join operation. One of these operations can be evidently more inefficient than the other.

Thus, we provide both — standard functions to compute the atoms and irreducible elements, respectively. We start with the function `atomSetBy`.

```
atomSetBy :: PoSet el -> el -> (el -> el -> el) -> [el] -> [el]
atomSetBy p botEl inf els =
  let
    li = if null els then atomSet' [] (poSet_elements p) 0
         else atomSet' [] els 0
  in [head el | el <- map (\x -> filter (\y -> (poSet_lEq p) y x) li) li,
      null $ tail el]
  where
    atomSet' at []  n = if n>0 then atomSet' [] at 0 else at
    atomSet' at [a] n = if n>0 then atomSet' [] (a:at) 0 else (a:at)
    atomSet' at (a1:(a2:els)) n
               | (poSet_lEq p) a1 botEl          = atomSet' at (a2:els) (n+1)
               | (poSet_lEq p) a2 botEl          = atomSet' at (a1:els) (n+1)
               | (poSet_lEq p) a1 a2             = atomSet' (a1:at) els (n+1)
               | (poSet_lEq p) a2 a1             = atomSet' (a2:at) els (n+1)
               | (poSet_lEq p) (inf a1 a2) botEl = atomSet' (a1:(a2:at)) els n
               | otherwise                       = atomSet' at els (n+1)
```

It is parametrized by the underlying poset `p`, the least element `botEl`, the meet operation `inf` and a list `els` of elements to which the computation shall be reduced (e.g., a sublattice). If `els` is empty, all atoms of the underlying poset are computed.
The parametrization comes due to the intent to make it comfortable for the user to compute the atoms directly when he creates the data structure for the corresponding (lower semi) lattice. Thus, a lower semilattice for the divisibility relation could be created by

```
divLat = let pS = PoSet { poSet_isElem   = flip elem $ [1,2,3,4,6,12]
                         ,poSet_elements = [1,2,3,4,6,12]
                         ,poSet_lEq      = \x y -> mod y x == 0 }
         in LSemiLat { lSemiLat_poSet = pS
                      ,lSemiLat_inf   = gcd
                      ,lSemiLat_botEl = 1
                      ,lSemiLat_atomS = atomSetBy pS 1 gcd [] }
```

Alternatively, the user can assign the empty list to `lSemiLat_atomS`. For this case, we provide the standard function

```
atomSet :: LSemiLat el -> [el]
atomSet l = atomSetBy (lSemiLat_poSet l) (lSemiLat_botEl l) (lSemiLat_inf l) []
```

which can be used later on to compute the atoms "on demand". As shown here, the extended parametrized variant of any function will be ended by `By` throughout the module.
Back to the function above, the atoms are computed by filtering out those elements $\neq 0$ that have exactly one element in the lattice that is lower or equal to them. This computation tends to be rather extensive. Hence, we reduce the list of possible atoms before we filter them out. This is done by calling `atomSet'` which establishes the key functionality. It successively constructs the list of atoms `at` by local comparison of the first two elements in the element list. The element list steadily becomes shorter. To detect when the element list has been traversed without causing any changes in the atom list, we use the parameter `n`. If it is zero and the list of remaining elements is empty, the computation is completed. Obviously, the result of `atomSet'` strongly depends on the arrangement of the elements within the element list. Thus, it does not necessarily deliver the set of atoms so that they have to be filtered out after `atomSet'` is finished. Consider the following example computation for the lattice induced by the powerset $\mathbb{P}(\{1,2,3\})$. If we, for example, have the element list `[[1],[1,2],[2],[2,3],[3],[1,3],[],[1,2,3]]` and the underlying poset `pS`, calling `atomSetBy pS [] intersect []` results in

```
  atomSet' []            [[1],[1,2],[2],[2,3],[3],[1,3],[],[1,2,3]] 0
```

`atomSet' [[1]]`	`[[2],[2,3],[3],[1,3],[],[1,2,3]]`	`1`	(guard 3)
`atomSet' [[1],[2]]`	`[[3],[1,3],[],[1,2,3]]`	`2`	(guard 3)
`atomSet' [[1],[2],[3]]`	`[[],[1,2,3]]`	`3`	(guard 3)
`atomSet' [[1],[2],[3]]`	`[[1,2,3]]`	`4`	(guard 1)
`atomSet' []`	`[[1,2,3][1],[2],[3]]`	`0`	(since n>0)
`atomSet' [[1]]`	`[[2],[3]]`	`1`	(guard 4)
`atomSet' [[1],[2],[3]]`	`[]`	`1`	(guard 5).

Since `n` is 1, the list is again traversed but no changes take place so that we omit these function calls. In this example, we use an arrangement of the elements such that `atomSet'` does indeed deliver the atoms. But, consider the list `[[1],[2,3],[2],[1,3],[3],[1,2],[],[1,2,3]]`. Obviously, here only the least element `[]` is taken out. Thus, the final filtering in `atomSet` is necessary.

Now, we switch to the determination of the irreducible elements. Since their definition is very inefficient for computational purposes, we take the detour over the reducible elements.

```
redSetBy :: (Eq el) => PoSet el -> (el -> el -> el) -> [el] -> [el]
redSetBy p sup_inf els =
  let li = if null els then poSet_elements p
           else els
  in nub $ redSet' [] (takeN 2 li)
  where
    redSet' red []     = red
    redSet' red (x:xs) = let hX   = head x
                             lX   = last x
                             s    = sup_inf hX lX
                         in if s==hX || s==lX then redSet' red xs
                                               else redSet' (s : red) xs
```

The parametrization is analogous to `atomSet`, but we here have the opportunity to compute either the join reducible or the meet reducible elements. Thus, we call the corresponding parameter `sup_inf`. First, we compute all pairs of two elements by means of the function `takeN` and then call `redSet'`.

```
takeN :: Int -> [a] -> [[a]]
takeN _ []        = []
takeN 1 list      = [[x] | x<-list]
takeN n (l:list) = [l:li | li<-takeN (n-1) list]++takeN n list
```

Here, the sup/inf of any pair of elements is computed and compared to these elements. If it is not equal to any of these two elements, it is reducible (to these elements). Notice, that `takeN` can only be used for commutative operations since it only delivers pairs of elements without considering their arrangement. Calling `takeN 2 [1,2,3]`, for instance, results in the list `[[1,2],[1,3],[2,3]]`. If we have a non-commutative operation that shall be applied to *all* pairs of elements, we thus have to use list comprehension or equivalent constructs.

With the reducible elements we automatically have the irreducible elements.

```
irredSetBy :: (Eq el) => PoSet el -> el -> (el -> el -> el) -> [el] -> [el]
irredSetBy p bot_topEl sup_inf els = let li = if null els then poSet_elements p
                                                else els
                                         in li \\ (bot_topEl:(redSetBy p sup_inf li))
```

The only thing we have to mind is to take out the least/greatest element since it is not join/meet-irreducible by definition. Notice that we need the least element and the join operation to compute the join-irreducible elements and the greatest element and the meet operation to compute the meet-irreducible elements.
Again, we have standard versions for calling the functions above.

```
jRedSet,mRedSet,jIrredSet,mIrredSet :: (Eq el) => Lat el -> [el]
jRedSet l   = redSetBy   (lat_poSet l) (lat_sup l) []
mRedSet l   = redSetBy   (lat_poSet l) (lat_inf l) []

jIrredSet l = irredSetBy (lat_poSet l) (lat_botEl l) (lat_sup l) []
mIrredSet l = irredSetBy (lat_poSet l) (lat_topEl l) (lat_inf l) []
```

Notice that the determination of the atoms and irreducible elements, respectively, causes quadratic runtime costs in the worst case scenario. With `atomSet` it is possible that only one element is taken out with every pass through the element list. With `redSet` we have to generate all pairs of elements, which naturally results in quadratic computation times. But, consider that the average case for `atomSet` should be evidently better than for `irredSetBy` / `redSetBy` (provided that join and meet have comparable computation times). This comes due to the fact that with `irredSetBy` the worst case scenario is created a priori by computing all pairs of elements.

Now, we can proceed with the test functions. Before we start, we want to mention that the following data structures are already defined within the RATH module `RelAlg` to test category theoretical properties. We redefine them so that we have both — conformance to RATH and a stand alone module `Lattice`.

Of course, we want to know when a test failed and why it failed. Thus, the result of a failed test can be represented by the data structure

```
type Result el = (String,[el])
```

which holds the reason for the failure and a list of the affected elements. From the definitions of the lattice data structures, extensive list operations (in particular, the concatenation operator `(++)` to combine several tests) are to expect. As lists are defined recursively, their concatenation is very slow. In accordance to RATH, we use the standard escape. We "introduce" a new concatenation operator. Consider the lists `l1=["A","B"]` and `l2=["C","D"]`. If we represent them by the functions

```
l1,l2 :: [String] -> [String]
l1 = let
       f1 = ("A":)
       f2 = ("B":)
     in f1 . f2
l2 = let
       f3 = ("C":)
       f4 = ("D":)
     in f3 . f4
```

we have the composition operator `(.)` as our new list concatenation. The resulting list `l1++l2` then can be computed by `(l1.l2) []`. This motivates the definition of

```
type TestRes el = [Result el] -> [Result el]
```

which represents a list of failed tests. Furthermore, we need to compute a list of test results later on. This can be done by extending the predefined function `foldr` to `foldF` (fold function)

```
foldF :: [a -> a] -> a -> a
foldF l r = foldr id r l
```

which we here use at the type `[TestRes el] -> TestRes el`. Last but not least, we provide a function `testP` (test predicate)

```
testP :: Bool -> [el] -> String -> TestRes el
testP p els s = \res -> if p then res else (s,els) : res
```

to generate a parametrized test result, if the predicate `p` is `False`. Thus, we have all necessary data structures so that we now can proceed with the single consistency tests.

First, the question arises which tests are needed and how to parametrize them. Because of the two approaches to lattices, we provide order theoretical tests as well as algebraic ones. The three functions

```
testRefl, testTrans :: PoSet el -> TestRes el
testRefl     p  = foldF [testP (poSet_lEq p o o) [o] "Reflexivity:"
                        | o <- poSet_elements p]

testTrans    p  = let incl = poSet_lEq p
                      els  = poSet_elements p
                  in foldF [testP (not (incl x1 x2 && incl x2 x3) || incl x1 x3)
                                  [x1,x2,x3] "Transitivity:"
                           | x1 <- els, x2 <- els, x3<-els]

testAntiSymm :: (Eq el) => PoSet el -> TestRes el
testAntiSymm p  = let incl = poSet_lEq p
                  in foldF [let x1 = head els
                                x2 = last els
                           in testP (not (incl x1 x2 && incl x2 x1) || x1==x2)
                                    [x1,x2] "Antisymmetry:"
                           | els <- takeN 2 $ poSet_elements p]
```

together with the all-in-one test

```
testPoSet :: (Eq el) => PoSet el -> TestRes el
testPoSet p = testRefl p . testTrans p . testAntiSymm p
```

constitute the order theoretical part by testing reflexivity, transitivity and antisymmetry of a poset. Notice that `testTrans` does not use `takeN` because the arrangement of the three elements in question is essential. In contrast, `testAntiSymm` can be supplied by `takeN`.

The algebraic tests are quite more extensive. Because of the symmetric definition of (algebraic) lattices, we use an at first sight somehow unnatural parametrization. Let us have a look at the first test, which verifies that a poset is closed under a binary operation (e.g. join, meet, relative pseudo complement).

```
testConsBy :: (Eq el) => PoSet el -> (el -> el -> el) -> [el] -> [el] ->
                         TestRes el
testConsBy p op els targetEls =
  foldF [testP ((flip elem $ targetEls)$op (head xs) (last xs)) xs "Consistency:"
         |  xs<-takeN 2 $ els]
```

The function first of all needs the underlying poset `p`. Furthermore, the operation `op` that shall be tested is necessary. The third parameter `els` can be used to reduce the test to the elements given in it, i.e., to set the domain of `op`. In contrast to `atomSetBy` and `irredSetBy`, `els` must not be empty. The tests then are not automatically applied to all elements of the underlying poset. We prefer this variant to avoid longish `if ... then` phrases. Furthermore, we provide a standard variant for each test so that this approach should be acceptable. But, notice that we have to use the `elem` function to determine whether an element is in `els`. This can cause a loss in efficiency when membership within the lattice is decided by a certain predicate and not by checking if it is in the element list. Thus, it is up to the user to decide whether `testConsBy` can be used efficiently.
The last parameter can be used to set the range of `op`. Again, `targetEls` has no default setting if an empty list is given as parameter. The following three examples show how `testConsBy` can be used to achieve certain goals.

(1) If we want to test the join consistency of a lattice `l`, we can reduce the test to the join-irreducible elements whereas the range of the join operation is the set of all elements of the lattice. Thus, we can use the function above by
```
testConsBy (lat_poSet l) (lat_sup l) (lat_jIrredS l) (lat_elements l).
```

(2) Suppose we want to test whether some elements form a sublattice of an arbitrary lattice `l` with respect to the join operation. Then we can use `testConsBy` by
```
testConsBy (lat_poSet l) (lat_sup l) (<elements>) (<elements>).
```

(3) To test whether a lattice `l` is closed under join without reducing the test to any elements, one has to use the default call
```
testConsBy (lat_poSet l) (lat_sup l) (lat_elements l) (lat_elements l).
```

Consider that, if the order theoretic tests as well as the two consistency tests above succeed for join and meet, all remaining algebraic tests for standard lattices are redundant. Nearly the same is true for the opposite. If all algebraic tests (even without the join/meet consistency) succeed, the induced order is indeed an order and, hence, the order theoretic tests need not be performed.
To make the use of the test functions more comfortable, we introduce standard calls to check join/ meet consistency as well as consistency with the relative pseudo complements.

```
testJCons :: (Eq el) => USemiLat el -> TestRes el
testJCons l = testConsBy (uSemiLat_poSet l)    (uSemiLat_sup l)
                         (uSemiLat_elements l) (uSemiLat_elements l)
```

```
testMCons :: (Eq el) => LSemiLat el -> TestRes el
testMCons l = testConsBy (lSemiLat_poSet l)    (lSemiLat_inf l)
                         (lSemiLat_elements l) (lSemiLat_elements l)

testRelCompCons :: (Eq el) => RelCompLat el -> TestRes el
testRelCompCons l = testConsBy (relCompLat_poSet l)    (relCompLat_relComplem l)
                               (relCompLat_elements l) (relCompLat_elements l)
```

The complement is a unary operation, so that `testConsBy` cannot be used. Thus, we introduce a second variant together with a standard call.

```
testConsUnBy :: (Eq el) => PoSet el -> (el -> el) -> [el] -> [el] -> TestRes el
testConsUnBy p op els targetEls =
    foldF [testP ((flip elem) targetEls $ op x) [x] "ConsistencyUn:"
           | x<-els]

testCompCons :: (Eq el) => CompLat el -> TestRes el
testCompCons l = testConsUnBy (compLat_poSet l)    (compLat_complem l)
                              (compLat_elements l) (compLat_elements l)
```

Commutativity, associativity, idempotency and the absorption laws of the join/meet operation can be verified by the following test routines.

```
testCommBy,testIdemBy, testAssBy :: (Eq el) =>
                              PoSet el -> (el -> el -> el) -> [el] -> TestRes el
testCommBy p op els = foldF [let o1   = head os
                                 o2   = last os
                             in testP (op o1 o2 == op o2 o1) os "Commutativity:"
                             | os<-takeN 2 els]

testIdemBy p op els = foldF [testP (op o o == o) [o] "Idempotency:"
                             | o<-els]

testAssBy p op els = foldF [let o1    = head os
                                o2    = head $ tail os
                                o3    = last os
                            in testP ((op o1 $ op o2 o3) == (op (op o1 o2) o3))
                                     os "Associativity:"
                            | os<-takeN 3 els]
testAbsBy :: (Eq el) => PoSet el -> (el -> el -> el) ->
```

```
                                    (el -> el -> el) -> [el] -> TestRes el
testAbsBy p op1 op2 els = foldF [testP ((op1 o1 $ op2 o1 o2)==o1) [o1,o2]
                                 "Absorption:" | o1<-els, o2<-els]
```

They are parametrized analogous to `testConsBy`. Obviously, the list `targetEls` can be left out since it plays no role for these tests. Notice that `testAbsBy` cannot rely on `takeN`. Again, the standard variants are provided by the following functions.

```
testJComm,testJIdem, testJAss :: (Eq el) => USemiLat el -> TestRes el
testJComm l= testCommBy (uSemiLat_poSet l) (uSemiLat_sup l) (uSemiLat_elements l)
testJIdem l= testIdemBy (uSemiLat_poSet l) (uSemiLat_sup l) (uSemiLat_elements l)
testJAss  l= testAssBy  (uSemiLat_poSet l) (uSemiLat_sup l) (uSemiLat_elements l)

testMComm,testMIdem, testMAss :: (Eq el) => LSemiLat el -> TestRes el
testMComm l= testCommBy (lSemiLat_poSet l) (lSemiLat_inf l) (lSemiLat_elements l)
testMIdem l= testIdemBy (lSemiLat_poSet l) (lSemiLat_inf l) (lSemiLat_elements l)
testMAss  l= testAssBy  (lSemiLat_poSet l) (lSemiLat_inf l) (lSemiLat_elements l)

testJAbs,testMAbs :: (Eq el) => Lat el -> TestRes el
testJAbs  l= testAbsBy (lat_poSet l) (lat_sup l) (lat_inf l) (lat_elements l)
testMAbs  l= testAbsBy (lat_poSet l) (lat_inf l) (lat_sup l) (lat_elements l)
```

Furthermore, we need two functions to verify the correctness of the greatest and least element, respectively.

```
testTopElBy :: USemiLat el -> [el] -> TestRes el
testTopElBy l els =
  let top = uSemiLat_topEl l
  in foldF ( testP (uSemiLat_isElem l top) [top] "Top in List:" :
            [testP (uSemiLat_lEq l x top)  [x] "Top element:" | x<- els] )

testBotElBy :: LSemiLat el -> [el] -> TestRes el
testBotElBy l els =
  let bot = lSemiLat_botEl l
  in foldF ( testP (lSemiLat_isElem l bot) [bot] "Bot in List:" :
            [testP (lSemiLat_lEq l bot x)  [x] "Bottom element:" | x<- els] )

testTopEl :: USemiLat el -> TestRes el
testTopEl l = testTopElBy l (uSemiLat_elements l)

testBotEl :: LSemiLat el -> TestRes el
testBotEl l = testBotElBy l (lSemiLat_elements l)
```

Hence, we have accomplished the pure consistency tests for standard (upper/lower semi)lattices. Now, we remain to provide all-in-one tests. This is done by the following functions.

```
testUpSemiLatticeBy :: (Eq el) => USemiLat el -> [el] -> [el] -> TestRes el
testUpSemiLatticeBy  l els targetEls =
      let pS  = uSemiLat_poSet l
          s   = uSemiLat_sup   l
      in testCommBy pS s els . testAssBy  pS s els .
         testIdemBy pS s els . testConsBy pS s els targetEls .
         testTopElBy l els

testLoSemiLatticeBy :: (Eq el) => LSemiLat el -> [el] -> [el] -> TestRes el
testLoSemiLatticeBy  l els targetEls =
      let pS = lSemiLat_poSet l
          i  = lSemiLat_inf   l
      in testCommBy pS i els . testAssBy  pS i els .
         testIdemBy pS i els . testConsBy pS i els targetEls .
         testBotElBy  l els

testLatticeBy       :: (Eq el) => Lat el -> [el] -> [el] ->
                                            [el] -> [el] -> TestRes el
testLatticeBy        l els1 els2 els3 targetEls =
         testUpSemiLatticeBy (lat_uSemiLat l) els1 targetEls  .
         testLoSemiLatticeBy (lat_lSemiLat l) els2 targetEls  .
         testAbsBy (lat_poSet l) (lat_sup l) (lat_inf l) els3 .
         testAbsBy (lat_poSet l) (lat_inf l) (lat_sup l) els3
```

It should be clear from the previous explanations how the first two functions work. The test for lattices is special in the sense that we have three parameters `els1`, `els2` and `els3`, respectively, to reduce the tests for upper and lower semilattices. Good examples for `els1` and `els2` are the join resp. meet-irreducible elements. Notice that it makes no sense to differentiate the parameter `targetEls` since we want to test a lattice structure.

The standard use of the function above is the following.

```
testUpSemiLattice :: (Eq el) => USemiLat el -> TestRes el
testUpSemiLattice l = testUpSemiLatticeBy l
                          (uSemiLat_elements l) (uSemiLat_elements l)
```

```
testLoSemiLattice :: (Eq el) => LSemiLat el -> TestRes el
testLoSemiLattice l = testLoSemiLatticeBy l
                         (lSemiLat_elements l) (lSemiLat_elements l)

testLattice       :: (Eq el) => Lat el -> TestRes el
testLattice       l = testLatticeBy l (lat_jIrredS l)  (lat_mIrredS l)
                                      (lat_elements l) (lat_elements l)
```

Notice that `testLattice` automatically reduces the test of the upper resp. lower semilattice to the join resp. meet-irreducible elements. Of course, this presupposes that the lists are not empty.

The next step is to check modularity, distributivity and atomicity. We start with modularity.

```
testModularBy :: (Eq el) => Lat el -> [el] -> TestRes el
testModularBy l els =
   let sup = lat_sup l
       inf = lat_inf l
   in foldF [testP (not (lat_lEq l x1 x3) || (sup x1 $ inf x2 x3) ==
                   inf (sup x1 x2) x3) [x1,x2,x3] "Modularity:"
             | x1<-els, x2<-els, x3<-els]

testModular :: (Eq el) => Lat el -> TestRes el
testModular l = testModularBy l (lat_elements l)
```

From Lemma 2.3.6 we know that the validities of the modular laws imply each other. Thus, we directly parametrize the function above by the underlying lattice `l`. Notice that we cannot use `takeN` to determine all combinations of any three elements out of `l`, because the arrangement of the three elements in question is essential (cf. Definition 2.3.10). Furthermore, the modularity test can, in general, not be reduced to the atoms or join-irreducible elements, respectively.

The same is true for the following routine which can be used to check distributivity.

```
testDistrBy :: (Eq el) => PoSet el -> (el -> el -> el) ->
               (el -> el -> el) -> [el] -> TestRes el
testDistrBy p op1 op2 els =
    foldF [let x1   = head xs
               x2   = xs!!1
```

```
            x3   = last xs
        in testP ((op1 x1 $ op2 x2 x3) == op2 (op1 x1 x2) (op1 x1 x3))
                 xs "Distributivity:"
        | xs<-takeN 3  els]
```

Again, the validity of one of the distributive laws implies the validity of the other (cf. Lemma 2.3.6) so that the parametrization of this function by the two operations `op1` and `op2` is redundant from a mathematical point of view. But, again efficiency considerations with respect to the join and meet operations drove us to this decision. To determine join distributivity, we use the join operation three times and the meet operation only twice. The opposite is true for meet distributivity. The user now can decide which variant is the most effective. Hence, we provide standard tests for join and meet distributivity, respectively.

```
testJDistr,testMDistr :: (Eq el) => Lat el -> TestRes el
testJDistr l = testDistrBy (lat_poSet l) (lat_sup l) (lat_inf l) (lat_elements l)

testMDistr l = testDistrBy (lat_poSet l) (lat_inf l) (lat_sup l) (lat_elements l)
```

From Theorem 2.3.8 we know that the notions of complete upwards distributivity and complete Brouwerian lattices are equivalent. Furthermore, we can conclude the equivalence of complete upwards and downwards distributivity within finite lattices. Thus, testing distributivity can be substituted by testing the existence of the relative pseudo complements for all pairs of elements and vice versa.

```
testRelComplBy :: (Eq el) => RelCompLat el -> [el] -> TestRes el
testRelComplBy l els =
    foldF [testP ((flip elem) els (relCompLat_relComplem l
                          (head xs) (last xs))) xs "Rel.pseudo compl.:"
           | xs <- takeN 2 els ]

testRelCompl :: (Eq el) => RelCompLat el -> TestRes el
testRelCompl l = testRelComplBy l (relCompLat_elements l)
```

Now, we proceed with atomicity. This test tends to be rather costly with respect to efficiency. Unfortunately, we have no general possibility to delimit it. One approach could be to test whether all join-irreducible elements are atoms. But, the quadratic complexity of determining the join-irreducible elements indicated by `irredSetBy` only permits a gain in efficiency, if the user is able to provide a faster way to compute them. Thus, we deliver two functions — one that tests the join-irreducible elements and one that directly checks whether all elements can be computed by the disjunction of a certain number of atoms.

```
testAtomicIrred :: (Eq el) => Lat el -> TestRes el
testAtomicIrred l =
    let incl   = lat_lEq l
    in foldF [ let x1 = head xs
                   x2 = last xs
               in testP (not (incl x1 x2 || incl x2 x1 ) || x1==x2)
                        xs "Atomicity(irr.el.):"  | xs<-takeN 2 $ lat_jIrredS l]

testAtomicBy :: (Eq el) => Lat el -> [el] -> TestRes el
testAtomicBy l els =
  foldF [let as  = [ a  | a<-lat_atomS l, (lat_lEq l) a x ]
         in if null as then id
            else testP (x==foldl1 (lat_sup l) as) (x:as) "Atomicity:"| x<-els]

testAtomic :: (Eq el) => Lat el -> TestRes el
testAtomic l = testAtomicBy l (lat_elements l)
```

The parametrization of these two functions is a bit different. If the user, for example, wants to test a sublattice of `l`, he has to create a new data structure or to use `testAtomicBy`.
The atomicity test within `testAtomicIrred` is done by checking whether there are two (different) elements such that they are in the underlying order relation. If not, all join-irreducible elements are atoms.
Notice that we leave the user to check whether the atom set and set of irreducible elements, respectively, are correct. This can, for example, be done by comparing them to the result of `atomSetBy` and `irredSetBy`, respectively.

Now, only the complement operation is left. The check is done by the following functions.

```
testComplBy :: CompLat el -> [el] -> TestRes el
testComplBy l els =
  let lEq     = compLat_lEq l
      complem = compLat_complem l
      in foldF [testP (lEq (compLat_topEl l) (compLat_sup l x (complem x)) &&
                       lEq (compLat_inf l x $ complem x) (compLat_botEl l) &&
                       lEq x (complem $ complem x)                         &&
                       lEq (complem $ complem x) x   ) [x] "Complement:"
                | x <- els]

testCompl :: CompLat el -> TestRes el
testCompl l = testComplBy l (compLat_elements l)
```

Obviously, we test whether $x \vee \overline{x} = 1$, $x \wedge \overline{x} = 0$ and $\overline{\overline{x}} = x$ hold.

Thus, we can introduce all-in-one tests for Boolean lattices.

```
testBoolLatticeBy :: (Eq el)=>CompLat el->[el]->[el]->[el]->[el]->TestRes el
testBoolLatticeBy l els1 els2 els3 targetEls =
    testComplBy l els1 . testLatticeBy (compLat_lat l) els2 els3 els1 targetEls

testBoolLattice :: (Eq el) => CompLat el -> TestRes el
testBoolLattice      l =
  let els = compLat_elements l
  in testBoolLatticeBy l els (compLat_atomS l) (compLat_mIrredS l) els
```

With the standard variant we again automatically reduce the tests to the atoms and meet-irreducible elements, respectively, to test the lattice structure.

Finally, our attention goes to morphism tests. According to Definition 2.3.5, we differentiate between (upper/lower semi) lattice and co-lattice morphisms, respectively. Furthermore, morphisms between Brouwerian/Boolean lattices, which respect the relative pseudo complements/complements are of interest. Obviously, all kinds of morphisms have in common that they respect unary or binary operations within their domain and range, respectively. Hence, we can introduce two functions as follows.

```
testMorphBy :: (Eq el2) => PoSet el1 -> PoSet el2 ->
             (el1 -> el1 -> el1) ->
             (el2 -> el2 -> el2) -> String -> (el1 -> el2) -> TestRes el1
testMorphBy l1 l2 opL1 opL2 text f =
    foldF [let x1 = head xs
               x2 = last xs
           in testP ((f $ opL1 x1 x2) == (opL2 (f x1) $f x2)) xs text
           | xs <- takeN 2 $ poSet_elements l1]

testMorphUnBy :: (Eq el2) => PoSet el1 -> PoSet el2 ->
                          (el1 -> el1) -> (el2 -> el2) ->
                          String -> (el1 -> el2) -> TestRes el1
testMorphUnBy l1 l2 opL1 opL2 text f =
    foldF [testP ((f $ opL1 x) == (opL2 $ f x)) [x] text | x <- poSet_elements l1]
```

In both functions `f` is the morphism in question. The only difference is that `testMorphBy` gets binary operations (e.g., *sup*, *inf*) and `testMorphUnBy` gets unary (e.g., complement). The string `text` is used to generate an appropriate output when a test fails.

With help of these two functions, we now can easily provide the required tests.

```
testUpSemiLatMorph :: (Eq el2) =>
                      USemiLat el1 -> USemiLat el2 -> (el1 -> el2) -> TestRes el1
testUpSemiLatMorph   l1 l2   =
    testMorphBy (uSemiLat_poSet l1) (uSemiLat_poSet l2)
              (uSemiLat_sup   l1) (uSemiLat_sup   l2) "USemiLatMorph:"

testLoSemiLatMorph :: (Eq el2) =>
                      LSemiLat el1 -> LSemiLat el2 -> (el1 -> el2) -> TestRes el1
testLoSemiLatMorph   l1 l2   =
    testMorphBy (lSemiLat_poSet l1) (lSemiLat_poSet l2)
              (lSemiLat_inf   l1) (lSemiLat_inf   l2) "LSemiLatMorph:"

testLatMorph       :: (Eq el2) =>
                      Lat el1 -> Lat el2 -> (el1 -> el2) -> TestRes el1
testLatMorph         l1 l2 f =
    testUpSemiLatMorph (lat_uSemiLat l1) (lat_uSemiLat l2) f.
    testLoSemiLatMorph (lat_lSemiLat l1) (lat_lSemiLat l2) f

testUpCoSemiLatMorph :: (Eq el2) =>
                        USemiLat el1 -> LSemiLat el2 -> (el1 -> el2) -> TestRes el1
testUpCoSemiLatMorph l1 l2   =
    testMorphBy (uSemiLat_poSet l1) (lSemiLat_poSet l2)
              (uSemiLat_sup   l1) (lSemiLat_inf   l2) "UCoSemiLatMorph:"

testLoCoSemiLatMorph :: (Eq el2) =>
                        LSemiLat el1 -> USemiLat el2 -> (el1 -> el2) -> TestRes el1
testLoCoSemiLatMorph l1 l2   =
    testMorphBy (lSemiLat_poSet l1) (uSemiLat_poSet l2)
              (lSemiLat_inf   l1) (uSemiLat_sup   l2) "LCoSemiLatMorph:"

testCoLatMorph :: (Eq el2) => Lat el1 -> Lat el2 -> (el1 -> el2) -> TestRes el1
testCoLatMorph       l1 l2 f =
    testUpCoSemiLatMorph (lat_uSemiLat l1) (lat_lSemiLat l2) f .
    testLoCoSemiLatMorph (lat_lSemiLat l1) (lat_uSemiLat l2) f

testBoolLatMorph :: (Eq el2) =>
                    CompLat el1 -> CompLat el2 -> (el1 -> el2) -> TestRes el1
testBoolLatMorph     l1 l2 f =
    testLatMorph  (compLat_lat l1    ) (compLat_lat l2    ) f .
    testMorphUnBy (compLat_poSet l1  ) (compLat_poSet l2  )
                  (compLat_complem l1) (compLat_complem l2) "Complement:" f
```

Special endomorphisms, namely monotone and antitone functions, can be tested using the following routines.

```
testMonoFunc, testAntiFunc :: PoSet el -> (el -> el) -> TestRes el
testMonoFunc l f = let lEq = poSet_lEq l
                   in foldF [let x1 = head xs
                                 x2 = last xs
                             in testP ((not (lEq x1 x2) || lEq (f x1) (f x2)) &&
                                       (not (lEq x2 x1) || lEq (f x2) (f x1)) )
                             xs "Func.Mono :" | xs <- takeN 2 $ poSet_elements l ]

testAntiFunc l f = let lEq = poSet_lEq l
                   in foldF [let x1 = head xs
                                 x2 = last xs
                             in testP ((not (lEq x1 x2) || lEq (f x2) (f x1)) &&
                                       (not (lEq x2 x1) || lEq (f x1) (f x2)) )
                             xs "Func.Anti :" | xs <- takeN 2 $ poSet_elements l ]
```

Obviously, the tests are done componentwise. Notice that the weakest structure in which the two notions above make sense are posets.

Thus, our module `Lattice` is complete and should suffice for our purposes. The only thing left to do, is to connect the class view and the module view by building suitable instances. This is done within the module `LatticeInstances`. A listing can be found in Appendix A.

3.2 Using the lattice module

Now, we want to demonstrate how the module of the last section can be used to instantiate some standard lattices. With these remarks it then should be clear how the module works. Throughout this section, we only instantiate data structures that form lattices or complementary lattices. The corresponding (semi)lattice structures then can be extracted using the standard routines of the last section.

```
module StandardLattices where
import Lattice
import List (delete,nub,union)
import Sets
```

First, we want to create the complete Boolean lattice induced by the powerset of a given set. To do so, we import the `Sets` module. Unfortunately, this module provides no `Eq` instance for the `Set` data structure. Since we need this, we have to implement our own instance.

```
instance (Eq obj,Ord obj) => Eq (Set obj) where
  (==) s1 s2 = let ls1 = sizeSet s1
               in ls1 == (sizeSet s2) &&
                  ls1 == (sizeSet $ intersectSet s1 s2)
```

Furthermore, we need a function that, given a set, returns the powerset.

```
power :: [a] -> [[a]]
power l = power' id l []
 where
  power' f [] = ((f []):)
  power' f (x:xs) = power' f xs . power' (f . (x:)) xs
```

With this sets are represented by a simple element list. Hence, we are able to instantiate the induced Boolean lattice using the `Set` data structure.

```
powerSetLat :: (Eq el,Ord el) => [el] -> CompLat (Set el)
powerSetLat els =
          let els' = map listToSet $ power els
              ats  = map (listToSet.(:[])) $ nub els
              tEl  = listToSet els
          in  CompLat { compLat_lat    =
                        Lat { lat_poSet   =
                               PoSet { poSet_isElem   = flip elem $ els'
                                      ,poSet_elements = els'
                                      ,poSet_lEq      = \x y -> intersectSet
                                                                x y == x }
                             ,lat_sup     = joinSet
                             ,lat_inf     = intersectSet
                             ,lat_topEl   = tEl
                             ,lat_botEl   = zeroSet
                             ,lat_atomS   = ats
                             ,lat_jIrredS = ats
                             ,lat_mIrredS = map (diffSet tEl) ats }
```

```
                                ,compLat_complem = diffSet tEl }
```

The implementation is straightforward and mainly uses the predefined functions of the `Sets` module. Notice that we explicitly provide the atoms and irreducible elements, respectively, to avoid unnecessary computation times.
Now, we have to check whether the lattice is instantiated correctly. First, we exemplary want to have a look at the created elements and the atoms of the powerset lattice induced by the set $\{1, 2, 3, 4\}$. The Hugs session

```
StandardLattices> compLat_elements $ powerSetLat [1..4]
[{},{4},{3},{3, 4},{2},{2, 4},{2, 3},{2, 3, 4},{1},
 {1, 4},{1, 3},{1, 3, 4},{1, 2},{1, 2, 4},{1, 2, 3},{1, 2, 3, 4}]

StandardLattices> compLat_atomS $ powerSetLat [1..4]
[{1},{2},{3},{4}]
```

shows that these elements are generated correctly. The same is true for all other parts of the data structure. Calling the consistency tests

```
StandardLattices> testBoolLattice (powerSetLat [1..4]) []
[]

StandardLattices> testAtomic (compLat_lat $ powerSetLat [1..4]) []
[]
```

makes sure that we indeed have a correctly instantiated Boolean lattice.

The second standard Boolean lattice consists of the truth values with implication $\Rightarrow$ as the ordering and $True$ resp. $False$ as greatest resp. least element. It is the base for constructing Boolean matrices using the `LFuzzyRel` module.

```
logicLat :: CompLat Bool
logicLat = let els = [True,False]
           in  CompLat { compLat_lat    =
                          Lat { lat_poSet   =
                                 PoSet { poSet_isElem   = flip elem $ els
                                        ,poSet_elements = els
                                        ,poSet_lEq      = \x y -> not x || y }
                               ,lat_sup     = (||)
                               ,lat_inf     = (&&)
                               ,lat_topEl   = True
```

```
                              ,lat_botEl   = False
                              ,lat_atomS   = [True]
                              ,lat_jIrredS = [True]
                              ,lat_mIrredS = [False] }
                       ,compLat_complem = not }
```

Obviously, this definition is consistent.

Another important class are linear lattices (i.e., $a \leq b$ or $b \leq a$ holds for every two elements). The unit interval $[0, 1]$, for instance, constitutes such a lattice. We provide the function

```
linLat :: Ord el => [el] -> Lat el
linLat els = let
               mi    = minimum els
               ma    = maximum els
               notMi = delete mi els
             in Lat { lat_poSet   = PoSet { poSet_isElem   = flip elem $ els
                                          ,poSet_elements = els
                                          ,poSet_lEq      = (<=) }
                    ,lat_sup     = max
                    ,lat_inf     = min
                    ,lat_topEl   = ma
                    ,lat_botEl   = mi
                    ,lat_atomS   = [minimum notMi]
                    ,lat_jIrredS = notMi
                    ,lat_mIrredS = delete ma els }
```

which instantiates the lattice in question. Notice that all elements except the least are join-irreducible and all elements except the greatest are meet-irreducible. Again, typing, for example

```
StandardLattices> testLattice (linLat [1,2,3,4,6,12]) []
[]
```

assures the consistency.

The next construction is of a somehow mathematical nature. We provide the lattice induced by the divisibility relation with `lcm` and `gcd` as supremum resp. infimum operation.

```
divLat :: Int -> Int -> Lat Int
divLat lB uB = let
```

```
                els = [ x | x<-[lB..uB],mod uB x == 0]
                mi  = minimum els
                pS  = PoSet { poSet_isElem   = flip elem $ els
                             ,poSet_elements = els
                             ,poSet_lEq      = \x y -> mod y x == 0 }
              in  Lat { lat_poSet   = pS
                       ,lat_sup     = lcm
                       ,lat_inf     = gcd
                       ,lat_topEl   = uB
                       ,lat_botEl   = mi
                       ,lat_atomS   = atomSetBy pS mi gcd []
                       ,lat_jIrredS = irredSetBy pS mi lcm []
                       ,lat_mIrredS = irredSetBy pS uB gcd [] }
```

Since we are dealing with finite structures, the user has to give a lower and upper bound, respectively, which delimit the possible elements of the lattice. We take `uB` as the greatest element so that `lB` is not necessarily the least element of the resulting lattice. It is the least element, if and only if it is a divisor of `uB`. Notice that the atoms of the resulting lattice are exactly the prime factors of `uB` if `lB` is 1.

Finally, we want to implement the non-modular and non-distributive lattice. They are interesting in the way that we can compare whether the modularity resp. distributivity check fail.

```
nonModLat, nonDistrLat :: Lat String
nonModLat =
  let els = ["0","a","b","c","1"]
      irr = ["a","b","c"]
  in Lat { lat_poSet   = PoSet { poSet_isElem   = flip elem $ els
                               ,poSet_elements = els
                               ,poSet_lEq      = \x y -> x=="0" || y=="1" ||
                                                      x==y || (x=="a" && y=="b") }
          ,lat_sup     = \x y -> if x=="0" then y else
                                   if y=="0" then x else
                                     if x==y then x else
                                       if (x=="a")&&(y=="b")||
                                          (x=="b")&&(y=="a") then "b" else "1"
          ,lat_inf     = \x y -> if x=="1" then y else
                                   if y=="1" then x else
                                     if x==y then x else
```

```
                                        if (x=="a")&&(y=="b")||
                                           (x=="b")&&(y=="a") then "a" else "0"
           ,lat_topEl   = "1"
           ,lat_botEl   = "0"
           ,lat_atomS   = ["a","c"]
           ,lat_jIrredS = irr
           ,lat_mIrredS = irr }

nonDistrLat =
  let els = ["0","a","b","c","1"]
      ats = ["a","b","c"]
  in Lat { lat_poSet   = PoSet { poSet_isElem   = flip elem $ els
                               ,poSet_elements = els
                               ,poSet_lEq      = \x y -> x=="0" || y=="1" ||
                                                          x==y }
           ,lat_sup     = \x y -> if x=="0" then y else
                                    if y=="0" then x else
                                      if x==y then x else "1"
           ,lat_inf     = \x y -> if x=="1" then y else
                                    if y=="1" then x else
                                      if x==y then x else "0"
           ,lat_topEl   = "1"
           ,lat_botEl   = "0"
           ,lat_atomS   = ats
           ,lat_jIrredS = ats
           ,lat_mIrredS = ats }
```

The implementations directly correspond to the lattices shown in Figure 2.2 and 2.3. The Hugs session

```
StandardLattices> testModular nonModLat []
[("Modularity:",["a","c","b"])]

StandardLattices> testModular nonDistrLat []
[]

StandardLattices> testJDistr nonDistrLat []
[("Distributivity:",["a","b","c"])]

StandardLattices> testMDistr nonDistrLat []
```

```
[("Distributivity:",["a","b","c"])]
```

indeed shows that `nonModLat` is not modular and `nonDistrLat` is modular but not distributive.

3.3 A general extension for Goguen categories

In Section 2.6, we gave the definition of a Goguen category introduced in [11]. We aim to provide a suitable computer-aided framework for this kind of category by extending the RATH module system. Fortunately, Goguen categories do not make any changes within the RATH modules necessary since the possibility to express crispness is not essential for the hierarchical definition of relational categories. As mentioned before, RATH uses two different views of relational categories — the class view and the module view (brought about by record data structures). To guarantee consistency, we also differentiate here, and provide the modules `GoguenClass`, `Goguen` and `GoguenInstances`.

First, we introduce a new type class for Goguen categories

```
module GoguenClass where

import RelAlgClasses (DedCat)

class DedCat gog obj mor =>
      GoguenCat gog loos obj mor | gog -> loos, gog -> obj, gog -> mor where
    up    :: gog -> mor -> mor
    down  :: gog -> mor -> mor
    derOp :: gog -> String -> loos -> mor -> mor -> mor
```

such that the new operations $^\uparrow$ and $^\downarrow$ (cf. Section 2.6) are supported. Furthermore, derived operations play an important role within Goguen categories to reason over and represent fuzzy controllers. Hence, we provide the function `derOp` which makes it possible to use as many user-defined operations as wanted. The certain operations are identified by a label and are parametrized by a given loos and the two arguments. Notice that the `loos` data structure is still an abstract parameter within this module. We will have to provide a proper definition within our module for $\mathcal{L}$-fuzzy relations later on.

Obviously, in conformance to the definition, our type class for Goguen categories is an extension of Dedekind categories. Thus, we have the new type class hierarchy shown in Figure 3.2.

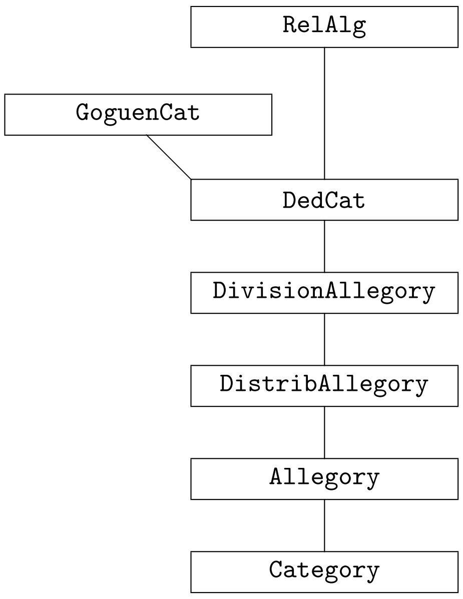

Figure 3.2: Extended type class hierarchy of RATH

The next part is to provide a suitable record data structure as well as consistency tests. This is done within the following module.

```
module Goguen where


import RelAlg
import Lattice
import List (intersect)
import LFuzzyRel
```

Later on, it will become clear why we make use of `Lattice`. The data structure

```
data Gog loos obj mor = Gog { gog_ded   :: Ded obj mor
                            ,gog_up    :: mor -> mor
                            ,gog_down  :: mor -> mor
                            ,gog_derOp :: String -> loos -> mor -> mor -> mor }
```

directly reflects our type class for Goguen categories. Notice, that *standard* definitions for $^{\uparrow}$ and $^{\downarrow}$ that are valid for all Goguen categories do not exist. Hence, the user has to provide these functions. The opposite is true for derived operations. M. Winter in [11] gave a standard definition that operates on all scalars of a given object. This is rather inefficient, and we think that a standard implementation can be omitted. Thus, the user has to provide this routine himself. Later on, we will give an implementation for the standard model within the `LFuzzyRel` module.

For abbreviated notations, we introduce the following functions.

```
gog_isObj    = ded_isObj      .gog_ded
gog_isMor    = ded_isMor      .gog_ded
gog_objects  = ded_objects    .gog_ded
gog_homset   = ded_homset     .gog_ded
gog_source   = ded_source     .gog_ded
gog_target   = ded_target     .gog_ded
gog_idmor    = ded_idmor      .gog_ded
gog_comp     = ded_comp       .gog_ded
gog_converse = ded_converse   .gog_ded
gog_meet     = ded_meet       .gog_ded
gog_incl     = ded_incl       .gog_ded
gog_join     = ded_join       .gog_ded
gog_bottom   = ded_bottom     .gog_ded
gog_rres     = ded_rres       .gog_ded
gog_lres     = ded_lres       .gog_ded
gog_syq      = ded_syq        .gog_ded
gog_top      = ded_top        .gog_ded
gog_divAll   = ded_divAll     .gog_ded
gog_distrAll = divAll_distrAll.gog_divAll
gog_all      = distrAll_all   .gog_distrAll
gog_cat      = all_cat        .gog_all
```

Thus, the extension of RATH with respect to the new data structures is complete. Our attention now goes to the consistency tests. Which checks are necessary, can directly be inferred from Definition 2.6.1. For Property (1) we only have to check whether $\top\!\!\top_{AB} \neq \perp\!\!\perp_{AB}$ holds for all objects A, B. The test for Dedekind categories is already provided by RATH. With property (2) the parts (2a)-(2d) are essential with respect to the $^{\uparrow}$ and $^{\downarrow}$ operation, because they guarantee that our interpretation of these two operations is valid. Part (2e) is fundamental for the α-cut Theorem (2.6.2) and, thus, plays a key role. But, unfortunately, for any two objects A and B of the underlying Dedekind category, we need all antimorphisms between $Sc_{\mathcal{G}}[A]$ and $Crisp_{\mathcal{G}}[A, B]$. Obviously, this computation tends to be very inefficient. The trivial implementation would be to compute all mappings and then test which one is an antimorphism. Of course, this is unacceptable. Therefore, our first thoughts go to possible improvements.

It is clear that there will be no general function that delivers a certain computation formula for every antimorphism. Furthermore, an approach to find certain structural interactions between $Sc_{\mathcal{G}}[A]$ and $Crisp_{\mathcal{G}}[A,B]$ will fail. If we, for example, take the standard Goguen category provided by Theorem 2.6.1, we see that the structure of $Sc_{\mathcal{G}}[A]$ strongly depends on the underlying lattice $\mathcal{L}$. But, the structure of $Crisp_{\mathcal{G}}[A,B]$ only depends on the objects A and B.

Thus, we have to go another way. Again, our rescue is that we are dealing with finite structures. The scalars $Sc_{\mathcal{G}}[A]$ of an arbitrary object A within our Goguen category form a lattice (cf. Lemma 2.5.6). Thus, every scalar can be determined by the corresponding join-irreducible elements and, hence, any mapping $f : \mathcal{L}_1 \to \mathcal{L}_2$ fulfilling

$$f(\bigvee M) = \bigwedge_{x \in M} f(x) \text{ for } M \subseteq \mathcal{L}_1$$

can even uniquely be described by its reduction to the join-irreducible elements of $\mathcal{L}_1$. We therefore only need to compute all (possible) images of f reduced to the join-irreducible elements of $\mathcal{L}_1$ such that the property above holds. To do so, we have to hold in mind that there can be an order relationship between two irreducible elements. Let, for example, $x_1, x_2 \in \mathcal{L}_1$ be join-irreducible elements such that $x_1 \leq_1 x_2$ holds. Then we have to take care that $f(x_2) \leq_2 f(x_1)$ is true. In the case that x_1 and x_2 are incomparable, any combination of $f(x_1)$ and $f(x_2)$ in respect to $\leq$ is possible. Having computed the images of all join-irreducible elements of $\mathcal{L}_1$, we can compute the image of any element of $x \in \mathcal{L}_1$ by

$$f(x) = \bigwedge\{\ f(y) \mid y \leq_1 x,\ y \text{ join-irreducible }\}. \tag{3.1}$$

Notice the special role of the least element of $\mathcal{L}_1$. We have to guarantee $f(0_1) = f(1_2)$. This follows immediately if we set $M = \emptyset$ in the equation above.

Hence, we are able to provide a relatively efficient standard function for this purpose. But first, we need some auxiliary functions. The explanations above indicate that we need our `Lattice` module to compute the set of join-irreducible elements. Therefore, we introduce the following functions

```
gog_poSet g src trg elements =
                  PoSet { poSet_isElem   = flip elem $ gog_homset g src trg
                         ,poSet_elements = elements
                         ,poSet_lEq      = gog_incl g }
```

```
gog_lat   g src trg elements = let pS = gog_poSet  g src trg elements
                                   bE = gog_bottom g src trg
                                   tE = gog_top    g src trg
                                   m  = gog_meet   g
                                   j  = gog_join   g
                               in Lat   { lat_poSet   = pS
                                         ,lat_sup     = j
                                         ,lat_inf     = m
                                         ,lat_topEl   = gog_top    g src trg
                                         ,lat_botEl   = bE
                                         ,lat_atomS   = atomSetBy  pS bE m []
                                         ,lat_jIrredS = irredSetBy pS bE j []
                                         ,lat_mIrredS = irredSetBy pS tE m [] }
```

which create the according data structures for the morphisms between `src` and `trg`. Furthermore, we need two more functions for the determination of the scalars and crisp relations, respectively.

```
scalars :: (Eq mor) => Ded obj mor -> obj -> [mor]
scalars d a =  filter (\m -> ded_incl d m (ded_idmor d a) &&
                             ded_comp d m (ded_top d a a) ==
                             ded_comp d (ded_top d a a) m)
                      $ ded_homset d a a

crispRel :: (Eq mor) => Gog l obj mor -> obj -> obj -> [mor]
crispRel g a b = filter (\m -> gog_up g m == m) $ gog_homset g a b
```

Obviously, they are direct realizations of the underlying definitions. Having this, we can compute the antimorphisms. Since extensive computations are to expect, we provide a convenient parametrization such that the user is able to support our functions throughout this module. For antimorphisms, we provide two routines that are parametrized analogous to our `Lattice` module.

```
antiMorphBy :: (Eq mor) =>
               Gog l obj mor -> (Ded   obj mor -> obj -> [mor]       ) ->
                                (Gog l obj mor -> obj -> obj -> [mor]) ->
               obj -> obj -> [mor -> mor]
antiMorphBy g sc crisp src trg =
  [\x -> foldl1 (gog_meet g) [f_i | (i,f_i)<-m,gog_incl g i x]
   | m <- anti (sc (gog_ded g) src) (crisp g src trg)]
  where
    anti os1 os2 =
      let botG  = gog_bottom g src src
          irred = irredSetBy (gog_poSet g src src os1)
                             botG (gog_join g) []
      in map ((botG, gog_top g src trg):) $ anti' irred os2 []
    anti' []     os2 _      = [[]]
    anti' (a:os) os2 morph  = [ res:f | f<-anti' os os2 morph,
                                  res<-foldl1 intersect $ getPart f a os2]
    getPart []          a os2 = [[(a,f_a)| f_a <- os2]]
    getPart ((b,f_b):fs) a os2
      | gog_incl g a b = [ (a,f_a)| f_a <- os2, gog_incl g f_b f_a]
                         : getPart fs a os2
      | gog_incl g b a = [ (a,f_a)| f_a <- os2, gog_incl g f_a f_b]
                         : getPart fs a os2
      | otherwise      = getPart fs a os2

antiMorph :: (Eq mor) => Gog l obj mor -> obj -> obj -> [mor -> mor]
antiMorph g = antiMorphBy g scalars crispRel
```

The function `antiMorphBy` is, among others, parametrized by two functions to determine the scalars and crisp relations, respectively. The standard variant then is delivered by `antiMorph`. Obviously, we represent an antimorphism f as a list of tuples $(x, f(x))$. The computation is divided into three steps.

The first one is done by the function `getPart`. It is called by `anti'` and gets an (already partially computed) antimorphism `f`, which shall be extended by all possible combinations of an element `a` and its image `f_a` such that the new morphisms are still antimorphisms. To guarantee this, `getPart` generates a list of possible tuples `(a,f_a)` for every single tuple `(b,f_b)`, which is already in the list representing `f`. The intersection of all these lists delivers all possible extensions for `f`. The extension then is done in `anti'` where a list of the resulting new morphisms is generated.

The second step is to guarantee that $f(0_1) = f(1_2)$ holds. This is made by the function `anti`, which inserts the pair `(gog_bottom g src src, gog_top g src trg)` into every computed list after `anti'` is finished.

As we only have lists of tuples up to now, we finally have to provide suitable functions that can be returned. This is done by `antiMorphBy` which makes use of Formula 3.1. Notice that the computation in `antiMorphBy` definitely succeeds, even for the special case $f(0_1)$.

Now, we are ready to test Property (2e) of Definition 2.6.1. Notice that we (in analogy to the `Lattice` module) prefer to extract this test so that the user has direct access and does not necessarily need to perform the whole consistency check.

```
testAntiMorphBy :: (Eq obj, Eq mor) => Gog l obj mor ->
                   (Gog l obj mor -> obj -> obj -> [mor -> mor]) ->
                   (Ded   obj mor -> obj -> [mor])               ->
                   (Gog l obj mor -> obj -> obj -> [mor])        ->
                   obj -> obj -> TestResult obj mor
testAntiMorphBy g anti sc crisp src trg =
  ffold [let scG   = sc (gog_ded g) src
             inclG = gog_incl g
             supSc = foldl1 (gog_join g) [gog_comp g s (a s) | s<-scG]
         in ffold [test (gog_incl g m supSc == foldl1 (&&)
                        [gog_incl g (gog_down g (gog_rres g s m)) (a s)|s<-scG])
                        [src] [m] "Antimorphisms:"]
         | m <- gog_homset g src trg,
           a <- anti g src trg]

testAntiMorphAllBy :: (Eq obj, Eq mor) => Gog l obj mor ->
                      (Gog l obj mor -> obj -> obj -> [mor -> mor]) ->
                      (Ded   obj mor -> obj -> [mor])               ->
                      (Gog l obj mor -> obj -> obj -> [mor])        ->
                      TestResult obj mor
testAntiMorphAllBy g anti sc crisp =
  ffold [ testAntiMorphBy g anti sc crisp src trg | src <- gog_objects g,
                                                    trg <- gog_objects g]

testAntiMorph    :: (Eq obj, Eq mor) =>
                    Gog l obj mor -> obj -> obj -> TestResult obj mor
testAntiMorph    g = testAntiMorphBy    g antiMorph scalars crispRel

testAntiMorphAll :: (Eq obj, Eq mor) => Gog l obj mor -> TestResult obj mor
testAntiMorphAll g = testAntiMorphAllBy g antiMorph scalars crispRel
```

Obviously, we separate the tests twofold. Thus, the user on the one hand has the chance to provide its own functions for determining the needed antimorphisms, scalars and crisp relations, respectively. On the other hand, he can decide whether he wants to test every pair of objects of the underlying Goguen category or if he only wants to check one specific combination of source and target. The tests themselves are a direct realization of Definition

2.6.1(2e). Again, the standard variants are given by `testAntiMorph` and `testAntiMorphAll`, respectively.

The all-in-one tests for $^{\uparrow}$ and $^{\downarrow}$ are provided analogously.

```
testUpDownBy :: (Eq obj, Eq mor) => Gog l obj mor ->
                (Gog l obj mor -> obj -> obj -> [mor -> mor]) ->
                (Ded   obj mor -> obj -> [mor])               ->
                (Gog l obj mor -> obj -> obj -> [mor])        ->
                obj -> obj -> obj -> TestResult obj mor
testUpDownBy g anti sc crisp src trg trg2 =
  let incl   = gog_incl     g;   up     = gog_up       g
      down   = gog_down     g;   comp   = gog_comp     g
      source = gog_source   g;   target = gog_target   g
      homset = gog_homset   g;   conv   = gog_converse g
      scG    = sc (gog_ded g) src
  in
   testAntiMorphBy g anti sc crisp src trg .
   ffold [test (s == gog_bottom g src src || up s == gog_idmor g src)
               [src] [s] "Up(a) = I:" | s<-scG] .
   ffold [ffold [test (source (up   m) == src && target (up   m) == trg)
                      [src,trg] [m] "Domain/Range Up:" .
                 test (source (down m) == src && target (down m) == trg)
                      [src,trg] [m] "Domain/Range Down:"] .
          ffold [test (incl m (down m2) == incl (up m) m2)
                      [src,trg] [m,m2] "UpDown-Galois:"
                | m2 <- homset src trg]
          |   m <- homset src trg] .
          ffold [test (up (comp (conv m) (down m2)) ==
                       comp (conv $ up m) (down m2))
                      [src,trg,trg2] [m,m2] "Comp/Conv/UpDown:"
                | m  <- homset trg src,
                  m2 <- homset trg trg2]

testUpDownAllBy :: (Eq obj, Eq mor) => Gog l obj mor ->
                   (Gog l obj mor -> obj -> obj -> [mor -> mor]) ->
                   (Ded   obj mor -> obj -> [mor])               ->
                   (Gog l obj mor -> obj -> obj -> [mor])        ->
                   TestResult obj mor
testUpDownAllBy g anti sc crisp =
    ffold [ testUpDown g src trg trg2 | src  <- gog_objects g,
                                        trg  <- gog_objects g,
                                        trg2 <- gog_objects g]

testUpDown :: (Eq obj, Eq mor) =>
              Gog l obj mor -> obj -> obj -> obj -> TestResult obj mor
testUpDown g = testUpDownBy g antiMorph scalars crispRel

testUpDownAll :: (Eq obj, Eq mor) => Gog l obj mor -> TestResult obj mor
testUpDownAll g = testUpDownAllBy g antiMorph scalars crispRel
```

Hence, we can test Goguen categories as follows.

```
gog_TESTBy :: (Eq obj, Eq mor) => Gog l obj mor ->
              (Gog l obj mor -> obj -> obj -> [mor -> mor]) ->
              (Ded   obj mor -> obj -> [mor])               ->
              (Gog l obj mor -> obj -> obj -> [mor])        ->
              TestResult obj mor
gog_TESTBy g anti sc crisp =
   let top = gog_top g
       bot = gog_bottom g
   in  ffold [test (not (top src trg == bot src trg))
                   [src,trg] [top src trg,bot src trg] "Top==Bot:"
              | src <- gog_objects g, trg <- gog_objects g] .
       testUpDownAllBy g anti sc crisp .
       ded_top_incl_TEST (gog_ded g)   .
       divAll_lres_TEST (gog_divAll g) .
       distrAll_TEST (gog_distrAll g)

gog_TEST :: (Eq obj, Eq mor) => Gog l obj mor -> TestResult obj mor
gog_TEST g = gog_TESTBy g antiMorph scalars crispRel
```

Linear Goguen categories are essential to get the equivalence of the three notions of crispness, s-crispness and l-crispness (cf. Theorem 2.6.3). Thus, we provide predefined tests to check linearity within a Goguen category.

```
testLinearBy :: Eq mor => Gog l obj mor -> (Ded obj mor -> obj -> [mor]) ->
                          TestResult obj mor
testLinearBy g sc =
    ffold [ let scG = sc (gog_ded g) o
                bG  = gog_bottom g o o
```

```
            in ffold [test (s1 == bG || gog_meet g s1 s2 /= bG || s2 == bG)
                           [o] [s1,s2] "Linearity:"
                       |s1 <- scG ,s2 <- scG]
            | o <- gog_objects g]

testLinear :: (Eq mor) => Gog l obj mor -> TestResult obj mor
testLinear g = testLinearBy g scalars
```

Obviously, the test is done by checking all scalars of all objects for the linearity property.

With all kinds of relational categories, functors (cf. Definition 2.5.2) are essential to express interactions between them. Hence, we provide suitable tests to check whether a given functor is a functor between Goguen categories. RATH already provides the data structure `Fun` shown in Section 2.5.1. Unfortunately, suitable tests are only included up to the level of allegories so that we cannot rely on a preimplemented test for functors between Dedekind categories. Hence, we have to provide the missing test ourself.

```
gog_fun_TEST :: Eq mor2 => Gog l obj1 mor1 -> Gog l obj2 mor2 ->
                           Fun obj2 mor2 obj1 mor1 -> TestResult obj1 mor1
gog_fun_TEST g1 g2 f@(Fun fObj fMor) =
  let os = gog_objects g1
  in ffold [ test (fMor (gog_join g1 m1 m2)==gog_join g2 (fMor m1) (fMor m2))
                  [o1,o2] [m1,m2] "Join:" .
             test (fMor (gog_lres g1 m1 m2)==gog_lres g2 (fMor m1) (fMor m2))
                  [o1,o2] [m1,m2] "LRes:"
             | o1<-os, o2<-os, m1<-gog_homset g1 o1 o2, m2<-gog_homset g1 o1 o2].
     ffold [ test (fMor (gog_up g1 m)==gog_up g2 (fMor m))
                  [o1,o2] [m] "Up:" .
             test (fMor (gog_down g1 m)==gog_down g2 (fMor m))
                  [o1,o2] [m] "Down:"
             | o1<-os, o2<-os, m<-gog_homset g1 o1 o2] .
     ffold [ test (fMor (gog_bottom g1 o1 o2)==gog_bottom g2 (fObj o1) (fObj o2))
                  [o1,o2] [gog_bottom g1 o1 o2] "Bottom:" .
             test (fMor (gog_top g1 o1 o2)==gog_top g2 (fObj o1) (fObj o2))
                  [o1,o2] [gog_top g1 o1 o2] "Top:"
             | o1<-os, o2<-os] .
     allrepr_TEST (gog_all g1) (gog_all g2) f
```

We check whether a given functor `f` between two given Goguen categories `g1` and `g2` respects join and the left residual as well as the up/down operation and the top/bottom elements.

The composition, conversion and meet operators are already tested by `allrepf_TEST`.

Finally, we again have to connect the class and module view, respectively, by providing suitable instances. A listing can be found in Appendix B.

3.4 A module for $\mathcal{L}$-fuzzy relations

In this chapter we aim at a module to support the standard instantiation of Goguen categories using $\mathcal{L}$-fuzzy relations. Thus, we first have to make ourselves clear what this module shall be able to do and, just as important, what it does not have to.
Of course, we need a data structure representing an $\mathcal{L}$-fuzzy relation. Furthermore, the user shall be able to manipulate and construct a relation. To guarantee extendability, a very important design rule with this is to hide the implementation. Thus, we have to provide certain access routines to create an $\mathcal{L}$-fuzzy relation starting from the top/bottom relation between certain source and target. Furthermore, the standard matrix operations shall be available. Within fuzzy control, new intersection and composition operators derived from lattice-ordered operator sets (loos') (cf. Definition 2.4.2) play an important role. Hence, we have to provide an appropriate data structure as well as special functions for this purpose.
The export list

```
module LFuzzyRel (FRel(..),Loos(..),fZip,fIncl,fComp,fJoin,fMeet,fConv,fCompLoos,
                  fOpLoos,fUp,fDown,fTop,fBot,fLRes,fRRes,fSyQ,fLResLoos,
                  fRResLoos,fSyQLoos,fDerOp,fScalar,fId,fUpd,fUpdUnBy,
                  fUpdAllUnBy,fUpdBinBy,fEntryAt,fExtSrc,fExtTrg,getFRelsBy,
                  getFRels,getAllFRelsBy,getAllFRels,lat_infLoos,lat_supLoos,
                  ) where
```

shows the provided data structures `FRel` and `Loos`, the basic operations (join, meet, conversion etc.), as well as their counterparts for derived operations (`fOpLoos` and `fCompLoos` etc.), the top and bottom relation, some functions for manipulating relations componentwise (`fUpd`, `fUpdUnBy`, etc.) and different routines to compute all $\mathcal{L}$-fuzzy relations fulfilling a certain predicate between given source and target (`getFRelsBy`, `getAllFRelsBy`, etc.). The last functions become interesting when we instantiate relational categories later on.

This basic functionality suffices for our purposes and leaves space for future extensions. These extensions could, for example, go into deep with $\mathcal{L}$-fuzzy relations and provide an implementation of special operations like fuzzy implication or fuzzy negation.

Now, we are ready to switch to the implementation. Since the entries of our $\mathcal{L}$-fuzzy relations have to constitute a complete Brouwerian lattice, we obviously have to import the `Lattice` module. Furthermore, some special list operations are needed later on.

```
import Lattice
import List (transpose,nub)
```

The next step is the `FRel` data structure. Several design decisions have to be made here.

```
data FRel e obj1 obj2 =  FRel { lat   :: Lat e
                               ,src   :: [obj1]
                               ,trg   :: [obj2]
                               ,rel   :: [[e]] }
```

An $\mathcal{L}$-fuzzy relation consists of its underlying entry lattice `lat`, its source and target (`src`, `trg`) and the real relation `rel` represented as a matrix.
Notice that we deliberately do not use the `Set` module to describe source and target, respectively. The reason is that the interpretability of our relations would suffer because we would not know how the rows and columns are labeled. Thus, `src` resp. `trg` can be seen as the labeling of the given relation.

As mentioned above, we use the matrix representation for our $\mathcal{L}$-fuzzy relations and implement it by a list of lists of entries. Another possibility would have been to use the preimplemented `Array` module, which has the advantage that we can rely on predefined functions and have a comprehensive error detection. But, there are several drawbacks of this approach. The first one is that the range of an array has to be indexable, i.e., any type we would use for `src` and `trg` had to implement the `Ix` type class. For `Double` and `Rational` values this is at least difficult. The second (even worse) drawback is that we could only use coherent source and target ranges, respectively. Thus, the source $\{1, 2, 3, 5\}$ would not be allowed. At least with the automated construction of relations this causes problems. Consider, for example, two relations $R : A \to B$ and $S : A \to C$ with $B := \{1, 2, 3\}$ and $C := \{7, 8, 9, 10\}$. Obviously, we have to represent them by

```
relR = { lat = ...     relS = { lat = ...
        ,src = ...             ,src = ...
        ,trg = (1,3)           ,trg = (7,10)
        ,rel = ... }           ,rel = ... }
```

if we use the array implementation. But, if we want to construct the direct sum, we get problems computing the `trg` parameter of the resulting injections. Obviously, `(1,10)` is not allowed. Since the construction of trg would only be possible with considerable efforts and the interpretability of the resulting relations would suffer anyway, we use the list representation. Now, one could still use the Array module to represent the matrix by mapping source and target to suitable array ranges (e.g., each element could be mapped to its list index). But, tests showed that the Array module is slower than the list operations. All this drove us to define FRel as shown above. Of course, several consistency tests for matrices have to be provided with our approach.

Later on, we need Show and Eq instances of FRel.

```
instance (Show e,Show obj1, Show obj2) => Show (FRel e obj1 obj2) where
  show r = "Source: "++ (show $ src r) ++ "\n" ++
           "Target: "++ (show $ trg r) ++ "\n" ++
           "Rel    : "++ "\n" ++ (show $ rel r)++"\n"

instance (Eq e, Eq obj1, Eq obj2) => Eq (FRel e obj1 obj2) where
  (==) rel1 rel2 = src rel1 == src rel2 &&
                   trg rel1 == trg rel2 &&
                   rel rel1 == rel rel2
```

The Eq instance again makes clear that two relations are only equal if they have equally labeled rows and columns as well as equal matrix entries.

The data structure for loos' is implemented as follows.

```
data Loos el = Loos { loos_lat :: Lat el
                     ,loos_op  :: el -> el -> el
                     ,loos_e   :: el
                     ,loos_z   :: el }
```

It is a direct realization of the underlying definition whereas e is the neutral and z the zero element. This data structure induces some standard instantiations since conventional join and meet, respectively, constitute a loos.

```
lat_infLoos,lat_supLoos :: Lat el -> Loos el
lat_infLoos l = Loos { loos_lat = l
                      ,loos_op  = lat_inf l
                      ,loos_e   = lat_topEl l
                      ,loos_z   = lat_botEl l }

lat_supLoos l = Loos { loos_lat = l
```

```
                        ,loos_op  = lat_sup l
                        ,loos_e   = lat_botEl l
                        ,loos_z   = lat_topEl l }
```

Obviously, the neutral and zero elements, respectively, are dual with `lat_infLoos` and `lat_supLoos`.

In the following some auxiliary functions are defined. Notice, that they are not exported. Later on, we want to provide different functions to access certain entries of $\mathcal{L}$-fuzzy relations and to set/update/read them. Thus, we provide the functions

```
updateAtBy :: (a -> a) -> Int -> Int -> [[a]] -> [[a]]
updateAtBy f i j mat = let
                         matSplitted = splitAt i mat
                         rowSplitted = splitAt j (head $ snd matSplitted)
                       in (fst matSplitted)++[(fst rowSplitted)++
                          (f (head $ snd rowSplitted) : tail(snd rowSplitted))]++
                          (tail $ snd matSplitted)

insertAt :: Int -> Int -> a -> [[a]] -> [[a]]
insertAt i j el = updateAtBy (const el) i j
```

for this purpose. They are different in the manner that `insertAt` only inserts an element `el` in a matrix `mat` at position (`i`,`j`), and `updateAtBy` updates the element in question using the unary function `f`.

For error handling we provide the following error message to inform the user properly.

```
srcTrgError :: String
srcTrgError = "Underlying sources/targets not suitable !"
```

Hence, we are ready to implement the standard operations for $\mathcal{L}$-fuzzy relations. To avoid name space conflicts, all standard functions for $\mathcal{L}$-fuzzy relations will be started by `f`.
The inclusion operation is delivered by the following function.

```
fIncl :: (Eq obj1, Eq obj2) => FRel e obj1 obj2 -> FRel e obj1 obj2 -> Bool
fIncl rel1 rel2 = let l    = lat rel1
                  in if (src rel1)==(src rel2) && (trg rel1) == (trg rel2) then
                       fIncl' l (rel rel1) (rel rel2)
                     else error srcTrgError
                  where
```

```
fIncl' l []       []             = True
fIncl' l ([]:r1s) ([]:r2s)       = fIncl' l r1s r2s
fIncl' l ((x:xs):r1s) ((y:ys):r2s) = (lat_lEq l x y) &&
                                    fIncl' l (xs:r1s) (ys:r2s)
```

The inclusion check is done componentwise and immediately finished when two elements are found that are not in the corresponding order relation. Notice that our error detection is restricted to determining whether the matrices are compatible with respect to their sources and targets. We do not test whether the matrix given in `rel` really corresponds to the given source and target, respectively. This has to be done manually by the user after constructing the $\mathcal{L}$-fuzzy relation. To do so, the test functions introduced later on can be used. With this restriction we are able to guarantee consistency and acceptable run times.

The next step is to provide join, meet and composition operations. We start with operations derived from loos'. Since standard join and meet constitute a loos we then can rely on these functions.

```
fZip :: (Eq obj1, Eq obj2) =>
        (e -> e -> e) -> FRel e obj1 obj2 -> FRel e obj1 obj2 -> FRel e obj1 obj2
fZip f rel1 rel2 = let l = lat rel1
                   in
                     if (src rel1)==(src rel2) && (trg rel1) == (trg rel2) then
                       FRel l (src rel1) (trg rel2) $
                            zipWith (zipWith f) (rel rel1) (rel rel2)
                     else error srcTrgError

fOpLoos :: (Eq obj1, Eq obj2) =>
           Loos e -> FRel e obj1 obj2 -> FRel e obj1 obj2 -> FRel e obj1 obj2
fOpLoos loos = fZip (loos_op loos)

fCompLoos :: (Eq obj2) =>
           Loos e -> FRel e obj1 obj2 -> FRel e obj2 obj3 -> FRel e obj1 obj3
fCompLoos loos rel1 rel2 =
  let l = lat rel1
  in if (trg rel1)==(src rel2)then
       FRel l (src rel1) (trg rel2)
              [ [foldr1 (lat_sup l) $ zipWith (loos_op loos) x [y!!i|y<-rel rel2]
                 | i<-[0..length (head $ rel rel2)-1] ]
                | x<-rel rel1]
     else error srcTrgError
```

The key functionality for derived join/meet operations is established in `fZip` which takes two $\mathcal{L}$-fuzzy relations and then concatenates them using a given binary function `f`. Hence, `fOpLoos` can use `fZip` parametrized by the loos operator `loos_op` of `loos`. Finally, the derived composition operator is delivered by `fCompLoos`.

Using `lat_infLoos` and `lat_supLoos` from above, we are now able to provide the standard operations.

```
fJoin,fMeet :: (Eq obj1, Eq obj2) =>
               FRel e obj1 obj2 -> FRel e obj1 obj2 -> FRel e obj1 obj2
fJoin rel1 = fOpLoos (lat_supLoos $ lat rel1) rel1

fMeet rel1 = fOpLoos (lat_infLoos $ lat rel1) rel1

fComp :: (Eq obj2) => FRel e obj1 obj2 -> FRel e obj2 obj3 -> FRel e obj1 obj3
fComp rel1 = fCompLoos (lat_infLoos $ lat rel1) rel1
```

Conversion and the up/down operators are implemented as follows.

```
fConv :: FRel e obj1 obj2 -> FRel e obj2 obj1
fConv rel1 = FRel (lat rel1) (trg rel1) (src rel1) (transpose $ rel rel1)

fUp,fDown :: FRel e obj1 obj2 -> FRel e obj1 obj2
fUp rel1 = let l     = lat rel1
           in FRel l (src rel1) (trg rel1) $
                     map (map (\x -> if   lat_lEq l x $ lat_botEl l then x
                                     else lat_topEl l ) ) $ rel rel1

fDown rel1 = let l     = lat rel1
             in FRel l (src rel1) (trg rel1) $
                       map (map (\x -> if   lat_lEq l (lat_topEl l) x then x
                                       else lat_botEl l ) ) $ rel rel1
```

Now, we want to introduce some standard relations which have to be used to construct any other relation componentwise.

```
fScalar :: Lat e -> e -> [obj1] -> FRel e obj1 obj1
fScalar e el s = let lS   = length s
                     botEl = lat_botEl e
                 in FRel e s s [replicate i botEl ++
                                         (el : replicate (length s-i-1) botEl)
```

```
                                | i<-[0..length s-1]]

fId :: Lat e -> [obj1] -> FRel e obj1 obj1
fId e = fScalar e (lat_topEl e)

fTop,fBot :: Lat e -> [obj1] -> [obj2] -> FRel e obj1 obj2
fTop e s t = FRel e s t $ replicate (length s)
                                    (replicate (length t) $ lat_topEl e)

fBot e s t = FRel e s t $ replicate (length s)
                                    (replicate (length t) $ lat_botEl e)
```

From the function names it is obvious which kind of relation they implement. All routines have to get the underlying lattice as well as the source and target object. Since source and target are identical with the scalars and the identity relation, we need only one parameter. Notice that the set of all scalars for a given source plays an important role within Goguen categories since it is isomorphic to the elements of the entry lattice.

As indicated by the α-cut Theorem (cf. Theorem 2.6.2), the left and right residual as well as the symmetric quotient of two given $\mathcal{L}$-fuzzy relations are important operations within Goguen categories. Before we can implement them, we have to undergo some efficiency considerations. The following lemma characterizes the entries of the left residual. Notice that the underlying definition $Q;R \sqsubseteq S \Leftrightarrow Q \sqsubseteq (S/R)$ indicates that new residued operations can be derived from lattice-ordered semigroups. Hence, we consider the general case.

Lemma 3.4.1. *Let $R : B \to C$ and $S : A \to C$ be two $\mathcal{L}$-fuzzy relations over $\mathcal{L}$. Furthermore, let $*$ be a loos based derived operation on $\mathcal{L}$. Then the left residual $S/_*R$ can be computed componentwise by $(S/_*R)(x,y) = \bigvee\{a \in \mathcal{L} \mid \forall z : a * R(y,z) \leq S(x,z)\}$.*

Proof. We show

$$\forall x,z : (Q;_* R)(x,z) \leq S(x,z) \Leftrightarrow \forall x,y : Q(x,y) \leq \bigvee\{a \in \mathcal{L} \mid \forall z : a * R(y,z) \leq S(x,z)\} \quad .$$

The assertion immediately follows from

$$\begin{aligned}
& \forall x,y : Q(x,y) \leq \bigvee\{a \in \mathcal{L} \mid \forall z : a * R(y,z) \leq S(x,z)\} \\
\Leftrightarrow\ & \forall x,y,z : Q(x,y) * R(y,z) \leq S(x,z) \\
\Leftrightarrow\ & \forall x,z : \bigvee_y (Q(x,y) * R(y,z)) \leq S(x,z) \\
\Leftrightarrow\ & \forall x,z : (Q;_* R)(x,z) \leq S(x,z).
\end{aligned}$$

□

Since $Q \backslash_* S = (S^\smile /_* Q^\smile)^\smile$ we automatically have that $(Q \backslash_* S)(y,z)$ can be computed by $\bigvee\{a \in \mathcal{L} \mid \forall x : a * Q(x,y) \leq S(x,z)\}$.

With the implementation, we again first introduce functions for derived operations and then give the standard variants as a special case.

```
fLResLoos :: (Eq obj1, Eq obj2, Eq obj3) => Loos e ->
             FRel e obj1 obj3 -> FRel e obj2 obj3 -> FRel e obj1 obj2
fLResLoos loos t s =
   let targ = src s
       sour = src t
       l    = lat t
       sup  = lat_sup l
       lEq  = lat_lEq l
       f    = loos_op loos
   in FRel { lat = l
            ,src = sour
            ,trg = targ
            ,rel = part (length targ)
                        [let sl  = (rel s !! y)
                             tl  = (rel t !! x)
                             els = [ el | el<-lat_elements l, and $
                                          zipWith (\a b -> lEq (f el a) b) sl tl]
                         in if length els>1 then foldl1 sup els
                            else head els
                          | x<-[0..length sour-1], y<-[0..length targ-1]] }
   where
     part i [] = []
     part i xs = let (s1,s2) = splitAt i xs
                 in s1 : part i s2

fRResLoos,fSyQLoos :: (Eq obj1, Eq obj2, Eq obj3) =>  Loos e ->
             FRel e obj1 obj2 -> FRel e obj1 obj3 -> FRel e obj2 obj3
fRResLoos loos s t = fConv $ fLResLoos loos (fConv t) (fConv s)

fSyQLoos loos s t = fMeet (fRResLoos loos s t) $ fLResLoos loos (fConv s) (fConv t)

fLRes :: (Eq obj1, Eq obj2, Eq obj3) =>
```

```
         FRel e obj1 obj3 -> FRel e obj2 obj3 -> FRel e obj1 obj2
fLRes t = fLResLoos (lat_infLoos $ lat t) t

fRRes,fSyQ :: (Eq obj1, Eq obj2, Eq obj3) =>
         FRel e obj1 obj2 -> FRel e obj1 obj3 -> FRel e obj2 obj3
fRRes s = fRResLoos (lat_infLoos $ lat s) s

fSyQ s t = fMeet (fRRes s t) $ fLRes (fConv s) (fConv t)
```

Obviously, the realization of `fLResLoos` directly corresponds to Lemma 3.4.1. The other functions can rely on it so that no separate implementations are necessary. Notice that with `fSyQLoos` and `fSyq` we definitely have to use the `fMeet` function to concatenate the left resp. right residual. This comes due to the fact that the *and* operator of the underlying definition $X \sqsubseteq syQ(Q,S)$, iff $X \sqsubseteq (Q\backslash S)$ *and* $X \sqsubseteq (Q^{\smile}/S^{\smile})$ has to be represented by `fMeet`.
As we saw in the `Goguen` module, we need a standard routine that generates certain derived operations within Goguen categories. This is done by the following function.

```
fDerOp :: (Eq obj) => String -> Loos e ->
                      FRel e obj obj -> FRel e obj obj -> FRel e obj obj
fDerOp l =
  case l of
     "Meet" -> fOpLoos
     "Comp" -> fCompLoos
     "LRes" -> fLResLoos
     "RRes" -> fRResLoos
     "syQ"  -> fSyQLoos
```

The label `l` indicates which kind of derived operation shall be generated. According to the definition of `gog_derOp`, the user only has to provide the loos to have a derived operation within Goguen categories.

Up to now, we are only able to handle given $\mathcal{L}$-fuzzy relations. But, we need some suitable functions to construct them in a componentwise manner. The most simple operation is to read an element at a given position in the relation.

```
fEntryAt :: FRel e obj1 obj2 -> (Int,Int) -> e
fEntryAt rel1 (i,j) = ((rel rel1) !! i) !! j
```

The user has to give the row `i` and the column `j` of the entry he wants to read. Obviously, `i` and `j` have to describe the position in the matrix of the relation `rel1` so that we can rely on the `!!` operator. Notice that the row and column count, respectively, start with 0.

The following functions can be used to update given $\mathcal{L}$-fuzzy relations. This can be done threefold. First, the user can give a list of tuples, each tuple containing an index pair and the element that shall be set. This is covered by `fUpd` which relies on the above defined function `insertAt`.

```
fUpd :: FRel e obj1 obj2 -> [((Int,Int),e)] -> FRel e obj1 obj2
fUpd rel1 els = FRel (lat rel1) (src rel1) (trg rel1) $ insertAll els (rel rel1)
                where
                  insertAll []              m = m
                  insertAll (((i,j),x):xs) m = insertAt i j x $ insertAll xs m
```

The second variant is to update an element in the matrix by a unary function `f`. Here we can differentiate two cases. If one explicitly wants to give the indices that shall be updated, one can use `fUpdUnBy`. If the whole matrix shall be updated, the function `fUpdAllUnBy`, which makes use of the above defined function `updateAtBy`, is to use.

```
fUpdUnBy :: (e -> e) -> FRel e obj1 obj2 -> [(Int,Int)] -> FRel e obj1 obj2
fUpdUnBy f rel1 ixs = FRel (lat rel1) (src rel1) (trg rel1) $
                           updateAllBy f ixs (rel rel1)
                      where
                        updateAllBy _ []          m = m
                        updateAllBy f ((i,j):ixs) m = updateAtBy f i j $
                                  updateAllBy f ixs m


fUpdAllUnBy :: (e -> e) -> FRel e obj1 obj2 -> FRel e obj1 obj2
fUpdAllUnBy f rel1 = FRel (lat rel1) (src rel1) (trg rel1)
                          [[f y | y<-x] | x<-rel rel1]
```

Notice that these three functions would suffice to construct and manipulate $\mathcal{L}$-fuzzy relations quite comfortably. But, one may want to update certain entries by a binary function applied to the old entry and another (given) element. A typical scenario is the "partial join" operation of two matrices where only some selected fields shall be joined. We support this kind of operation by the function

```
fUpdBinBy :: (e -> e -> e) -> FRel e obj1 obj2 -> [((Int,Int),e)] ->
             FRel e obj1 obj2
fUpdBinBy f rel1 els = FRel (lat rel1) (src rel1) (trg rel1) $
                            updateAllBy f els (rel rel1)
                       where
                          updateAllBy _ []             m = m
                          updateAllBy f (((i,j),e):xs) m = updateAtBy (f e) i j $
```

```
                                                    updateAllBy f xs m
```

which again makes use of `updateAtBy`. It takes a binary function together with a list of tuples, each tuple containing an index pair and an element, and updates the entries in question. Obviously, `fUpdBinBy` could be easily reduced to `fUpdUnBy`. But, for efficiency considerations, we again implement `updateAllBy` with the only difference that `f` here is a binary (instead of unary) function.

Thus, we have the basic functionality. Later on, we want to instantiate different kinds of relational categories using this module. From the underlying definitions we know that the relations (morphisms) have to form certain kinds of structures. Hence, we can support these instantiations by providing functions that automatically compute certain $\mathcal{L}$-fuzzy relations between given source and target. The function

```
getFRelsBy :: (FRel e obj1 obj2 -> Bool) -> Lat e -> [obj1] -> [obj2] ->
              [FRel e obj1 obj2]
getFRelsBy pr e s t =
    let lS = length s
        lT = length t-1
    in filter pr [FRel e s t x | x <- map (split $ lT+1)
                                      (allocate (length s*length t-1) e)]
    where
      allocate 0 e = map (:[]) $ lat_elements e
      allocate n e = [ o:m | o<-lat_elements e, m<-allocate (n-1) e]
      split _ [] = []
      split i xs = let s = splitAt i xs
                   in fst s : split i (snd s)
```

takes a unary predicate `pr` on the set of all $\mathcal{L}$-fuzzy relations between source `s` and target `t` and an entry lattice `e`. It then returns a list of all $\mathcal{L}$-fuzzy relations between `s` and `t` fulfilling `pr`. This is done with the help of two functions. First, `allocate` constructs all possibilities to combine `length s * length t` elements of the entry lattice. It returns a list for every combination. Obviously, the length of such a list exactly corresponds to the number of entries in a relation between `s` and `t`. To achieve our matrix representation, we then use the `split` function to split the lists. The result then can be used to construct the final $\mathcal{L}$-fuzzy relations. At the end we filter out the relations fulfilling `pr`.
The standard variant of `getFRelsBy` returns all $\mathcal{L}$-fuzzy relations.

```
getFRels :: Lat e -> [obj1] -> [obj2] -> [FRel e obj1 obj2]
getFRels = getFRelsBy $ const True
```

Good examples for a special predicates are “is crisp” or “is total”.

The function

```
getAllFRelsBy :: Eq obj =>
                (FRel e obj obj -> Bool) -> Lat e -> [[obj]] -> [[FRel e obj obj]]
getAllFRelsBy pr entry objs = let os = nub objs
                              in [getFRelsBy pr entry src trg | src<-os,trg<-os]
```

goes a step further. It takes a list of sources and targets (`objs`) and computes all $\mathcal{L}$-fuzzy relations fulfilling the predicate `pr` between any two elements out of `objs`. With this it relies on `getFRelsBy`. Later on, `getAllFRelsBy` can be used in quite a comfortable way to instantiate certain relational categories. Furthermore, we see a little demerit of Haskell's strong typing. We can only use one type of sources/targets because we put them all together into one list. The same is true for the resulting $\mathcal{L}$-fuzzy relations.
Again, we provide a standard version that computes all relations.

```
getAllFRels :: Eq obj => Lat e -> [[obj]] -> [[FRel e obj obj]]
getAllFRels = getAllFRelsBy $ const True
```

Before we come to the end, we want to provide two auxiliary functions which will be used within our module for fuzzy controllers, later on.

```
fExtSrc,fExtTrg :: [[obj]] -> FRel e obj obj -> [[obj]] ->
                                    FRel e obj obj
fExtSrc usrcs r lsrcs = FRel (lat r) (concat usrcs++src r++concat lsrcs) (trg r)
                             (rel (fBot (lat r) (concat usrcs) $ trg r)++rel r++
                              rel (fBot (lat r) (concat lsrcs) $ trg r))
fExtTrg ltrgs r rtrgs =
   FRel (lat r) (src r) (concat ltrgs++trg r++concat rtrgs)
        (zipWith (++)         (rel $ fBot (lat r) (src r) $ concat ltrgs) $
         zipWith (++) (rel r) (rel $ fBot (lat r) (src r) $ concat rtrgs))
```

Their manner of function shall be explained with an example. Suppose, $R : A_1 \to B$ is a relation and A_2 is a set. If we want to extend R to be of type $A_1 + A_2 \to B$, we have to use the induced crisp injection $\iota : A_1 \to A_1 + A_2$ and compute $\iota_1^{\smile}; R$. But, this computation does nothing more than to add $|A_2|$ zero rows to R. Hence, we can avoid to use the composition operator and, thus, can compute the term rather efficiently. This is exactly what is done by `fExtSrc`. The parameter `usrcs` is a list of objects and causes `fExtSrc` to extend the given relation `r` to the upper side. Analogously, `lsrcs` is used to extend `r` to the lower side. The function `fExtTrg` does exactly the same, but extends the target of `r`. This corresponds to adding zero columns to the left resp. right side of `r`.

Last but not least, we provide a consistency check.

```
check :: FRel e obj1 obj2 -> Bool
check r = length (rel r) == length (src r) &&
          and (map ((== length (trg r)).length) $ rel r)
```

We test whether the given matrix is consistent with the given source and target of the $\mathcal{L}$-fuzzy relation, respectively.

At this point the implemented functions are sufficient for our purposes. We are now ready to provide suitable lattice instantiations as well as instantiations of the relational categories.

3.5 Lattices of $\mathcal{L}$-fuzzy relations

In this section we deliver predefined functions to instantiate certain kinds of lattices of $\mathcal{L}$-fuzzy relations.

```
module LFuzzyRelLattices (fRelPoSetBy   ,fRelPoSet    ,fRelPoSetList    ,
                          fRelUSemiLatBy,fRelUSemiLat,fRelUSemiLatList,
                          fRelLSemiLatBy,fRelLSemiLat,fRelLSemiLatList,
                          fRelLatBy     ,fRelLat      ,fRelLatList       ) where

import LFuzzyRel
import Lattice
```

The export list shows the structures that can be generated. To do so, we have to import the modules shown above.
We start with the weakest structure `PoSet`.

```
fRelPoSetBy :: (Eq obj1, Eq obj2) =>
               (FRel e obj1 obj2 -> Bool) -> Lat e ->
   [obj1] -> [obj2]  -> PoSet (FRel e obj1 obj2)
fRelPoSetBy pr e s t = PoSet { poSet_isElem  = \x -> src x==s && trg x==t && pr x
                              ,poSet_elements = getFRelsBy pr e s t
                              ,poSet_lEq      = fIncl }

fRelPoSet :: (Eq obj1, Eq obj2) =>
```

```
             Lat e -> [obj1] -> [obj2] -> PoSet (FRel e obj1 obj2)
fRelPoSet = fRelPoSetBy $ const True

fRelPoSetList :: (Eq e, Eq obj1, Eq obj2) =>
                 Lat e -> [FRel e obj1 obj2] -> PoSet (FRel e obj1 obj2)
fRelPoSetList e rs = PoSet { poSet_isElem   = flip elem $ rs
                            ,poSet_elements = rs
                            ,poSet_lEq      = fIncl }
```

As shown here, posets can be generated threefold. To get the poset of all relations between source `s` and target `t` fulfilling a unary predicate `pr`, one can use the function `fRelPoSetBy`. It relies on `getFRelsBy` to generate the requested $\mathcal{L}$-fuzzy relations. The test whether a certain relation belongs to this poset is done by checking source and target of the relation in question as well as the predicate property. As another possibility, one could check whether the relation is in the element list `poSet_elements`. But, we think testing a single predicate is faster in the most general cases (e.g., for the crispness property). Furthermore, we avoid a more detailed parametrization since a user can easily instantiate "special" posets himself. The second variant then is the standard version. It delivers the poset of all $\mathcal{L}$-fuzzy relations between given source and target.
The last function `fRelPoSetList` takes a list of the relations that shall belong to the created poset. It then delivers the requested structure. Notice that we do not perform any consistency checks with the given relation list.

Having this, we can proceed with lattice structures. The construction of upper semilattices is shown with the next three functions.

```
fRelUSemiLatBy :: (Eq obj1, Eq obj2) =>
                  (FRel e obj1 obj2 -> Bool) -> Lat e ->
  [obj1] -> [obj2]  -> USemiLat (FRel e obj1 obj2)
fRelUSemiLatBy pr e s t = USemiLat { uSemiLat_poSet = fRelPoSetBy pr e s t
                                    ,uSemiLat_sup   = fJoin
                                    ,uSemiLat_topEl = fTop e s t }

fRelUSemiLat :: (Eq obj1, Eq obj2) =>
                Lat e -> [obj1] -> [obj2] -> USemiLat (FRel e obj1 obj2)
fRelUSemiLat = fRelUSemiLatBy $ const True

fRelUSemiLatList :: (Eq e, Eq obj1, Eq obj2) =>
                    Lat e -> [FRel e obj1 obj2] -> USemiLat (FRel e obj1 obj2)
```

```
fRelUSemiLatList e rs = let
                            s = if null rs then [] else src $ head rs
                            t = if null rs then [] else trg $ head rs
                          in
                            USemiLat { uSemiLat_poSet = fRelPoSetList e rs
                                      ,uSemiLat_sup   = fJoin
                                      ,uSemiLat_topEl = fTop e s t }
```

Notice that with `fRelUSemiLatList` we have to catch the case that the relation list is empty. In this situation we take the empty list as source and target, respectively, to construct the top element `uSemiLat_topEl`.

From the definitions of the underlying data types we know that we need to compute the atoms for lower semi lattices and lattices as well as the join resp. meet-irreducible elements for lattices. To do so, we first have to make ourselves clear which $\mathcal{L}$-fuzzy relations over a certain lattice are irreducible and atoms, respectively. Obviously, an $\mathcal{L}$-fuzzy relation is an atom or irreducible, respectively, if and only if all occurrent entries are.
We support this by providing the auxiliary function `specialEls`.

```
specialEls :: Lat e -> FRel e obj1 obj2 -> [e] -> [FRel e obj1 obj2]
specialEls e bt xs = [fUpd bt [((i,j),x)]
     | i<-[0..length (src bt)-1],j<-[0..length (trg bt)-1],x<-xs]
```

First it takes the entry lattice `e`. For efficiency reasons the parameter `bt` is useful. If we consider the Boolean lattice with atoms a and b with least/greatest element $0/1$, the lattice of all $\mathcal{L}$-fuzzy relations between source $\{1,2\}$ and target $\{3,4\}$ has the following atoms and join-irreducible elements

$$\begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & a \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ a & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & a \end{pmatrix}, \begin{pmatrix} b & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & b \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ b & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & b \end{pmatrix}.$$

The meet-irreducible elements are given by

$$\begin{pmatrix} a & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & a \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ a & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & a \end{pmatrix}, \begin{pmatrix} b & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & b \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ b & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & b \end{pmatrix}.$$

Hence, for the atoms/join-irreducible elements it is useful to successively update the bottom relation between given source and target. For the meet-irreducible elements it is better to use the full relation. Which one to use is told by the parameter `bt`. Last but not least, the atoms/join- resp. meet-irreducible elements of the entry lattice `e` have to be given in the parameter `xs` so that the computation can be done.

The next functions generate lower semilattices in analogy to the poset functions above.

```
fRelLSemiLatBy :: (Eq obj1, Eq obj2) =>
                  (FRel e obj1 obj2 -> Bool) -> Lat e ->
                  [obj1] -> [obj2] -> LSemiLat (FRel e obj1 obj2)
fRelLSemiLatBy pr e s t =
       LSemiLat { lSemiLat_poSet = fRelPoSetBy pr e s t
                 ,lSemiLat_inf   = fMeet
                 ,lSemiLat_botEl = fBot e s t
                 ,lSemiLat_atomS = filter pr (specialEls e (fBot e s t) $
                                                      lat_atomS e) }

fRelLSemiLat :: (Eq obj1, Eq obj2) =>
                Lat e -> [obj1] -> [obj2] -> LSemiLat (FRel e obj1 obj2)
fRelLSemiLat = fRelLSemiLatBy $ const True

fRelLSemiLatList :: (Eq e, Eq obj1, Eq obj2) =>
                    Lat e -> [FRel e obj1 obj2] -> LSemiLat (FRel e obj1 obj2)
fRelLSemiLatList e rs =
      let
        s = if null rs then [] else src $ head rs
        t = if null rs then [] else trg $ head rs
      in
        LSemiLat { lSemiLat_poSet = fRelPoSetList e rs
                  ,lSemiLat_inf   = fMeet
                  ,lSemiLat_botEl = fBot e s t
                  ,lSemiLat_atomS = specialEls e (fBot e s t) $ lat_atomS e }
```

There is nothing special to mention except the fact that we have to take care with the computation of the atoms with `fRelLSemiLatBy`. Since the $\mathcal{L}$-fuzzy relations between given source and target fulfilling a certain predicate may form a true sub(semi)lattice of the lattice of all $\mathcal{L}$-fuzzy relations, we have to filter out the atoms fulfilling the predicate `pr`.

Analogously we provide standard routines to determine the lattice of certain $\mathcal{L}$-fuzzy relations between given source and target.

```
fRelLatBy :: (Eq obj1, Eq obj2) =>
             (FRel e obj1 obj2 -> Bool) -> Lat e ->
             [obj1] -> [obj2] -> Lat (FRel e obj1 obj2)
fRelLatBy pr e s t =
     Lat { lat_poSet   = fRelPoSetBy pr e s t
          ,lat_sup     = fJoin
          ,lat_inf     = fMeet
          ,lat_botEl   = fBot e s t
          ,lat_topEl   = fTop e s t
          ,lat_atomS   = filter pr (specialEls e (fBot e s t) $ lat_atomS e)
          ,lat_jIrredS = filter pr (specialEls e (fBot e s t) $ lat_jIrredS e)
          ,lat_mIrredS = filter pr (specialEls e (fTop e s t) $ lat_mIrredS e) }

fRelLat :: (Eq obj1, Eq obj2) =>
           Lat e -> [obj1] -> [obj2] -> Lat (FRel e obj1 obj2)
fRelLat = fRelLatBy $ const True

fRelLatList :: (Eq e, Eq obj1, Eq obj2) =>
               Lat e -> [FRel e obj1 obj2] -> Lat (FRel e obj1 obj2)
fRelLatList e rs =
      let
        s = if null rs then [] else src $ head rs
        t = if null rs then [] else trg $ head rs
      in Lat { lat_poSet   = fRelPoSetList e rs
              ,lat_sup     = fJoin
              ,lat_inf     = fMeet
              ,lat_botEl   = fBot e s t
              ,lat_topEl   = fTop e s t
              ,lat_atomS   = specialEls e (fBot e s t) $ lat_atomS e
              ,lat_jIrredS = specialEls e (fBot e s t) $ lat_jIrredS e
              ,lat_mIrredS = specialEls e (fTop e s t) $ lat_mIrredS e }
```

3.6 Relational categories of $\mathcal{L}$-fuzzy relations

In this section we provide suitable instances of certain relational categories using the module `LFuzzyRel`. We then have the bridge to a component-free (i.e., algebraic) treatment of $\mathcal{L}$-fuzzy relations.

```
module LFuzzyRelCategories (fRelCat,fRelAll,fRelDistrAll,fRelDivAll,fRelDedCat,
                            fRelGogCat,gogCatFromAnti) where

import LFuzzyRel
import LFuzzyRelLattices
import RelAlg
import List (nub)
import Goguen
import Lattice
```

From the export list one can see the relational categories covered by this module. Furthermore, the function `gogCatFromAnti` is exported. It implements the standard construction of a Goguen category from given antimorphisms between the set of scalars over an entry lattice $\mathcal{L}$ and the lattice of all crisp $\mathcal{L}$-fuzzy relations over $\mathcal{L}$.

We start with categories.

```
fRelCat :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
           Cat [obj] (FRel e obj obj)
fRelCat entryLat allMs =
   let objs = nub $ foldr (\(rel:rels) -> (src rel:) . (trg rel :)) [] $
              filter (not.null) allMs
       test o1 o2 f = if elem o1 objs && elem o2 objs then f
                      else error "Illegal objects!"
   in Cat { cat_isObj   = flip elem $ objs
           ,cat_isMor   = \o1 o2 z -> test o1 o2 $
                             elem z (head $ dropWhile (\(m:ms) -> (src m/=o1) ||
                                                                   (trg m/=o2)
                                                      ) allMs)
           ,cat_objects = objs
           ,cat_homset  = \o1 o2 -> test o1 o2 $
                             let l = dropWhile (\(m:ms) -> (src m/=o1) ||
                                                           (trg m/=o2)   ) allMs
                             in if null l then [] else head l
           ,cat_source  = src
           ,cat_target  = trg
           ,cat_idmor   = fId entryLat
           ,cat_comp    = fComp }
```

From the typing we can see the step from the componentwise consideration to the algebraic level. As we aim at *standard* routines, we avoid unnecessarily manifold parametrizations. Obviously, we only need the entry lattice `entryLat` and all morphisms (i.e., all $\mathcal{L}$-fuzzy relations) `allMs` over `entryLat` that shall form the category. Notice that we demand the morphisms of `allMs` to be ordered by source and target. Remember that the functions `getFRelsBy` and `getAllFRelsBy` (cf. Section 3.4), respectively, can be used to support the determination of `allMs`.

Finally, we are able to easily compute the list of objects `objs` and the remaining functions of the data structure `Cat`.

The rest can be implemented analogously. The different allegories are provided by the next three functions.

```
fRelAll :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
                             All [obj] (FRel e obj obj)
fRelAll entryLat allMs = All { all_cat      = fRelCat entryLat allMs
                              ,all_converse = fConv
                              ,all_meet     = fMeet
                              ,all_incl     = fIncl }

fRelDistrAll :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
                                  DistrAll [obj] (FRel e obj obj)
fRelDistrAll entryLat allMs =
  DistrAll { distrAll_all     = fRelAll entryLat allMs
            ,distrAll_bottom  = fBot entryLat
            ,distrAll_join    = fJoin
            ,distrAll_atomset = \s t -> lat_atomS $ fRelLat entryLat s t
            ,distrAll_atoms   = \m -> let fLat = fRelLat entryLat (src m) (trg m)
                                      in filter (\y -> lat_lEq fLat y m) $
                                                lat_atomS fLat }

fRelDivAll :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
              DivAll [obj] (FRel e obj obj)
fRelDivAll entryLat allMs =
    DivAll { divAll_distrAll = fRelDistrAll entryLat allMs
            ,divAll_rres = fRRes
            ,divAll_lres = fLRes
            ,divAll_syq  = fSyQ }
```

The implementations are straightforward. We use the standard functions of the module `LFuzzyRel` to compute the resulting allegory. The same is true for the distributive allegory. But, here we additionally need the `Lattice` and `LFuzzyRelLattices` modules to determine the atom set of the given $\mathcal{L}$-fuzzy relations between source and target (`distrAll_atomset`) resp. the atoms that can be used to compute a given morphism (`distrAll_atoms`). For division allegories we need the residual operations. Here we use the standard implementations `fRRes`, `fLRes` and `fSyQ`, respectively, from the module `LFuzzyRel`.

Finally, Dedekind and Goguen categories can be instantiated through the following functions. They again only use standard routines of the module `LFuzzyRel`.

```
fRelDedCat :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
                                Ded [obj] (FRel e obj obj)
fRelDedCat entryLat allMs = Ded { ded_divAll = fRelDivAll entryLat allMs

                                 ,ded_top    = fTop entryLat }

fRelGogCat :: (Eq e, Eq obj) => Lat e -> [[FRel e obj obj]] ->
                                         Gog (Loos e) [obj] (FRel e obj obj)
fRelGogCat entryLat allMs = Gog { gog_ded    = fRelDedCat entryLat allMs
                                 ,gog_up     = fUp
                                 ,gog_down   = fDown
                                 ,gog_derOp  = fDerOp }
```

At the end we provide a function to generate a Goguen category $\mathcal{G}$ from given antimorphisms. For this construction we normally would have to use antimorhpisms $f : Sc_{\mathcal{G}}(A) \to Crisp_{\mathcal{G}}[A, B]$ for given objects A and B. For $\mathcal{L}$-fuzzy relations this corresponds to the computation $R = \bigsqcup\limits_{u\in\mathcal{L}} \alpha^u_A; f(\alpha^u_A)$ known from Section 2.4. This is rather inefficient. But, we know that the set of scalars of a given source is isomorphic to the set of elements of the entry lattice. We use this fact to gain efficiency here. Consider the following lemma.

Lemma 3.6.1. *Let $\mathcal{L}$ be a complete Brouwerian lattice, $R : A \to B$ be an $\mathcal{L}$-fuzzy relation and $f : \mathcal{L} \to Crisp_{\mathcal{L}}[A, B]$ be an antimorphism such that $R = \bigsqcup\limits_{u\in\mathcal{L}} \alpha^u_A; f(u)$. Then we have $R = \bigsqcup\limits_{u\in\mathcal{L}} (\mathbb{T}^u_{A,A} \sqcap f(u))$.*

Proof. This is immediately seen by the computation

$$\begin{aligned}
R(x, z) &= [\bigsqcup_{u\in\mathcal{L}} (\alpha^u_A; f(u))](x, z) \\
&= \bigvee_{u\in\mathcal{L}} \bigvee_{y\in A} (\alpha^u_A(x, y) \wedge f(u)(y, z)) && \text{(definition ;)} \\
&= \bigvee_{u\in\mathcal{L}} (\alpha^u_A(x, x) \wedge f(u)(x, z)) && \text{(definition } \alpha^u_A) \\
&= \bigvee_{u\in\mathcal{L}} (u \wedge f(u)(x, z)) && \text{(definition } \alpha^u_A) \\
&= \bigvee_{u\in\mathcal{L}} (\mathbb{T}^u_{AB}(x, z) \wedge f(u)(x, z)) && \text{(definition } \mathbb{T}^u_{AB}) \\
&= [\bigsqcup_{u\in\mathcal{L}} (\mathbb{T}^u_{AB} \sqcap f(u))](x, z).
\end{aligned}$$

□

Notice that we do not need the antimorphism property of f in the proof. The lemma indeed is true for arbitrary mappings. We used antimorphisms only for convenience.
Hence, we can finally compute a Goguen category given by certain antimorphisms.

```
gogCatFromAnti :: (Eq e, Eq obj) => Lat e ->
                  [[e -> FRel e obj obj]] -> Gog (Loos e) [obj] (FRel e obj obj)
gogCatFromAnti e ms = fRelGogCat e $ map (computeRels $ lat_jIrredS e) ms
  where
    computeRels ir morphs = [foldl1 fJoin [fUpdAllUnBy (lat_inf e irEl) (m irEl)
                                          | irEl<-ir] | m<-morphs]
```

Chapter 4

Fuzzy Control Based on Goguen Categories

In this chapter we want to examine a very important application of Goguen categories. Fuzzy controllers have become a widespread interest of research over the last years. Their applications in the real world are manyfold and reach from controllers for traffic lights to washing machines. With these applications, fuzzy controllers often have to steer safety sensitive devices. Hence, it would be nice to have an algebraic framework for such controllers to make it possible to prove certain properties. M. Winter in [12] showed that the algebra of Goguen categories is suitable for this purpose.
After an introduction to fuzzy control, we present an algebraic model for a certain kind of fuzzy controller. Then we are ready to provide the mathematical base for the construction of the Haskell module. We introduce certain operations on fuzzy controllers and show their behavior. Finally, we provide a framework to construct and test such controllers within the algebraic surrounding of Goguen categories.

4.1 Introduction to fuzzy controllers

In the following we want to have a look at fuzzy controllers based on the *linguistic model* (explained below). We want to mention that there are also other approaches. But, they are not suitable for our purposes and, hence, we refer to the generous literature for further reading (e.g., [7]).
The linguistic model is interesting for us since it induces a kind of relational semantics.

Hence, it can be represented by certain $\mathcal{L}$-fuzzy relations. In the following we explain general controllers based on the linguistic model and then switch to so called *Mamdani* controllers which are of special interest in practice as we will see later on. The structure of a fuzzy controller is shown in Figure 4.1. The controller has to get an input value (e.g. a velocity),

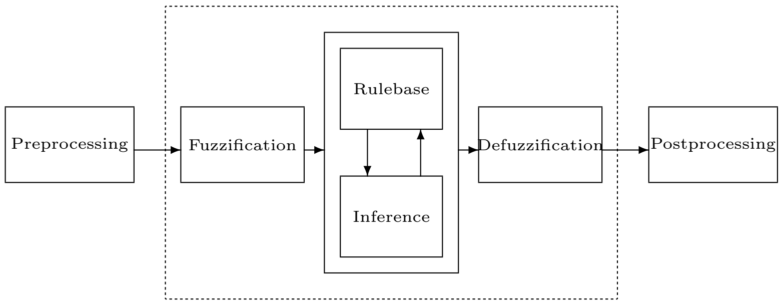

Figure 4.1: The general structure of a fuzzy controller

which often has to be extracted from a technical process. This is mostly done using certain measuring devices. These devices then transform an analog signal into a digital one, which can be interpreted by the controller. This whole procedure of generating an interpretable input value is called *preprocessing*. With the linguistic model, the input variables are called *linguistic variables*.

Having the input, the fuzzy controller has to transform the input value into an output value, which then can be used to steer the technical process. To do so, *linguistic entities* are used. They are fuzzy sets ranging over the input resp. output space of the controller. For a controller that regulates the speed of a certain vehicle, one could, for example, introduce the input entities LOW and $HIGH$. An entry in the corresponding fuzzy set then gives the degree to which the vehicle is slow resp. fast. The table below shows exemplary how these entities are interpreted.

$v =$	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**	**110**
$LOW =$	{0.9,	0.8,	0.7,	0.6,	0.5,	0.4,	0.3,	0.2,	0.1,	0.0,	0.0}
$HIGH =$	{0.1,	0.2,	0.3,	0.4,	0.5,	0.6,	0.7,	0.75,	0.8,	0.85,	0.9}

With the linguistic entities for the input, the controller is able to interprete given input values. Mapping the input to the input entities is also called *fuzzification*.

The next step is to react on the input and generate appropriate steering signals. Again, we need linguistic entities (output entities). In our example, this could be $SLOWER$ and $FASTER$. Mapping the output entities to the output value then is called *defuzzification*.

But, we furthermore need some rules, which express when the vehicle should go slower or faster. For this purpose, we have to provide a *rulebase*, which includes a *finite* number of rules of the form

If x_i is Q_j then y_i is S_k

where x_i resp. y_i are linguistic variables for the input and output, respectively, and Q_j resp. S_k are input resp. output entities.
Last but not least, the *inference engine* decides to which degree a certain rule is applicable. Rulebase and inference engine together are often called the *core* of a controller. Obviously, these two parts are the most important ones.
Finally, the output value has to be transformed into a steering signal, which the technical process can understand. This part is summarized under the notion of *postprocessing.*
After this short overview we want to go into detail with the rulebase and inference engine.

Inference within the linguistic model

The main question to answer while constructing a fuzzy controller is how the input and output entities, respectively, shall be connected so that the controller shows the intended behavior. With this it is important to see that the "If ... then ..." rulebase does not necessarily correspond to the mathematical implication $\Rightarrow$. We rather want to express a causal connection like

If *speed* is *HIGH* then *braking distance* is *LONG.*

Using this expression we mean that high speed induces a long braking distance. Furthermore, we want to model that the braking distance decreases with the speed. It is clear that this fact cannot be expressed using $\Rightarrow$. This becomes even clearer if we consider the equivalence of $x \Rightarrow y$ and $\neg x \vee y$.
Our intention with this example can be expressed more generally by using the *generalized modus ponens* introduced in [4]. A fuzzy controller has two inputs at any time — the measured input value from the technical process and the rulebase. The inference of the output using a certain input and a rule can be visualized by the following scheme :

$$\frac{\begin{array}{c}\text{If } x \text{ is } F \text{ then } y \text{ is } G\\ x \text{ is } F'\end{array}}{y \text{ is } G'}$$

In the example above, the rule

$$\frac{\begin{array}{c}\text{If } \textit{speed} \text{ is } HIGH \text{ then } \textit{braking distance} \text{ is } HIGH\\ \textit{speed} \text{ is } LOW\end{array}}{\textit{braking distance} \text{ is } LOW}$$

shall be valid. In contrast, the following output shall not be inferred.

$$\frac{\begin{array}{c}\text{If } \textit{speed} \text{ is } HIGH \text{ then } \textit{braking distance} \text{ is } HIGH\\ \textit{speed} \text{ is } MIDDLE\end{array}}{\textit{braking distance} \text{ is } VERY\ HIGH}$$

These examples show that the inference engine is a non-trivial construct and strongly depends on the application. Furthermore, we see that after modeling at least one correctness proof has to be brought about; the following shall hold for all rules in the rulebase :

$$\frac{\begin{array}{c}\text{If } x \text{ is } F \text{ then } y \text{ is } G\\ x \text{ is } F\end{array}}{y \text{ is } G}$$

This formula obviously expresses the modus ponens. If it is true for a rule, the controller is called to be *locally correct* for this rule. Analogously, it is *totally correct* if it is locally correct for all rules.

We now want to describe how to model such a rule inference in the general case. The first thing we need is a function to interprete the implicative meaning of the "If ... then ..." phrase. This function can differ from rule to rule, and we do not yet want to restrict it to be of a specific form. Let "If x is F then y is G" be a rule, $\mathcal{L}$ a lattice and F resp. G $\mathcal{L}$-fuzzy sets over given universes U resp. V. Furthermore, let $\odot : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ be a function. Then we can define the matrix

$$M_{\odot}^{F,G}(x, y) := F(x) \odot G(y)$$

which obviously fixes to which degree an element of the input fuzzy set F shall be mapped to an element of the output set G. This shows that $\odot$ really determines the implicative behavior of the corresponding rule. Hence, we call it the *implication function*. Notice that $\odot$ induces a crisp relation R on $\mathcal{L} \times \mathcal{L}$ with $(a_1, a_2) \in R \ :\Leftrightarrow\ a_1 \odot a_2 \neq 0_{\mathcal{L}}$. In the following we write $(a_1, a_2) \in \odot$ to express this fact. One often uses $\odot = \wedge$ which seems to be counterintuitive at first sight. But, consider the case where F and G are crisp and $\odot$ is the *min* operator. Then $M_{\odot}^{F,G}$ is nothing more than the matrix representing the cross product $F \times G$.

Now, we can proceed with modeling the inference. Let the input F' be an $\mathcal{L}$-fuzzy set over U. Using the rule from above, we want to deduce the corresponding output. An element $y \in V$ shall be in the output set G', if there is an $x \in U$ such that $x \in F'$ and $(x, y) \in \odot$. In other words : The element y shall be in the output set, if there is an x in the input set such that the fact that x is the input implies y is the output. In the theory of $\mathcal{L}$-fuzzy relations, this phrase can be expressed using a certain t-norm like loos ($\mathcal{L}$,$*$,$1_{\mathcal{L}}$,$0_{\mathcal{L}}$) as "and" operator. We define

$$G'(y) := \bigvee\{x \in U | F'(x) * (F(x) \odot G(y))\}$$

The output G' obviously directly corresponds to the composition $F';_* M_\odot^{F,G}$ using the derived operation $;_*$. The standard variant is given by $* = \wedge$ and, hence, $;_* =;$. With this approach, one has to show $F; M_\odot^{F,G} = G$ to prove partial correctness of the rule "If x is F then y is G". Notice that others than the existence quantifier are thinkable in the formula above. But, the existence quantifier is mostly used with actual applications.

The next step is to extend the rulebase such that it contains a finite number of rules. We then have to examine how to connect these rules to a globel inference engine. Let

$$\begin{array}{l} \text{If } x \text{ is } F_1 \text{ then } y \text{ is } G_1 \\ \text{If } x \text{ is } F_2 \text{ then } y \text{ is } G_2 \\ \qquad\vdots \\ \text{If } x \text{ is } F_n \text{ then } y \text{ is } G_n \end{array}$$

be our new rulebase whereas $F_1, ..., F_n$ resp. $G_1, ..., G_n$ are $\mathcal{L}$-fuzzy relations over U resp. V. Due to the explanations above, this induces n matrices

$$\begin{aligned} M_{\odot_1}^{F_1,G_1}(x,y) &:= F_1(x) \odot_1 G_1(y) \\ M_{\odot_2}^{F_2,G_2}(x,y) &:= F_2(x) \odot_2 G_2(y) \\ &\vdots \\ M_{\odot_n}^{F_n,G_n}(x,y) &:= F_n(x) \odot_n G_n(y). \end{aligned}$$

Now, let F' be the input of our controller. It is clear that we have to combine this input with every single rule to infer the output. This is done by combining (*aggregating*) all interpretaions of the rules to a global matrix and then applying the input to this matrix. The aggregation is done by a given function $\oplus$ and the new interpretation matrix is defined by

$$M_\oplus(x,y) := \oplus(M_{\odot_1}^{F_1,G_1}(x,y), ..., M_{\odot_n}^{F_n,G_n}(x,y)).$$

In the following we denote the to matrices extended version of $\oplus$ by $\biguplus$. Notice that this extension is well-defined since the rulebase (and, thus, the resulting matrix) is finite. Hence, $M_\oplus$ can be computed by $\biguplus_i M_{\odot_i}^{F_i,G_i}$.
Finally, the output is inferred using the formula

$$G'(F') := F';_* M_\oplus. \tag{4.1}$$

Mamdani inference

The considerations of the last section show that rule inference based on fuzzy implication induces a relational calculus to infer the output (cf. Formula 4.1) where we have to store the additional relation $M_\oplus$. The relational computation may cause problems in the case that the input sets range over a non-discrete universe. This would induce that M has both non-discrete source and target, respectively. Since the relational calculus is only applicable for relations of discrete source and target, one would have to discretize the universe. This can be critical for analytical considerations to show a specific behaviour of a given controller.
An escape can be found by restricting the implication and aggregation function (cf. [8]) which results in the *Mamdani inference*. Using the rulebase of the last section, the standard Mamdani model is given by the following restrictions :

(1) $\odot_i = \wedge$,
(2) $\oplus = \vee$,
(3) $* = \wedge$.

Hence, we have

$$\begin{array}{rcl} M_\wedge^{F_i,G_i}(x,y) & = & F_i(x) \wedge G_i(y) \\ M_\vee(x,y) & = & \bigvee\limits_{i\in\{1,\dots,n\}} M_\wedge^{F_i,G_i}(x,y) \end{array}$$

and finally can derive the output for a given input F' by

$$G'(F') = F'; M_\vee.$$

In the following we omit the indices and only write M^{F_i,G_i} for $M_\wedge^{F_i,G_i}$ resp. M for $M_\vee$. Now, we want to consider the last formula on the level of components and show how the output inference can be simplified. The little computation

$$\begin{array}{rcl} G'(F')(y) & = & (F'; M)(y) \\ & = & \bigvee\limits_{x\in U} (F'(x) \wedge (\bigvee\limits_{i\in\{1,\dots,n\}} M^{F_i,G_i}(x,y))) \\ & = & \bigvee\limits_{x\in U} \bigvee\limits_{i\in\{1,\dots,n\}} (F'(x) \wedge M^{F_i,G_i}(x,y)) \\ & = & \bigvee\limits_{x\in U} \bigvee\limits_{i\in\{1,\dots,n\}} (F'(x) \wedge F_i(x) \wedge G_i(y)) \end{array}$$

$$\begin{aligned} &= \bigvee_{i\in\{1,\ldots,n\}} ((\bigvee_{x\in U} (F'(x) \wedge F_i(x))) \wedge G_i(y)) \\ &:= \bigvee_{i\in\{1,\ldots,n\}} (\beta_i \wedge G_i(y)) \end{aligned}$$

shows that we can extract the supremum over all i to the left and, hence, avoid to explicitly store the aggregation matrix M and to use the relational calculus. This implies that a discretization of the universe is not necessary and one can use analytically defined membership functions. Notice that β_i delivers the degree of fulfillment of rule i. The last considerations have made Mamdani inference (often called *max-min*-inference) popular in real applications.

4.2 A relational model for fuzzy controllers

From the introduction to fuzzy controllers we see that the linguistic model is good to handle by $\mathcal{L}$-fuzzy relations and, thus, by Goguen categories. We aim at a module to support creating and testing controllers. This module shall rely on a mathematical base. Hence, we have to provide a suitable algebraic model. We will see that some restriction on the fuzzy controllers allow nice handling.
Michael Winter in [12] gave a proposal how to model controllers based on the linguistic model within Goguen categories. We want to take over this model and extend it at some points. In this approach the linguistic entities for the input as well as for the output are modelled by $\mathcal{L}$-fuzzy relations $Q : I \to A$ where A is an arbitrary object of the underlying Goguen category and I is a unit (i.e., $\mathbb{I}_I = \mathbb{T}_{II}$ and $\mathbb{T}_{BI}$ is total for all objects B). Hence, the interpretation of a rule "If x is Q then y is S" in the general case corresponds to the computation $Q^{\smile};_{\odot} S$ using a derived composition-based operation. With the standard model for Goguen categories this corresponds to the matrix $M_{\odot}^{Q,S}$ introduced in Section 4.1.
Now, we extend the rulebase to be of the form

if x is Q_1 then y is S_1
if x is Q_2 then y is S_2
$\vdots$
if x is Q_n then y is S_n.

whereas all rules are interpreted by their own interpretation function $;_{\odot_i}$. The aggregation in this model corresponds to a $\sqcap$-based derived operation $\sqcap_{\oplus}$. Since we want to restrict $\oplus$

to have neutral element $\perp\!\!\!\perp$ (i.e., to be t-conorm like), we denote the derived operation by $\sqcup_\oplus$. The extension to arbitrary sets is denoted by $\biguplus$. Hence, the core of the controller can be computed by

$$C := \biguplus_{i\in\{1,..,n\}} (Q_i^\smile ;_{\odot_i} S_i). \tag{4.2}$$

So far we can see that this model is a direct realization of the relational semantics of the linguistic model introduced in the last section. We have only lifted all operations to a component-free niveau.

If all interpretation functions are cloos-based (i.e., $\odot_i$ preserves suprema, $1 \leq i \leq n$), a rulebase can be represented in quite a nice way using a crisp relation. If we, for example, have three input entites $Q_1, Q_2, Q_3 : I \to A$ and two output entities $S_1, S_2 : I \to B$, the rules

if x is Q_1 then y is S_1
if x is Q_2 then y is S_2
if x is Q_3 then y is S_2

can be represented by the relation

$$R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

Hence, R has source $I+I+I$ (for the three input entities) and target $I+I$ (for the output entities). In an algebraic way R can be computed using the *crisp* injections $\iota_i : I \to I+I+I$ resp. $\kappa_j : I \to I+I$ of the corresponding input resp. output entities. We then have $R = \iota_1^\smile;\kappa_1 \sqcup \iota_2^\smile;\kappa_2 \sqcup \iota_3^\smile;\kappa_2$ or, more generally,

$$R = \bigsqcup_{\substack{i,j \\ \textit{if } x \textit{ is } Q_i \textit{ then } y \textit{ is } S_j \\ \textit{is a rule}}} \iota_i^\smile;\kappa_j.$$

Thus, the core of the respective controller can be computed by

$$C' := \biguplus_{\substack{i,j \\ j\in R(i)}} ((Q_i^\smile;\iota_i);_{\odot_i} (R;(\kappa_j^\smile;S_j))) \tag{4.3}$$

where i ranges over the input and j over the output entities. The following lemma shows that Formulae 4.2 and 4.3 are equivalent if additionally $\mathbb{I}$ is a left and right neutral element for $\odot_i$ and $\odot_i$ is associative for all $1 \leq i \leq n$, i.e., $\odot_i$ is closg-based.

Lemma 4.2.1. *Let the rulebase and operations be given as above. Furthermore, $\mathbb{I}$ shall be left and right neutral element for the closg-based operations $\odot_1, ..., \odot_n$. Then we have $C = C'$.*

Proof. Throughout the proof we will use $*$ to refer to the fact that $\iota_i;\iota_{j^\smile} = \perp\!\!\!\perp$ if $i \neq j$ for a pair of crisp injections. Hence, the computation

$$\begin{aligned}
\biguplus_{\substack{i,j\\ j\in R(i)}} ((Q_i^\smile;\iota_i);_{\odot_i}(R;(\kappa_j^\smile;S_j))) &= \biguplus_{\substack{i,j\\ j\in R(i)}} ((Q_i^\smile;\iota_i);_{\odot_i}(\bigsqcup_{\substack{k,l\\ l\in R(k)}} (\iota_k^\smile;\kappa_l);(\kappa_j^\smile;S_j))) && \text{(definition } R\text{)}\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} ((Q_i^\smile;\iota_i);_{\odot_i}(\bigsqcup_{\substack{k\\ R(i)\cap R(k)\neq\emptyset}} (\iota_k^\smile;S_j))) && (*)\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} (\bigsqcup_{\substack{k\\ R(i)\cap R(k)\neq\emptyset}} (Q_i^\smile;\iota_i);_{\odot_i}(\iota_k^\smile;S_j)) && (\odot_i \text{ complete})\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} (\bigsqcup_{\substack{k\\ R(i)\cap R(k)\neq\emptyset}} (Q_i^\smile;_{\odot_i}\iota_i);_{\odot_i}(\iota_k^\smile;_{\odot_i}S_j)) && \text{(Lem. 2.6.5)}\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} (\bigsqcup_{\substack{k\\ R(i)\cap R(k)\neq\emptyset}} (Q_i^\smile;_{\odot_i}(\iota_i;_{\odot_i}\iota_k^\smile);_{\odot_i}S_j)) && (\odot_i \text{ ass.})\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} (\bigsqcup_{k,i=k} (Q_i^\smile;_{\odot_i}(\iota_i;\iota_k^\smile);_{\odot_i}S_j)) && \text{(Lem. 2.6.5,}*\text{)}\\
&= \biguplus_{\substack{i,j\\ j\in R(i)}} (Q_i^\smile;_{\odot_i}S_j)
\end{aligned}$$

shows the result. □

We want this equation to be valid with our fuzzy controllers. Hence, we demand all interpretation functions $\odot_i$ to satisfy

(1) $\odot_i$ is closg-based, $1 \leq i \leq n$,

(2) $\mathbb{I}$ is left and right neutral element for $\odot_i$, $1 \leq i \leq n$.

It is clear that Formula 4.2 is the more likely variant with respect to computational purposes. Thus, it will be used within our module for fuzzy controllers later on. In contrast, the second alternative is better to visualize the developed model. Hence, we use it in Figure 4.2 to show the model developed so far.

In the following we call a controller with a core corresponding to this model a *simple controller*. We will give a formal definition later on.

Now, the question arises, how the output shall be inferred for a given input. Since we are dealing with a component-free view, the input value is represented by a crisp function

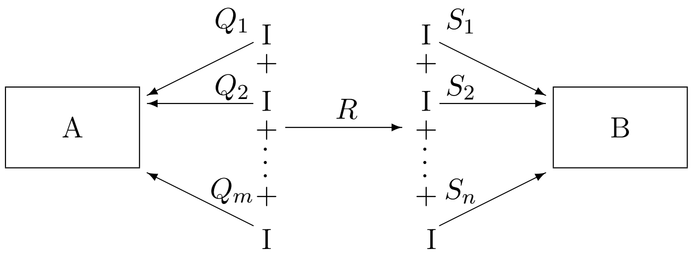

Figure 4.2: The core of a fuzzy controller

$x : I \to A$ whereas A is the source of the given controller and I is a unit. Thus, our input value is already an $\mathcal{L}$-fuzzy set so that $\mathbb{I}_A$ can be used as a special fuzzification function. There are other operations thinkable (e.g., relational shifting, weakening or strenghtening). The final decision which one to use strongly depends on the application. The fuzzification function will be denoted by Φ in the following.

The next step is to infer the output from the fuzzified input. This is, in general, done by

$$U(x) := \Phi(x);_* C$$

using a composition-based derived operation $;_*$.

But, the inferred output is neither necessarily crisp nor univalent. Hence, we have to extend the model so that a crisp output value is produced. In [11] the problem is divided into two aspects :

(1) The output shall be crisp.

(2) The output shall be univalent.

Again, many defuzzification functions are thinkable to generate a crisp output. But, one special possibility is delivered automatically by Goguen categories. If we take a function Θ which maps the input to a scalar $\Theta(x)$ on I we can compute

$$D(U(x)) := (\Theta(x)\backslash U(x))^{\downarrow}$$

which obviously delivers a crisp relation. Now, suppose $U(x) = \Phi(x);_* C$ with a closg-based operation $*$. Furthermore, assume that $\Phi(x) = x; F_\Phi$ for a certain F_Φ. If additionally $x^{\smile};\Theta(x) = S_A; x^{\smile}$ holds for a partial identity S_A we have $U(x) = x;(F_\Phi;_* C)$ and can conclude

$$D(U(x)) = x;(S_A\backslash(F_\Phi;_* C)).$$

This shows that the resulting controller is a relational term whereas the inference of the output is just done by the usual composition operator. A deduction of the formula above can be found in [11]. It is furthermore shown there that constant functions Θ as well as the function of maximal degree of membership $\Theta(x) = U(x); \mathbb{T}_{BI} \sqcap \mathbb{I}_I$ fulfill the property above.

It is clear that this approach does not necessarily deliver a univalent function. Hence, we have to dedicate some special thoughts to this property. The most general approach to achieve the goal is based on symmetric quotients and relational powers (introduced in [18]). Let $X : A \to B$ be a relation. If we compare the columns of $X^{\smile}$ and $\epsilon_{\mathbb{P}_B}$ via $syQ(X^{\smile}, \epsilon_{\mathbb{P}_B})$, we map every $a \in A$ to its image under X if X is total. If we now apply a function f to this result, we generate a univalent output. Thus, the final term is given by

$$D(U(x)) = syQ((\Theta(x) \backslash U(x))^{\downarrow}, \epsilon_{\mathbb{P}(B)}); f$$

if we use the cut approach as defuzzification. If again Θ and Φ fulfill the properties described above, $D(x)$ can be computed by

$$\begin{aligned} \Delta(U(x)) &= syQ((\Theta(x) \backslash U(x))^{\downarrow}, \epsilon_{\mathbb{P}(B)}); f \\ &= syQ(x; (S_A \backslash (F_\Phi ;_* C))^{\downarrow}, \epsilon_{\mathbb{P}(B)}); f \\ &= x; syQ((S_A \backslash (F_\Phi ;_* C))^{\downarrow}, \epsilon_{\mathbb{P}(B)}); f. \end{aligned}$$

At the end of this section we want to mention that the introduced approaches for fuzzification and defuzzification are only special alternatives. In general, we have abstract functions. Hence, the explanations above induce that a simple controller FC can be considered to be a tupel of the following form.

Definition 4.2.1. *Let I, A and B be objects of a Goguen category $\mathcal{G}$ such that I is a unit. Then we call a 7-tupel*

$$\begin{array}{lll} FC := (\ \Phi & : \mathcal{G}[I,A] \to \mathcal{G}[I,A] & \textit{(fuzzification)} \\ \quad , \ \Pi & : \mathcal{G}[I,A] \to \mathcal{G}[A,B] \to \mathcal{G}[I,B] & \textit{(application)} \\ \quad , \ L_{in} & : \{\mathcal{G}[I,A]\} & \textit{(ling. entities input)} \\ \quad , \ L_{out} & : \{\mathcal{G}[I,B]\} & \textit{(ling. entities output)} \\ \quad , \ R & : \{(\mathcal{G}[A,I] \to \mathcal{G}[I,B]) \to \mathcal{G}_[A,B]) \times Rule\} & \textit{(rulebase)} \\ \quad , \ \sqcup_\oplus & : \mathcal{G}[A,B] \to \mathcal{G}[A,B] \to \mathcal{G}[A,B] & \textit{(aggregation)} \\ \quad , \ \Delta & : \mathcal{G}[I,B] \to \mathcal{G}[I,B]) & \textit{(defuzzification)} \end{array}$$

a **simple controller** *iff*

(1) $\Delta(y)$ *is defined and crisp for all* $y \in \mathcal{G}[I, B]$*,*

(2) Π *is a derived operation* $;_*$ *such that* $*$ *is closg-based with left and right neutral element* $\mathbb{I}$*,*

(3) all interpretation functions in R *are derived operations* $;_{\odot_i}$ *such that* $\odot_i$ *is closg-based and* $\mathbb{I}$ *is left and right neutral element for* $\odot_i$*,*

(4) Φ *is a mapping,*

(5) the induced core $\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R}} (Q^{\smile};_\odot S)$ *is total.*

Obviously, the rulebase is represented by pairs consisting of an interpretation function $;_{\odot_i}$ and the corresponding rule i. The other functions should be clear. Restriction (1) is a direct conclusion from the explanations above. Property (3) assures that the interpretation functions are t-norm-based and, hence, the rulebase of FC can be represented by a crisp relation (cf. Equation 4.3). Finally, (4) and (5) together with (1) assure that each input can be interpreted by the controller.
Hence, we can compute C using L_{in}, L_{out}, R and the to sets of arguments extended aggregation function $\sqcup_\oplus$. Furthermore, we can infer the output for a given input x using the term

$$\begin{aligned} FC(x) &= \Delta(\Pi(\Phi(x))(C)) && (4.4)\\ &= \Delta(\Pi(\Phi(x))(\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R}} (Q^{\smile};_\odot S))). && (4.5) \end{aligned}$$

This implies that a simple controller in general constitutes a function $FC : \mathcal{G}[I, A] \to \mathcal{G}[I, B]$. Definition 4.2.1 and the last considerations will be the base for our Haskell module. In the following we often write FC and omit the induced tuple.
Finally, we want to mention that storing the linguistic entities within the tuple above is not necessary for the following mathematical considerations. But, we use the representation above as a data structure in the Haskell module later on. Hence, it seems advantageous to make the reader get familiar with it as early as possible.

4.2.1 Operations on fuzzy controllers

The simple model for fuzzy controllers induces certain operations. In this section we want to introduce suitable combinators which allow reasoning on the level of fuzzy controllers.

They constitute the mathematical base for the Haskell module developed later on.
The following standard constructions are thinkable :

(1) derived join ($\widehat{\sqcup_{\oplus'}}$),
(2) derived direct sum ($\widehat{+_{\sqcup_{\oplus'}}}$),
(3) derived meet ($\widehat{\sqcap_*}$),

and

(4) derived cross product ($\widehat{\times_{\sqcap_*}}$).

In a first account, these operations shall be operations on fuzzy controllers (cf. Definition 4.2.1), i.e., on tuples of certain elements and not necessarily on terms of the underlying Goguen category. We will examine conditions such that these operations are operations on the core and, hence, on given terms of the Goguen category. If we mean an operation to be on the level of fuzzy controllers, we designate it by an additional $\widehat{\ }$ (e.g., $\widehat{\sqcup_{\oplus'}}$). These operations naturally induce an operation on the cores under some circumstances. On this level we omit the $\widehat{\ }$ and only write $\sqcup_{\oplus'}$ instead of $\widehat{\sqcup_{\oplus'}}$, for example.
The next definition gives us the prerequisites we need throughout this section.

Definition 4.2.2. *Let FC_1 and FC_2 be two simple controllers of a Goguen category. We call FC_1 and FC_2* **combinable***, if the following holds :*

(1) $\sqcup^1_{\oplus} = \sqcup^2_{\oplus}$.
(2) All rules of FC_1 and FC_2 are interpreted by the same ;-based derived operation $;_{\odot}$.
(3) $R;(\biguplus_i x_i) = \biguplus_i (R;x_i)$ *if R is crisp and univalent, and*
$(\biguplus_i x_i);R = \biguplus_i (x_i;R)$ *if R is crisp and injective.*
(4) $(\biguplus_i x_i) \sqcap_{\odot} (\biguplus_j y_j) = \biguplus_{i,j}(x_i \sqcap_{\odot} y_j)$*, i.e., $\sqcup_{\oplus}$ distributes together with $\sqcap_{\odot}$.*
(5) $(Q \sqcap_{\odot} Q');_{\odot} (S \sqcap_{\odot} S') = Q;_{\odot} S \sqcap_{\odot} Q';_{\odot} S'$ *for all $Q, Q' : A \to I$ and $S, S' : I \to B$ whereas I is a unit.*
(6) $\perp\!\!\!\perp$ is left and right neutral element for $\oplus$.

Notice that the fact that FC_1 and FC_2 are simple controllers implies that $\odot$ is closg-based. Property (1) demands FC_1 and FC_2 to have the same aggregation function. Otherwise the respective function for the resulting controller could not be determined properly. The necessity of restrictions (2)-(6) will become clear with the following explanations. But, already notice that $\sqcup_{\oplus} = \sqcup$, $\sqcap_{\odot} = \sqcap$ and $;_{\odot} =;$ fulfill these properties. That (3),(4) and (6) are satisfied should be clear and (5) is shown by the following lemma.

Lemma 4.2.2. *Let $Q, Q' : A \to I$ and $S, S' : I \to B$ be relations such that I is a unit. Then we have*

$$(Q \sqcap Q'); (S \sqcap S') = Q; S \sqcap Q'; S'$$

Proof. From $Q^{\smile}; Q \sqsubseteq \top\!\!\!\top_{II} = \mathbb{I}_I$ and $S; S^{\smile} \sqsubseteq \top\!\!\!\top_{II} = \mathbb{I}_I$ we know that Q and Q' (thus, $Q \sqcap Q'$) are univalent and S and S' (thus, $S \sqcap S'$) are injective. Using this fact we can compute

$$\begin{aligned}
Q; S \sqcap Q'; S' &\sqsubseteq Q; (S \sqcap Q^{\smile}; Q'; S') && \text{(modular law)}\\
&\sqsubseteq Q; (S \sqcap \top\!\!\!\top_{II}; S') &&\\
&= Q; (S \sqcap \mathbb{I}_I; S') && (I \text{ unit})\\
&= Q; S \sqcap Q; S' && (Q \text{ univalent})\\
Q'; S' \sqcap Q; S &\sqsubseteq Q'; (S' \sqcap Q'^{\smile}; Q; S) && \text{(modular law)}\\
&\sqsubseteq Q'; (S' \sqcap \top\!\!\!\top_{II}; S) &&\\
&= Q'; (S' \sqcap \mathbb{I}_I; S) && (I \text{ unit})\\
&= Q'; S' \sqcap Q'; S. && (Q' \text{ univalent})
\end{aligned}$$

Finally, we have

$$\begin{aligned}
Q; S \sqcap Q'; S' &= Q; S \sqcap Q; S' \sqcap Q'; S' \sqcap Q'; S && \text{(above)}\\
&= Q; (S \sqcap S') \sqcap Q'; (S \sqcap S') && (Q, Q' \text{ univalent})\\
&= (Q \sqcap Q'); (S \sqcap S'). && (S \sqcap S' \text{ injective})
\end{aligned}$$

□

Hence, we conclude that Mamdani inference is covered by the following considerations and all results for the examined operations can be applied.

The following definition connects combinable controllers and operations on them.

Definition 4.2.3. *Let FC_1 and FC_2 be two combinable fuzzy controllers with aggregation function $\sqcup_\oplus$ and interpretation function $\odot$. Furthermore, let $Op := (\widehat{\sqcup_{\oplus'}}, \widehat{+_{\sqcup_{\oplus'}}}, \widehat{\sqcap_*}, \widehat{\times_{\sqcap_*}})$ be a 4-tupel of binary operations on fuzzy controllers. Then Op* **combines** *FC_1 and FC_2 If the conditions*

(1) $\sqcup_{\oplus'} = \sqcup_\oplus$, i.e., $\oplus' = \oplus$
(2) $* = \odot$

are satisfied.

Throughout this section let $\mathcal{G}$ be a Goguen category, FC_1 and FC_2 be two combinable controllers and Op a tupel of operations such that Op combines FC_1 and FC_2. If we use components that are equal with FC_1 and FC_2 (e.g., Φ_1), we automatically omit the index. The to finite sets of elements extended variant of $\sqcup_\oplus$ is again denoted by $\biguplus$.

The derived join operation

To apply the join operation the two controllers need to have the same source and target as well as the same fuzzification, defuzzification and application functions. Hence, we additionally suppose both FC_1 and FC_2 to be of the form $\mathcal{G}[I,A] \to \mathcal{G}[I,B]$ with $\Phi_1 = \Phi_2$, $\Delta_1 = \Delta_2$, $\Pi_1 = \Pi_2$. In the following we omit the indices for the respective functions. Notice that a join operation on two fuzzy controllers after defuzzification makes no sense since we need to have the same output with both controllers to make the joined output univalent. This induces that this operation can only be an operation on the core resp. on the linguistic entities and the rulebase. We first define $\widehat{\sqcup_\oplus}$ on the level of the rulebase.

Definition 4.2.4. *Let FC_1 and FC_2 be two combinable controllers. Then we define*

$$FC_1 \widehat{\sqcup_\oplus} FC_2 := (\Phi, \Pi, L_{in_1} \cup L_{in_2}, L_{out_1} \cup L_{out_2}, R_1 \cup R_2, \sqcup_\oplus, \Delta).$$

We denote the respective elements of the resulting controller by the subscript $\widehat{\sqcup_\oplus}$ (e.g., $R_{\widehat{\sqcup_\oplus}}$ for the rulebase).

The little computation

$$\begin{aligned} C_{\widehat{\sqcup_\oplus}} &= \biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1 \cup R_2}} (Q^\smile ;_\odot S) \\ &= \biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} ((Q^\smile ;_\odot S)) \sqcup_\oplus \biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} (Q'^\smile ;_\odot S') \\ &= C_1 \sqcup_\oplus C_2 \end{aligned}$$

shows that $\widehat{\sqcup_\oplus}$ induces an operation $\sqcup_\oplus$ on the cores. Obviously, prerequisites (1) of Definition 4.2.3 and (1),(2) of Definition 4.2.2 from above are the key properties that this can hold.

The derived direct sum

The next step is to provide separate input/output spaces for separate input and output entities, respectively. We can omit the additional restrictions of the last section, i.e., the fuzzification, defuzzification and application functions of FC_1 and FC_2 need not necessarily be identical. Michael Winter in his example in [11], for example, provided some output entities for regulating a temperature and a seperate output entity for an alert signal. In the general case this corresponds to the model shown in Figure 4.3.

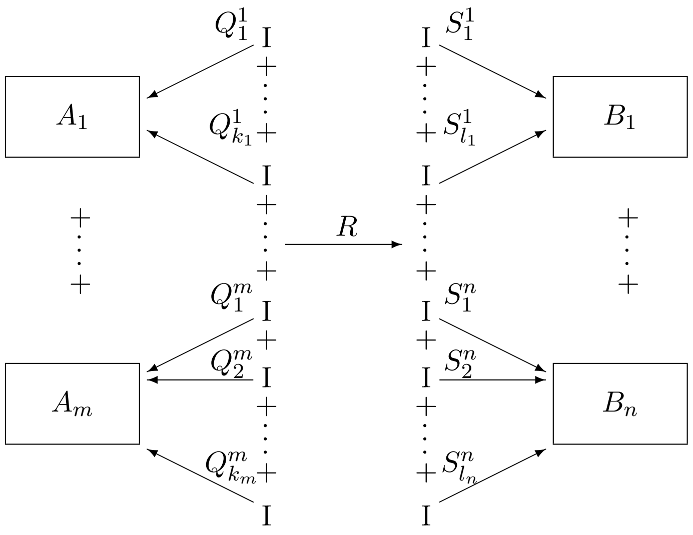

Figure 4.3: The direct sum of the cores of two controllers

One could say that these controllers can equivalently be described by controllers with only one range for the input resp. output entities using the crisp injections $\iota_i : A_i \to A_1 + ... + A_m, 1 \le i \le m$ resp. $\kappa_j : B_j \to B_1 + ... + B_n, 1 \le j \le n$. But, we aim at a comfortable treatment of fuzzy controllers with our module. Hence, we support the user with automatic computations where it is possible. A controller of the form shown in Figure 4.3 will be called *sum controller* in the following.

Now, we define the derived direct sum operation on the level of fuzzy controllers.

Definition 4.2.5. *Let $FC_1 : \mathcal{G}[I, A_1] \to \mathcal{G}[I, B_1]$ and $FC_2 : \mathcal{G}[I, A_2] \to \mathcal{G}[I, B_2]$ be two combinable controllers. Furthermore, let $\iota_i : A_i \to A_1 + A_2$, $\kappa_i : B_i \to B_1 + B_2$, $1 \le i \le 2$, be the induced crisp injections. Then we define :*

$$FC_1 \widehat{+_{\sqcup_\oplus}} FC_2 \quad := \quad (\Phi_{\widehat{+_{\sqcup_\oplus}}}, \Pi_{\widehat{+_{\sqcup_\oplus}}},$$

$$\begin{aligned}
&\{(Q;\iota_1)^{\smile} \mid Q \in L_{in_1}\} \cup \{(Q';\iota_2^{\smile})^{\smile} \mid Q' \in L_{in_2}\},\\
&\{S;\kappa_1 \mid S \in L_{out_1}\} \cup \{S';\kappa_2 \mid S' \in L_{out_2}\},\\
&\{(;_{\odot}, (Q;\iota_1, S;\kappa_1)) \mid (;_{\odot}, (Q,S)) \in R_1\} \cup\\
&\{(;_{\odot}, (Q';\iota_2, S';\kappa_2)) \mid (;_{\odot}, (Q',S')) \in R_2\},\\
&\sqcup_{\oplus}, \Delta_{\widehat{+_{\sqcup_{\oplus}}}})
\end{aligned}$$

whereas $\Phi_{\widehat{+_{\sqcup_{\oplus}}}}$, $\Pi_{\widehat{+_{\sqcup_{\oplus}}}}$ *and* $\Delta_{\widehat{+_{\sqcup_{\oplus}}}}$ *are defined elementwise for a given input* $x : I \to A_1 + A_2$ *and output* $y : I \to B_1 + B_2$ *as follows :*

$$\begin{aligned}
\Phi_{\widehat{+_{\sqcup_{\oplus}}}}(x) &:= \Phi_1(x;\iota_1^{\smile});\iota_1 \sqcup_{\oplus} \Phi_2(x;\iota_2^{\smile});\iota_2\\
\Pi_{\widehat{+_{\sqcup_{\oplus}}}}(x)(C) &:= (\Pi_1(x;\iota_1^{\smile})(\iota_1;C;\kappa_1^{\smile}));\kappa_1 \sqcup_{\oplus} (\Pi_2(x;\iota_2^{\smile})(\iota_2;C;\kappa_2^{\smile}));\kappa_2\\
\Delta_{\widehat{+_{\sqcup_{\oplus}}}}(y) &:= \Delta_1(y;\kappa_1^{\smile});\kappa_1 \sqcup_{\oplus} \Delta_2(y;\kappa_2^{\smile});\kappa_2.
\end{aligned}$$

The $\widehat{+_{\sqcup_{\oplus}}}$ operator is defined pretty intuitively by connecting the linguistic entities with the respective injections and lifting the rulebase to the new defined entities. The resulting fuzzification, defuzzification and application functions are lifted analogously. In the following we denote the respective elements of the resulting controller by the subscript $_{\widehat{+_{\sqcup_{\oplus}}}}$.
Again, we want to examine whether this operation is an operation on the cores. We abbreviate

$$\begin{aligned}
R_1' &:= \{(;_{\odot}, (Q;\iota_1, S;\kappa_1)) \mid (;_{\odot}, (Q,S)) \in R_1\}\\
R_2' &:= \{(;_{\odot}, (Q';\iota_2, S';\kappa_2)) \mid (;_{\odot}, (Q',S')) \in R_2\}
\end{aligned}$$

and compute

$$\begin{aligned}
&C_{\widehat{+_{\sqcup_{\oplus}}}}\\
= &\biguplus_{\substack{(;_{\odot},(Q,R))\\ \in R_1' \cup R_2'}} (Q^{\smile};_{\odot} S) && \text{(definition } C_{\widehat{+_{\sqcup_{\oplus}}}}\text{)}\\
= &\biguplus_{\substack{(;_{\odot},(Q,S))\\ \in R_1'}} (Q^{\smile};_{\odot} S) \sqcup_{\oplus} \biguplus_{\substack{(;_{\odot},(Q',S'))\\ \in R_2'}} (Q'^{\smile};_{\odot} S')\\
= &\biguplus_{\substack{(;_{\odot},(Q,S))\\ \in R_1}} ((Q;\iota_1)^{\smile};_{\odot} (S;\kappa_1)) \sqcup_{\oplus} \biguplus_{\substack{(;_{\odot},(Q',S'))\\ \in R_2}} ((Q';\iota_2)^{\smile};_{\odot} (S';\kappa_2)) && \text{(Definition 4.2.5)}\\
= &\biguplus_{\substack{(;_{\odot},(Q,S))\\ \in R_1}} ((Q;_{\odot}\iota_1)^{\smile};_{\odot} (S;_{\odot}\kappa_1)) \sqcup_{\oplus} \biguplus_{\substack{(;_{\odot},(Q',S'))\\ \in R_2}} ((Q';_{\odot}\iota_2)^{\smile};_{\odot} (S';_{\odot}\kappa_2)) && \text{(Lemma 2.6.5)}\\
= &\biguplus_{\substack{(;_{\odot},(Q,S))\\ \in R_1}} (\iota_1^{\smile};_{\odot} (Q^{\smile};_{\odot} S);_{\odot}\kappa_1) \sqcup_{\oplus} \biguplus_{\substack{(;_{\odot},(Q',S'))\\ \in R_2}} (\iota_2^{\smile};_{\odot} (Q'^{\smile};_{\odot} S');_{\odot}\kappa_2) && (\odot \text{ closg-based})\\
= &\biguplus_{\substack{(;_{\odot},(Q,S))\\ \in R_1}} (\iota_1^{\smile};(Q^{\smile};_{\odot} S);\kappa_1) \sqcup_{\oplus} \biguplus_{\substack{(;_{\odot},(Q',S'))\\ \in R_2}} (\iota_2^{\smile};(Q'^{\smile};_{\odot} S');\kappa_2) && \text{(Lemma 2.6.5)}
\end{aligned}$$

$$= \iota_1^{\smile}; (\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} (Q^{\smile};_\odot S)); \kappa_1 \sqcup_\oplus \iota_2^{\smile}; (\biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} (Q'^{\smile};_\odot S')); \kappa_2 \qquad \text{(Definition 4.2.2(3))}$$
$$= (\iota_1^{\smile}; C_1; \kappa_1) \sqcup_\oplus (\iota_2^{\smile}; C_2; \kappa_2) \qquad \text{(definition)}$$
$$=: C_1 +_{\sqcup_\oplus} C_2.$$

Hence, we see that our prerequisites suffice to induce the operation $+_{\sqcup_\oplus}$ on the cores. In particular, we need prerequisites (1)-(3) of Definition 4.2.2 to make this computation possible.
To show the convenience of the $\widehat{+_{\sqcup_\oplus}}$ operator we want to study how the input-output behavior of the resulting controller is affected by the lifted fuzzification, defuzzification and application function. This is shown by the following theorem.

Theorem 4.2.1. *Let FC_1 and FC_2 be two combinable simple controllers such that $FC_1 \widehat{+_{\sqcup_\oplus}} FC_2$ is defined. Then we have*

$$(FC_1 \widehat{+_{\sqcup_\oplus}} FC_2)(x) = (FC_1(x; \iota_1^{\smile})); \kappa_1 \sqcup_\oplus (FC_2(x; \iota_2^{\smile})); \kappa_2$$

whereas ι_i and κ_i, $1 \leq i \leq 2$, are the induced crisp injections.

Proof. First, we have

$$\begin{aligned}
\Phi_{\widehat{+_{\sqcup_\oplus}}}(x); \iota_1^{\smile} &= (\Phi_1(x; \iota_1^{\smile}); \iota_1 \sqcup_\oplus \Phi_2(x; \iota_2^{\smile}); \iota_2); \iota_1^{\smile} && \text{(definition } \Phi_{\widehat{+_{\sqcup_\oplus}}}) \\
&= \Phi_1(x; \iota_1^{\smile}); \iota_1; \iota_1^{\smile} \sqcup_\oplus \Phi_2(x; \iota_2^{\smile}); \iota_2; \iota_1^{\smile} && \text{(Definition 4.2.2(3))} \\
&= \Phi_1(x; \iota_1^{\smile}) && \text{(Definition 4.2.2(6))}
\end{aligned}$$

and analogously $\Phi_{\widehat{+_{\sqcup_\oplus}}}(x); \iota_2^{\smile} = \Phi_2(x; \iota_2^{\smile})$.
Furhtermore, the following is true

$$\begin{aligned}
\iota_1; C_{\widehat{+_{\sqcup_\oplus}}}; \kappa_1^{\smile} &= \iota_1; ((\iota_1^{\smile}; C_1; \kappa_1) \sqcup_\oplus (\iota_2^{\smile}; C_2; \kappa_2)); \kappa_1^{\smile} && \text{(definition } C_{\widehat{+_{\sqcup_\oplus}}}) \\
&= (\iota_1; \iota_1^{\smile}; C_1; \kappa_1; \kappa_1^{\smile}) \sqcup_\oplus (\iota_1; \iota_2^{\smile}; C_2; \kappa_2; \kappa_1^{\smile}) && \text{(Definition 4.2.2(3))} \\
&= C_1
\end{aligned}$$

and analogously $\iota_2; C_{1\widehat{+_{\sqcup_\oplus}}2}; \kappa_2^{\smile} = C_2$, so that we can compute

$$\begin{aligned}
&(FC_1 \widehat{+_{\sqcup_\oplus}} FC_2)(x) \\
&= \Delta_{\widehat{+_{\sqcup_\oplus}}}(\Pi_{\widehat{+_{\sqcup_\oplus}}}(\Phi_{\widehat{+_{\sqcup_\oplus}}}(x))(C_{\widehat{+_{\sqcup_\oplus}}})) && \text{(definition)} \\
&= \Delta_{\widehat{+_{\sqcup_\oplus}}}((\Pi_1(\Phi_{\widehat{+_{\sqcup_\oplus}}}(x); \iota_1^{\smile})(\iota_1; C_{\widehat{+_{\sqcup_\oplus}}}; \kappa_1^{\smile})); \kappa_1 \sqcup_\oplus \\
&\qquad (\Pi_2(\Phi_{\widehat{+_{\sqcup_\oplus}}}(x); \iota_2^{\smile})(\iota_2; C_{\widehat{+_{\sqcup_\oplus}}}; \kappa_2^{\smile})); \kappa_2) && \text{(definition } \Pi_{\widehat{+_{\sqcup_\oplus}}})
\end{aligned}$$

$$\begin{aligned}
&= \Delta_{\widehat{+_{\sqcup_\oplus}}}((\Pi_1(\Phi_1(x;\iota_1^{\smile}))(C_1));\kappa_1 \sqcup_\oplus ((\Pi_2(\Phi_2(x;\iota_2^{\smile}))(C_2));\kappa_2) && \text{(above)}\\
&= \Delta_1((\Pi_1(\Phi_1(x;\iota_1^{\smile}))(C_1));\kappa_1;\kappa_1^{\smile});\kappa_1 \sqcup_\oplus \\
&\quad \Delta_2((\Pi_2(\Phi_2(x;\iota_2^{\smile}))(C_2));\kappa_2;\kappa_2^{\smile});\kappa_2 && \text{(definition } \Delta_{\widehat{+_{\sqcup_\oplus}}}\text{)}\\
&= \Delta_1(\Pi_1(\Phi_1(x;\iota_1^{\smile})(C_1)));\kappa_1 \sqcup_\oplus \Delta_2(\Pi_2(\Phi_2(x;\iota_2^{\smile})(C_2)));\kappa_2 \\
&= (FC_1(x;\iota_1^{\smile}));\kappa_1 \sqcup_\oplus (FC_2(x;\iota_2^{\smile}));\kappa_2. && \text{(definition)}
\end{aligned}$$

$\square$

This shows that $FC_1 \widehat{+_{\sqcup_\oplus}} FC_2$ infers the output in the intuitively expected way. If the input set $x : I \to A_1 + A_2$ is an element of $\mathcal{G}[I, A_1]$, the output is inferred by FC_1. Otherwise, it is inferred by FC_2.

Now, we want to introduce two modifications of the $\widehat{+_{\sqcup_\oplus}}$ operator.

Definition 4.2.6. *Let FC_1 and FC_2 be two combinable controllers. Then we define*

(1) $FC_1 \widehat{+_{\sqcup_\oplus}}^{\,t} FC_2 := (\ \Phi, \Pi, L_{in_1} \cup L_{in_2}, L_{out_{\widehat{+_{\sqcup_\oplus}}}},$
$\qquad \{(;_*, (Q, S;\kappa_1)) \mid (;_*, (Q, S)) \in R_1\} \cup$
$\qquad \{(;_*, (Q', S';\kappa_2)) \mid (;_*, (Q', S')) \in R_2\},$
$\qquad \sqcup_\oplus, \Delta_{\widehat{+_{\sqcup_\oplus}}}),$

iff $FC_1 : \mathcal{G}[I, A] \to \mathcal{G}[I, B_1]$, $FC_2 : \mathcal{G}[I, A] \to \mathcal{G}[I, B_2]$, $\Phi_1 = \Phi_2 =: \Phi$ *and* $\Pi_1 = \Pi_2 =: \Pi$,

(2) $FC_1 \widehat{+_{\sqcup_\oplus}}^{\,s} FC_2 := (\ \Phi_{\widehat{+_{\sqcup_\oplus}}}, \Pi_{\widehat{+_{\sqcup_\oplus}}}, L_{in_{\widehat{+_{\sqcup_\oplus}}}}, L_{out_1} \cup L_{out_2},$
$\qquad \{(;_*, (Q;\iota_1, S)) \mid (;_*, (Q, S)) \in R_1\} \cup$
$\qquad \{(;_*, (Q';\iota_2, S')) \mid (;_*, (Q', S')) \in R_2\},$
$\qquad \sqcup_\oplus, \Delta),$

iff $FC_1 : \mathcal{G}[I, A_1] \to \mathcal{G}[I, B]$, $FC_2 : \mathcal{G}[I, A_2] \to \mathcal{G}[I, B]$ *and* $\Delta_1 = \Delta_2 =: \Delta$,

whereas ι_i and κ_i, $1 \le i \le 2$, are the induced crisp injections.

Obviously, $\widehat{+_{\sqcup_\oplus}}^{\,t}$ only creates the direct sum of the targets of the cores of FC_1 and FC_2. This is indicated by the superscript ^{t}. To let this operation be well defined, both controllers must have the same source. In contrast, $\widehat{+_{\sqcup_\oplus}}^{\,s}$ computes the direct sum of the sources and lets the target of both cores unchanged. Hence, both controllers must have the same target. Notice that letting both source and target unchanged leads to the derived join operation (cf. Definition 4.2.4). With these remarks and Theorem 4.2.1 it is easy to see that

$$(FC_1 \widehat{+_{\sqcup_\oplus}}^{\,t} FC_2)(x) \quad = \quad (FC_1(x));\kappa_1 \sqcup_\oplus (FC_2(x));\kappa_2$$

$$\begin{aligned}(FC_1 \widehat{+_{\sqcup_\oplus}}^{s} FC_2)(x') &= \Delta(\Pi_1(\Phi_1(x;\iota_1^{\smile}))(C_1) \sqcup_\oplus \Pi_2(\Phi_2(x;\iota_2^{\smile}))(C_2)) \\ (FC_1 \widehat{\sqcup_\oplus} FC_2)(x) &= \Delta(\Pi(\Phi(x))(C_1) \sqcup_\oplus \Pi(\Phi(x))(C_2))\end{aligned}$$

holds for given inputs $x : A \to B_1 + B_2$ resp. $x' : A_1 + A_2 \to B$ and controllers FC_1 and FC_2 on which the operations are defined.

At the end we want to show a special behavior of a meet-based derived operation $\sqcup_\oplus$ together with crisp injections if $\oplus$ has neutral element $\perp\!\!\!\perp$. Since we prerequired this property of $\oplus$, the result is applicable for $\widehat{+_{\sqcup_\oplus}}$.

Lemma 4.2.3. *Let $\mathcal{G}$ be a Goguen category and $R_1 : A_1 \to B$, $R_2 : A_2 \to B$ be two relations. Furthermore, let $\iota_i : A_i \to A_1 + A_2$, $1 \leq i \leq 2$, be a pair of crisp injections. Then we have*

$$\iota_1^{\smile};R_1 \sqcup_\oplus \iota_2^{\smile};R_2 = \iota_1^{\smile};R_1 \sqcup \iota_2^{\smile};R_2$$

if $\perp\!\!\!\perp$ is the neutral element of $\oplus$.

Proof. First, let $\alpha \neq \perp\!\!\!\perp$ and $\beta \neq \perp\!\!\!\perp$ be two scalars on $A_1 + A_2$. Then we have

$$\begin{aligned}
&(\alpha \oplus \beta);((\alpha\backslash\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\beta\backslash\iota_2^{\smile};R_2)^{\downarrow}) && \\
&\sqsubseteq (\alpha \oplus \beta);((\iota_2;\alpha\backslash\iota_2;\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\iota_1;\beta\backslash\iota_1;\iota_2^{\smile};R_2)^{\downarrow}) && \text{(Lemma 2.5.4, }{}^{\downarrow}\text{ monotonic)} \\
&= (\alpha \oplus \beta);((\iota_2;\alpha\backslash\perp\!\!\!\perp_{A_2B})^{\downarrow} \sqcap (\iota_1;\beta\backslash\perp\!\!\!\perp_{A_1B})^{\downarrow}) && \text{(Definition 2.5.10)} \\
&= (\alpha \oplus \beta);\perp\!\!\!\perp_{(A_1+A_2)B} && (\alpha, \beta \neq \perp\!\!\!\perp) \\
&= \perp\!\!\!\perp_{(A_1+A_2)B}.
\end{aligned}$$

With this preparation, we now can compute

$$\begin{aligned}
\iota_1^{\smile};R_1 \sqcup_\oplus \iota_2^{\smile};R_2 &= \bigsqcup_{\alpha,\beta\in Sc[\mathcal{G}]} (\alpha \oplus \beta);((\alpha\backslash\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\beta\backslash\iota_2^{\smile};R_2)^{\downarrow}) && \text{(Definition 2.6.4)} \\
&= \bigsqcup_{\substack{\alpha,\beta\in Sc[\mathcal{G}] \\ \alpha=\perp\!\!\!\perp \text{ or } \beta=\perp\!\!\!\perp}} (\alpha \oplus \beta);((\alpha\backslash\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\beta\backslash\iota_2^{\smile};R_2)^{\downarrow}) && \text{(above)} \\
&= \bigsqcup_{\substack{\alpha,\beta\in Sc[\mathcal{G}] \\ \alpha=\perp\!\!\!\perp \text{ or } \beta=\perp\!\!\!\perp}} (\alpha \sqcup \beta);((\alpha\backslash\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\beta\backslash\iota_2^{\smile};R_2)^{\downarrow}) && (\perp\!\!\!\perp\text{ neutral element}) \\
&= \bigsqcup_{\alpha,\beta\in Sc[\mathcal{G}]} (\alpha \sqcup \beta);((\alpha\backslash\iota_1^{\smile};R_1)^{\downarrow} \sqcap (\beta\backslash\iota_2^{\smile};R_2)^{\downarrow}) && \text{(above)} \\
&= \iota_1^{\smile};R_1 \sqcup \iota_2^{\smile};R_2. && \text{(definition)}
\end{aligned}$$

$\square$

The result indicates that the function `fExtSrc` (cf. Section 3.4) is applicable. Since the similar result

$$S_1;\iota_1 \sqcup_{\oplus} S_2;\iota_2 = S_1;\iota_1 \sqcup S_2;\iota_2$$

follows by transposition for suitable relations S_1 and S_2, we also can use `fExtTrg`. Hence, we can efficiently compute the extended linguistic entities within our module for fuzzy controllers later on.

The derived meet operation

As with derived join, we again need FC_1 and FC_2 to have same source and target to apply the derived meet operation. Furthermore, we need $\Phi_1 = \Phi_2$, $\Pi_1 = \Pi_2$ and $\Delta_1 = \Delta_2$ so that we can omit the indices of these functions in the following. Now, we define $\widehat{\sqcap_{\odot}}$ on the level of simple controllers.

Definition 4.2.7. *Let FC_1 and FC_2 be two combinable fuzzy controllers. Then we define :*

$$\begin{aligned} FC_1 \widehat{\sqcap_{\odot}} FC_2 \quad := \quad & (\Phi, \Pi, \\ & \{Q \sqcap_{\odot} Q' \mid Q \in L_{in_1},\ Q' \in L_{in_2}\}, \\ & \{S \sqcap_{\odot} S' \mid S \in L_{out_1},\ S' \in L_{out_2}\}, \\ & \{(;_{\odot}, (Q \sqcap_{\odot} Q', S \sqcap_{\odot} S')) \mid (;_{\odot}, (Q,S)) \in R_1,\ (;_{\odot}, (Q',S')) \in R_2\} \\ & \sqcup_{\oplus}, \Delta) \end{aligned}$$

This operation seems not to be as intuitive as the other ones. We have to compute a completely new set of linguistic entites for both input and output and, hence, a new rulebase. Again, we identify the respective elements of the resulting controller by the additional subscript $\widehat{\sqcap_{\odot}}$.
In the following we want to show that Definition 4.2.7 also induces an operation on the cores of FC_1 and FC_2. We conclude

$$\begin{aligned} C_{\widehat{\sqcap_{\odot}}} &= \biguplus_{\substack{(;_{\odot},(Q,S)) \\ \in R_{1\widehat{\sqcap_{\odot}}2}}} (Q^{\smile};_{\odot} S) && \text{(definition)} \\ &= \biguplus_{\substack{(;_{\odot},(Q,S))\in R_1 \\ (;_{\odot},(Q',S'))\in R_2}} ((Q \sqcap_{\odot} Q')^{\smile};_{\odot} (S \sqcap_{\odot} S')) && \text{(definition } R_{\widehat{\sqcap_{\odot}}}) \\ &= \biguplus_{\substack{(;_{\odot},(Q,S))\in R_1 \\ (;_{\odot},(Q',S'))\in R_2}} ((Q^{\smile};_{\odot} S) \sqcap_{\odot} (Q'^{\smile};_{\odot} S')) && \text{(Definition 4.2.2(5))} \\ &= (\biguplus_{(;_{\odot},(Q,S))\in R_1} (Q^{\smile};_{\odot} S)) \sqcap_{\odot} (\biguplus_{(;_{\odot},(Q',S'))\in R_2} (Q'^{\smile};_{\odot} S')) && \text{(Definition 4.2.2(4))} \end{aligned}$$

$$= C_1 \sqcap_{\odot} C_2$$

which shows the assertion. As one can see, prerequisites (4) and (5) of Definition 4.2.2 are essential for the equality. We again want to mention that $\bigsqcup$ and $\sqcap$ provide these properties. With the operation above we are able to model extended rulebases. If we take the standard case $\sqcup_{\oplus} = \sqcup$ and $\sqcap_{\odot} = \sqcap$, we can represent rules of the form

if x is Q_{i_1} and ... and x is Q_{i_n} then y is S_{j_1} and ... and y is S_{j_m}

whereas we only could express rules of the form

if x is Q_{i_1} or ... or x is Q_{i_n} then y is S_{j_1} or ... or y is S_{j_m}

before.

The derived cross product

Although we have already extended the amount of expressable rules by the derived meet operation, we are up to now only able to model controllers with one linguistic variable for input and output, respectively. But, in practice one often wants to make the output variable(s) dependent from several input variables or vice versa. This, in general, corresponds to rules of the form

if x_1 is Q_{i_1} and ... and x_n is Q_{i_n} then y_1 is S_{j_1} and ... and y_m is S_{j_m}.

These rules can be modeled by relational products (cf. Definition 2.5.11), and more generally, by derived relational products.

Definition 4.2.8. *Let $FC_1 : \mathcal{G}[I, A_1] \to \mathcal{G}[I, B_1]$ and $FC_2 : \mathcal{G}[I, A_2] \to \mathcal{G}[I, B_2]$ be two combinable fuzzy controllers. Furthermore, let $\pi_i : A_1 \times A_2 \to A_i$ and $\rho_i : B_1 \times B_2 \to B_i$,$1 \leq i \leq 2$ be the induced crisp projections. Then we define :*

$$\begin{aligned}
FC_1 \widehat{\times_{\sqcap_\odot}} FC_2 \quad := \quad & (\Phi_{\widehat{\times_{\sqcap_\odot}}}, \Pi_{\widehat{\times_{\sqcap_\odot}}}, \\
& \{(Q; \pi_1^{\smile}) \sqcap_{\odot} (Q'; \pi_2^{\smile}) \mid Q \in L_{in_1},\ Q' \in L_{in_2}\}, \\
& \{(S; \rho_1^{\smile}) \sqcap_{\odot} (S'; \rho_2^{\smile}) \mid S \in L_{out_1},\ S' \in L_{out_2}\}, \\
& \{(;_{\odot}, ((Q; \pi_1^{\smile}) \sqcap_{\odot} (Q'; \pi_2^{\smile}), (S; \rho_1^{\smile}) \sqcap_{\odot} (S'; \rho_2^{\smile}))), \\
& \quad \mid (;_{\odot}, (Q, S)) \in R_1,\ (;_{\odot}, (Q', S')) \in R_2\}, \\
& \sqcup_{\oplus}, \Delta_{\widehat{\times_{\sqcap_\odot}}}).
\end{aligned}$$

whereas $\Phi_{\widehat{\times_{\sqcap_\odot}}}$, $\Pi_{\widehat{\times_{\sqcap_\odot}}}$ *and* $\Delta_{\widehat{\times_{\sqcap_\odot}}}$ *are defined elementwise for a given input* $x : I \to A_1 \times A_2$ *and output* $y : I \to B_1 \times B_2$ *as follows :*

$$\begin{aligned}
\Phi_{\widehat{\times_{\sqcap_\odot}}}(x) &:= \Phi_1(x;\pi_1);\pi_1^\smile \sqcap_\odot \Phi_2(x;\pi_2);\pi_2^\smile,\\
\Pi_{\widehat{\times_{\sqcap_\odot}}}(x)(C) &:= (\Pi_1(x;\pi_1)(\pi_1^\smile;C;\rho_1));\rho_1^\smile \sqcap_\odot (\Pi_2(x;\pi_2)(\pi_2^\smile;C;\rho_2));\rho_2^\smile,\\
\Delta_{\widehat{\times_{\sqcap_\odot}}}(y) &:= \Delta_1(y;\rho_1);\rho_1^\smile \sqcap_\odot \Delta_2(y;\rho_2);\rho_2^\smile.
\end{aligned}$$

We again have that Definition 4.2.8 induces an operation on the cores of FC_1 and FC_2. This is shown by the computation

$$\begin{aligned}
C_{\widehat{\times_{\sqcap_\odot}}} &= \biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_{1\widehat{\times_{\sqcap_\odot}}2}}} (Q^\smile;_\odot S) && \text{(definition)}\\
&= \biguplus_{\substack{(;_\odot,(Q,S))\in R_1\\ (;_\odot,(Q',S'))\in R_2}} (((Q;\pi_1^\smile)\sqcap_\odot(Q';\pi_2^\smile))^\smile;_\odot((S;\rho_1^\smile)\sqcap_\odot(S';\rho_2^\smile))) && (\text{def. } R_{1\widehat{\times_{\sqcap_\odot}}2})\\
&= \biguplus_{\substack{(;_\odot,(Q,S))\in R_1\\ (;_\odot,(Q',S'))\in R_2}} (((Q;\pi_1^\smile)^\smile;_\odot(S;\rho_1^\smile))\sqcap_\odot((Q';\pi_2^\smile)^\smile;_\odot(S';\rho_2^\smile))) && \text{(Def. 4.2.2(5))}\\
&= (\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} ((\pi_1;Q^\smile);_\odot(S;\rho_1^\smile)))\sqcap_\odot(\biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} ((\pi_2;Q'^\smile);_\odot(S';\rho_2^\smile))) && \text{(Def. 4.2.2(4))}\\
&= (\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} ((\pi_1;_\odot Q^\smile);_\odot(S;_\odot \rho_1^\smile)))\sqcap_\odot\\
&\quad\; (\biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} ((\pi_2;_\odot Q'^\smile);_\odot(S';_\odot \rho_2^\smile))) && \text{(Lem. 2.6.5)}\\
&= (\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} (\pi_1;_\odot(Q^\smile;_\odot S);_\odot \rho_1^\smile))\sqcap_\odot(\biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} (\pi_2;_\odot(Q'^\smile;_\odot S');_\odot \rho_2^\smile)) && (\odot\text{ ass.})\\
&= \pi_1;(\biguplus_{\substack{(;_\odot,(Q,S))\\ \in R_1}} (Q^\smile;_\odot S));\rho_1^\smile \sqcap_\odot \pi_2;(\biguplus_{\substack{(;_\odot,(Q',S'))\\ \in R_2}} (Q'^\smile;_\odot S'));\rho_2^\smile && \text{(Def. 4.2.2(3))}\\
&= (\pi_1;C_1;\rho_1^\smile)\sqcap_\odot(\pi_2;C_2;\rho_2^\smile) && \text{(definition)}\\
&=: C_1 \times_{\sqcap_\odot} C_2.
\end{aligned}$$

This implies that the $\widehat{\times_{\sqcap_\odot}}$ operation with the restrictions in Definition 4.2.2 and 4.2.3 is equivalent to having separate controllers for each pair of input/output variables.

With this result we are able to treat derived products of fuzzy controllers on the level of the cores. The corresponding model is shown in Figure 4.4. Controllers of this form are called *product controllers* in the following.

The next step is to examine the input-output behavior of product controllers. Unfortunately, we are not be able to show an analogous equality as in Theorem 4.2.1. This comes due to the fact that we cannot even guarantee $\sqcap_\odot$ subdistributivity for a $\sqcap$-based derivded operation.

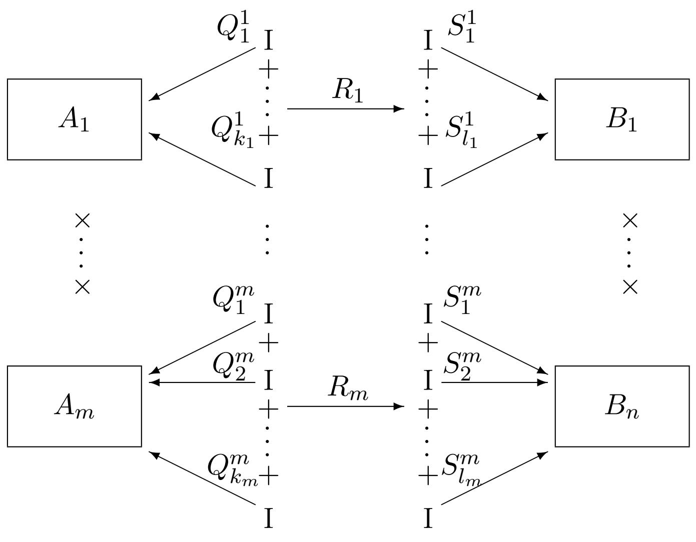

Figure 4.4: The core of the relational product of simple controllers

But, we want to provide a theorem which delivers a convenient result for the standard case of Mamdani inference. To do so, we first need the following lemma.

Lemma 4.2.4. *Let $\mathcal{G}$ be a Goguen category, A, B, C objects of $\mathcal{G}$ and $Q, Q' : A \to B$ and $R : B \to C$ relations such that R is crisp. Furthermore, let $\sqcap_\odot$ be a meet-based derived operation. Then we have*

$$(Q \sqcap_\odot Q'); R \sqsubseteq Q; R \sqcap_\odot Q'; R$$

Proof. We immediately compute

$$\begin{aligned}
(Q \sqcap_\odot Q'); R &= (\bigsqcup_{\alpha,\beta \in Sc[\mathcal{G}]} (\alpha \odot \beta); ((\alpha \backslash Q)^\downarrow \sqcap (\beta \backslash Q')^\downarrow)); R && \text{(Definition 2.6.4)} \\
&= \bigsqcup_{\alpha,\beta \in Sc[\mathcal{G}]} (\alpha \odot \beta); ((\alpha \backslash Q)^\downarrow \sqcap (\beta \backslash Q')^\downarrow); R && \text{(Lemma 2.5.5(3))} \\
&= \bigsqcup_{\alpha,\beta \in Sc[\mathcal{G}]} (\alpha \odot \beta); ((\alpha \backslash Q)^\downarrow \sqcap (\beta \backslash Q')^\downarrow); R^\downarrow && \text{(R crisp)} \\
&\sqsubseteq \bigsqcup_{\alpha,\beta \in Sc[\mathcal{G}]} (\alpha \odot \beta); ((\alpha \backslash Q)^\downarrow; R^\downarrow \sqcap (\beta \backslash Q')^\downarrow; R^\downarrow) && \text{($\sqcap$ subdistributivity)} \\
&\sqsubseteq \bigsqcup_{\alpha,\beta \in Sc[\mathcal{G}]} (\alpha \odot \beta); ((\alpha \backslash Q; R)^\downarrow \sqcap (\beta \backslash Q'; R)^\downarrow) && \text{(Lemma 2.5.4(4),$^\downarrow$ mon.)} \\
&= Q; R \sqcap_\odot Q'; R. && \text{(Definition 2.6.4)}
\end{aligned}$$

□

Hence, the following can be deducted from our prerequisites.

Theorem 4.2.2. *Let FC_1 and FC_2 be two combinable simple controllers such that $FC_1 \widehat{\times_{\sqcap_\odot}} FC_2$ is defined. Then we have*

$$\Pi_{\widehat{\times_{\sqcap_\odot}}}(\Phi_{\widehat{\times_{\sqcap_\odot}}}(x))(C_{\widehat{\times_{\sqcap_\odot}}}) \sqsubseteq (\Pi_1(\Phi_1(x;\pi_1))(C_1));\rho_1^{\smile} \sqcap_\odot (\Pi_2(\Phi_2(x;\pi_2))(C_2));\rho_2^{\smile}$$

whereas π_i and ρ_i, $1 \leq i \leq 2$, are the induced crisp projections.

Proof. First we have

$$\begin{array}{cll}
& \Pi_1(\Phi_{\widehat{\times_{\sqcap_\odot}}};\pi_1)(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & \\
= & \Pi_1((((\Phi_1(x;\pi_1));\pi_1^{\smile}) \sqcap_\odot ((\Phi_2(x;\pi_2));\pi_2^{\smile}));\pi_1)(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & (\text{def. } \Phi_{\widehat{\times_{\sqcap_\odot}}}) \\
\sqsubseteq & \Pi_1(((\Phi_1(x;\pi_1));\pi_1^{\smile};\pi_1) \sqcap_\odot ((\Phi_2(x;\pi_2));\pi_2^{\smile};\pi_1))(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & (\text{Lemma 4.2.4}) \\
= & \Pi_1(\Phi_1(x;\pi_1) \sqcap_\odot ((\Phi_2(x;\pi_2));\top\!\!\top_{A_2A_1}))(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & \\
\sqsubseteq & \Pi_1(\Phi_1(x;\pi_1) \sqcap_\odot \top\!\!\top_{IA_1})(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & \\
= & \Pi_1(\Phi_1(x;\pi_1)(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1) & (\text{Lemma 2.6.5}) \\
= & \Pi_1(\Phi_1(x;\pi_1))(\pi_1^{\smile}((\pi_1;C_1;\rho_1^{\smile}) \sqcap_\odot (\pi_2;C_2;\rho_2^{\smile}));\rho_1) & (\text{def. } C_{\widehat{\times_{\sqcap_\odot}}}) \\
\sqsubseteq & \Pi_1(\Phi_1(x;\pi_1))((\pi_1^{\smile};\pi_1;C_1;\rho_1^{\smile};\rho_1) \sqcap_\odot (\pi_1^{\smile};\pi_2;C_2;\rho_2^{\smile};\rho_1)) & (\text{Lemma 4.2.4}) \\
\sqsubseteq & \Pi_1(\Phi_1(x;\pi_1))(C_1 \sqcap_\odot (\top\!\!\top_{A_1A_2};C_2;\top\!\!\top_{B_1B_2})) & \\
\sqsubseteq & \Pi_1(\Phi_1(x;\pi_1))(C_1 \sqcap_\odot \top\!\!\top_{A_1B_2}) & \\
= & \Pi_1(\Phi_1(x;\pi_1))(C_1) & (\text{Lemma 2.6.5})
\end{array}$$

and analogously $\Pi_2(\Phi_{\widehat{\times_{\sqcap_\odot}}};\pi_2)(\pi_2^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_2) \sqsubseteq \Pi_2(\Phi_2(x;\pi_2))(C_2)$. Hence, we compute

$$\begin{array}{ll}
(\Pi_{\widehat{\times_{\sqcap_\odot}}}(\Phi_{\widehat{\times_{\sqcap_\odot}}}(x))(C_{\widehat{\times_{\sqcap_\odot}}}) & \\
= (\Pi_1(\Phi_{\widehat{\times_{\sqcap_\odot}}};\pi_1)(\pi_1^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_1));\rho_1^{\smile}\sqcap_\odot & \\
\quad (\Pi_2(\Phi_{\widehat{\times_{\sqcap_\odot}}};\pi_2)(\pi_2^{\smile};C_{\widehat{\times_{\sqcap_\odot}}};\rho_2));\rho_2^{\smile} & (\text{definition } \Pi_{\widehat{\times_{\sqcap_\odot}}}) \\
\sqsubseteq (\Pi_1(\Phi_1(x;\pi_1))(C_1));\rho_1^{\smile} \sqcap_\odot (\Pi_2(\Phi_2(x;\pi_2))(C_2));\rho_2^{\smile}. & (\text{above, } \sqcap_\odot \text{ mon.})
\end{array}$$

□

Notice that an analogous result including the defuzzification part $\Delta_{\widehat{\times_{\sqcap_\odot}}}$ can, in general, not be deducted since we did not prerequire the defuzzification function to be monotonic.

Considering the cases where only subequality holds in the last theorem motivates the following corollary.

Corollary 4.2.1. *Let FC_1 and FC_2 be two combinable simple controllers such that $FC_1 \widehat{\times_\sqcap} FC_2$ is defined. Furthermore, suppose both FC_1 and FC_2 use identity as fuzzification, ; as application and $\sqcup$ as aggregation function. If additionally $C_1^\downarrow \neq \perp\!\!\!\perp$ and $C_2^\downarrow \neq \perp\!\!\!\perp$ holds, we have*

$$(FC_1 \widehat{\times_\sqcap} FC_2)(x) = \Delta_{\widehat{\times_\sqcap}}(\Pi_1(\Phi_1(x;\pi_1))(C_1) \sqcap \Pi_2(\Phi_2(x;\pi_2))(C_2)).$$

Thus, Mamdani inference with two further restrictions on the fuzzification function and the cores is covered in quite a nice way by this operation. Before defuzzification we can apply the input separately to FC_1 and FC_2 without changing the behavior of $FC_1 \widehat{\times_\sqcap} FC_2$. Notice that the prerequisites $C_1^\downarrow \neq \perp\!\!\!\perp$ and $C_2^\downarrow \neq \perp\!\!\!\perp$ are essential to conclude

$$\begin{aligned} \top\!\!\!\top; C_1; \top\!\!\!\top &\sqsupseteq \top\!\!\!\top; C_1^\downarrow; \top\!\!\!\top \\ &= \top\!\!\!\top; C_1^{\downarrow\uparrow}; \top\!\!\!\top \\ &= \top\!\!\!\top \qquad\qquad \text{(Lemma 2.6.2)} \end{aligned}$$

and analogously $\top\!\!\!\top; C_2; \top\!\!\!\top = \top\!\!\!\top$. The restriction should not be too limitating since already one crisp entry within C_1 and C_2, respectively, suffices. This means that the rulebase of FC_1/FC_2 has to connect at least one element of the domain of FC_1/FC_2 to an element of the range of FC_1/FC_2 with degree 1.

Now, we again want to introduce two modifications of $\widehat{\times_{\sqcap_\odot}}$ (cf. Definition 4.2.6). Imagine the special situations that

(1) the cores of FC_1 and FC_2 have the same source or
(2) the cores of FC_1 and FC_2 have the same target.

Case (1) obviously makes it possible to "melt" the controllers in the way that different output variables are controlled by a single input variable. The second case represents the dual situation that separate input variables control only one output variable. These operations are provided with the next definition.

Definition 4.2.9. *Let FC_1 and FC_2 be two combinable controllers. Then we define*

$$\begin{aligned} (1)\ FC_1 \widehat{\times_{\sqcap_\odot}}^t FC_2 := (\ &\Phi, \Pi, \\ &\{Q \sqcap_\odot Q' \mid Q \in L_{in_1},\ Q' \in L_{in_2}\}, \\ &L_{out_{\widehat{\times_{\sqcap_\odot}}}}, \\ &\{(;_\odot, ((Q \sqcap_\odot Q'), (S;\rho_1^\smile) \sqcap_\odot (S';\rho_2^\smile))) \end{aligned}$$

$$\begin{aligned}
&\qquad\qquad\qquad\qquad \mid (;_\odot, (Q,S)) \in R_1,\ (;_\odot, (Q',S')) \in R_2\},\\
&\qquad\qquad\qquad \sqcup_\oplus, \Delta_{\widehat{\times_{\sqcap_\odot}}}\textit{)},\\
&\textit{iff } FC_1 : \mathcal{G}[I,A] \to \mathcal{G}[I,B_1],\ FC_2 : \mathcal{G}[I,A] \to \mathcal{G}[I,B_2],\ \Phi_1 = \Phi_2 =: \Phi\\
&\textit{and } \Pi_1 = \Pi_2 =: \Pi,
\end{aligned}$$

$$\begin{aligned}
\textit{(2) } FC_1 \widehat{\times_{\sqcap_\odot}}^{s} FC_2 := \textit{(}\ &\Phi_{\widehat{\times_{\sqcap_\odot}}}, \Pi_{\widehat{\times_{\sqcap_\odot}}},\\
&L_{in_{\widehat{\times_{\sqcap_\odot}}}},\\
&\{S \sqcap_\odot S' \mid S \in L_{out_1},\ S' \in L_{out_2}\},\\
&\{(;_\odot, ((Q;\pi_1^{\smile}) \sqcap_\odot (Q';\pi_2^{\smile}), S \sqcap_\odot S'))\\
&\qquad \mid (;_\odot, (Q,S)) \in R_1,\ (;_\odot, (Q',S')) \in R_2\},\\
&\sqcup_\oplus, \Delta\textit{)},
\end{aligned}$$

$$\textit{iff } FC_1 : \mathcal{G}[I,A_1] \to \mathcal{G}[I,B],\ FC_2 : \mathcal{G}[I,A_2] \to \mathcal{G}[I,B],\ \textit{and } \Delta_1 = \Delta_2 =: \Delta,$$

whereas π_i *and* ρ_i, $1 \le i \le 2$, *are the induced crisp projections.*

From these operations we see that melting both source and target of the cores of FC_1 and FC_2 results in the derived meet operation. Hence, we see from Theorem 4.2.2 that the input-output behavior of the modified operation $\widehat{\times_{\sqcap_\odot}}^{s}$ is given as follows

$$\Pi_{\widehat{\times_{\sqcap_\odot}}^{s}}(\Phi_{\widehat{\times_{\sqcap_\odot}}^{s}}(x))(C_{\widehat{\times_{\sqcap_\odot}}^{s}}) \ \sqsubseteq\ \Pi_1(\Phi_1(x;\pi_1))(C_1) \sqcap_\odot \Pi_2(\Phi_2(x;\pi_1))(C_2).$$

But, in contrast, we only can conclude

$$\begin{aligned}
\Pi_{\widehat{\times_{\sqcap_\odot}}^{t}}(\Phi_{\widehat{\times_{\sqcap_\odot}}^{t}}(x))(C_{\widehat{\times_{\sqcap_\odot}}^{t}}) &\ \sqsubseteq\ (\Pi(\Phi(x))(C_1));\rho_1^{\smile} \sqcap_\odot (\Pi(\Phi(x))(C_2));\rho_2^{\smile},\\
\Pi_{\widehat{\sqcap_\odot}}(\Phi_{\widehat{\sqcap_\odot}}(x))(C_{\widehat{\sqcap_\odot}}) &\ \sqsubseteq\ \Pi(\Phi(x))(C_1) \sqcap_\odot \Pi(\Phi(x))(C_2)
\end{aligned}$$

if the fuzzification part Φ delivers crisp values for all inputs x. Again, equality holds in the formulae above if we are dealing with Mamdani inference.

4.3 A module for fuzzy controllers

With the relational model and the operations on fuzzy controllers developed in the last section, we are now able to provide a module for handling controllers. With this we aim at suitable combinators to construct them and test their behavior in an algebraic (i.e., essentially component-free) manner. We then have the necessary background to develop a graphical user interface (GUI) to make the module

```
module FContr where


import LFuzzyRel
import LFuzzyRelCategories
import LFuzzyRelLattices
import Goguen
import Lattice
import List (elemIndex,nub,intersect,(\\))
```

comfortable to use.

The data structures

Essentially, we need two data structures — one for linguistic entities and one for fuzzy controllers. Since linguistic entities are single morphisms of the underlying Goguen category and it plays no role how they were created (by ordering-based weakening/strengthening, shifting etc.), the most simple variant to represent them is chosen.

```
data LingEntity mor  = LingEntity { entLabel :: String
                                   ,entRel   :: mor }
```

The only thing we have to add is a label for each entity. These labels are important to model the rulebase later on. Throughout this module we will use instances of type `LingEntity (FRel e obj obj)` of the data structure above.
Furthermore, the function

```
entTrg :: LingEntity (FRel e obj1 obj2) -> [obj2]
entTrg = trg . entRel
```

which obviously gives back the target object of the linguistic entity, is useful.
The `Eq` and `Show` instances are implemented as follows.

```
instance Eq (LingEntity mor) where
   x1 == x2 = entLabel x1 == entLabel x2


instance (Show mor) => Show (LingEntity mor) where
   show le = "ENTITY NAME     : "++entLabel le++"\n"++
             "ENTITY RELATION : "++"\n" ++ show (entRel le)
```

Notice that the user has to take care to avoid duplicate entity names since they are the only equality criteria of the `Eq` instance. We prefer this variant due to efficiency considerations.

From Definition 4.2.1 we know that a simple controller includes a fuzzification, application, aggregation and defuzzification function. Some of these parameters are restricted to be equal with two fuzzy controllers that shall be put together by one of the introduced operations. This implies that we will have to check them for equality at some point. Hence, they have to be labeled analogously to the linguistic entities.

```
data UnFunc mor  = UnFunc { unLabel :: String,
                            unFunc  :: mor -> mor }
data BinFunc mor = BinFunc { binLabel :: String,
                             binFunc  :: mor -> mor -> mor }
data DerOp e     = DerOp { opLabel :: String,
                           opLoos  :: Loos e,
                           comm    :: Bool,
                           idem    :: Bool }
```

Obviously, we have structures for unary functions `UnFunc` (fuzzification, defuzzification), binary functions `BinFunc` (application) and derived operations `DerOp` (aggregation). The structure for derived operations carries two additional flags to determine whether the loos is commutative (`comm`) and idempotent (`idem`), respectively. This helps to compute the rulebase more efficiently, later on. Notice that we also restricted the application function to be a composition-based derived operation. But, we have to generate a new application function, if we apply one of the operations $\widehat{+_{\sqcup_\oplus}}$ resp. $\widehat{\times_{\sqcap_\odot}}$ to given controllers. Hence, `DerOp` would not suffice to model this.
The induced `Show` and `Eq` instances are implemented as follows.

```
instance Eq (DerOp e) where
   op1 == op2 = opLabel op1 == opLabel op2
instance Eq (UnFunc mor) where
   f1 == f2 = unLabel f1 == unLabel f2
instance Eq (BinFunc mor) where
   f1 == f2 = binLabel f1 == binLabel f2

instance Show (DerOp e) where
   show = show . opLabel
instance Show (UnFunc mor) where
   show = show . unLabel
instance Show (BinFunc mor) where
   show = show . binLabel
```

Now, we are ready to focus on the controller data structure. Our target is a GUI for a comfortable creation of fuzzy controllers. Therefore, it is essential that the data structures make some kind of rollback possible. It has to carry the history of creation so that the user can step back to a previous state without losing information. This induces a recursive data structure. From Figures 4.3 and 4.4 we know that a fuzzy controller may form the sum or the product of several subcontrollers. Hence, we need a variant record covering three cases — a simple controller, a sum controller and a product controller.

```
type Rule           e   = (DerOp e,(String,[String]))

data Header   e obj mor = Header { contrLabel :: String      -- controller label
                                  ,unitObj    :: [obj]       -- unit object
                                  ,entryLat   :: Lat e       -- entry lattice
                                  ,fuzz       :: UnFunc mor  -- fuzzification
                                  ,appl       :: BinFunc mor -- application
                                  ,defuzz     :: UnFunc mor} -- defuzzification

data FuzContr e obj = SimpleContr { header     :: Header e obj (FRel e obj obj)
                                   ,aggr       :: DerOp e       -- aggregation
                                   ,rules      :: [Rule e]      -- rulebase
                                   ,lingEntIn  :: [LingEntity (FRel e obj obj)]
                                   ,lingEntOut :: [LingEntity (FRel e obj obj)]}
                    | SumContr    { header     :: Header e obj (FRel e obj obj)
                                   ,srcSum     :: Bool
                                   ,trgSum     :: Bool
                                   ,derJoin    :: DerOp e
                                   ,subContrs  :: [FuzContr e obj]}
                    | ProdContr { header     :: Header e obj (FRel e obj obj)
                                   ,srcProd    :: Bool
                                   ,trgProd    :: Bool
                                   ,derMeet    :: DerOp e
                                   ,subContrs  :: [FuzContr e obj]}
```

The data structures `Rule` (for a single rule) and `Header` are pretty intuitive. A rule consists of an interpretation function (`DerOp`), an input entity and a list of output entities it shall be connected with. A list of such rules then forms the rulebase of a controller. For example, the rulebase

if x is Q_1 then y is S_1
if x is Q_2 then y is S_3 and S_4

has to be represented by the list

```
[(<op1>,("Q1",["S1"])),(<op2>,("Q2",["S3","S4"]))]          .
```

Notice that the arrangement of the list entries plays no role if the aggregation function is commutative.

The header is, in wide areas, a realization of the needed functionality to infer the output. Thus, the header is needed by all three kinds of controller. As an extension to Definition 4.2.1, the header includes a label for the controller (`contrLabel`), the underlying entry lattice (`entryLat`) of the $\mathcal{L}$-fuzzy relations and a unit object (`unitObj`). The last two parameters are needed to automatically construct the underlying Goguen category, later on.

With these preparations, the `FuzContr` data structure is straightforward. The simple controller `SimpleContr`, in addition to the header, gets all needed functions to create the core. This includes an aggregation function `aggr`, a rulebase `rules` and the linguistic entities for the input (`lingEntIn`) and output (`lingEntOut`), respectively. Notice that `Header` is instantiated using `FRel e obj obj` for the type parameter. This implies that the modules `FContr` and `LFuzzyRel` can only be used together. A more general approach seems not to be senseful since too much specific functionality of `LFuzzyRel` is used.

The data structures `SumContr` and `ProdContr` are implemented equally. Besides the header, they get two flags which indicate, whether the direct sum resp. cross product shall be generated with the source resp. target of the controller. The meaning of these two flags is summarized in Table 4.3.

`sS`/`sP`	`tS`/`tP`	Operation `SumContr`	Operation `ProdContr`
`True`	`True`	$\widehat{+_{\sqcup_\oplus}}$	$\widehat{\times_{\sqcup_\odot}}$
`True`	`False`	$\widehat{+_{\sqcup_\oplus}}^{s}$	$\widehat{\times_{\sqcup_\odot}}^{s}$
`False`	`True`	$\widehat{+_{\sqcup_\oplus}}^{t}$	$\widehat{\times_{\sqcup_\odot}}^{t}$
`False`	`False`	$\sqcup_\oplus$	$\sqcup_\odot$

Table 4.1: The meaning of the flags `sS`/`tS` and `sP`/`tP`

Furthermore, both `SumContr` and `ProdContr` carry a $\sqcap$-based derived operation `derJoin` resp. `derMeet`. Notice, that the controllers in the subcontroller list `subctrs` have to be combinable (cf. Definition 4.2.2) and `derJoin` / `derMeet` have to fulfill Definition 4.2.3 to make all operations of the table above applicable.

Finally, we want to mention that the information the header carries could be extracted form the subcontroller list with `SumContr` and `ProdContr`. Hence, the header would not be necessary for these structures. But, we prefer this variant to have quick access to the parameters. Since no really high depth of the `FuzContr` data structure is to expect, the resulting overhead should be acceptable. Real applications will have to show whether this approach suffices or changes have to be made. Furthermore, the variants `SumContr` and `ProdContr` could have been put together to a single variant `ExtendedContr` or similar using an additional flag to determine which operation is wanted. But again, we prefer to have separate structures for convenience and clearness.

Again, we provide `Eq` and `Show` instances.

```
instance Eq (FuzContr e obj) where
   fc1 == fc2 = contrLabel (header fc1) == contrLabel (header fc2)

instance (Show obj, Show e) => Show (FuzContr e obj) where
   show (SimpleContr (Header l u e f ap d) ag rs lI lO) =
     let presentEnt =
           concat .
            map (\x -> "\n"++"    ENTITY NAME : "++ entLabel x++"\n"++
                              "    RELATION    : "++ "\n"++ show (entRel x))
     in
       "CONTROLLER NAME     : "++ l ++"\n"++
       "UNIT OBJECT         : "++ show u ++ "\n" ++
       "LING.ENTITUES INPUT : "++ presentEnt lI ++
       "LING.ENTITIES OUTPUT: "++ presentEnt lO ++ "\n"++
       "RULEBASE            : "++ (concat $ map (\x -> show x++"\n"++
                                                "                       ") rs )
   show (SumContr  (Header l u _ _ _ _) _ _ _ subctrs) =
     "CONTROLLER NAME     : "++ l ++"\n"++
     "UNIT OBJECT         : "++ show u ++"\n"++
     "SUBCONTROLLERS      : "++ concat (map (contrLabel.header) subctrs)
   show (ProdContr (Header l u e f ap d) dM sP tP subctrs) =
           show (SumContr(Header l u e f ap d) dM sP tP subctrs)
```

As with `LingEntity`, equality of two controllers is reduced to label equality. If the user explicitly wants to test whether the cores of two different controllers are equal, he has to use the function `core` introduced later on.

Computing the core of a controller

Having the necessary data structures, we now aim at suitable auxiliary functions to make the instantiation of fuzzy controllers and the computation of their cores comfortable for the user. With the next functions we want to support constructing the input resp. output linguistic entites. We introduce ordering and residual-based strengthening/weakening as well as shifting to build up linguistic entities that fit together.

```
greaterThan e m = fConv $ fLRes (fConv e) m
lessThan    e m = fConv $ fLRes e m

very xi m 1    = fLRes  m xi
very xi m i    = fLRes (very xi m $ i-1) xi

roughly xi m 1 = fComp m xi
roughly xi m i = fComp (roughly xi m $ i-1) xi

shiftL m bij 1 = fComp m $ fConv bij
shiftL m bij i = fComp (shiftL m bij $ i-1) $ fConv bij

shiftR m bij 1 = fComp m bij
shiftR m bij i = fComp (shiftR m bij $ i-1) bij
```

These routines are direct realizations of the underlying definitions. Notice that we deliberately do not yet use the `Goguen` module to compute the terms. Throughout this module we differentiate between two different levels of abstraction. Everything that has to be done to construct a controller is done on a level that is not pure algebraic. For these actions the modules `LFuzzyRel` and `Lattice` are used. But, all operations on controllers themselves are done using the underlying Goguen category, which is automatically generated. This has the further advantage that the user does not have to instantiate it himself.

The following functions help to compute the underlying Goguen category of a fuzzy controller.

```
contrDom, contrRan :: FuzContr e obj -> [obj]
contrDom (SimpleContr _ _ _ lI _)  = if null lI then [] else entTrg $ head lI
contrDom (SumContr    _ sS _ _ subctrs) =
    if sS then concat $ map contrDom subctrs
    else contrDom (head subctrs)
contrDom (ProdContr   _ sP _ _ subctrs) =
    if sP then crossProd $ map contrDom subctrs
    else contrDom (head subctrs)

contrRan (SimpleContr _ _ _ _ lO)  = if null lO then [] else entTrg $ head lO
contrRan (SumContr    _ _ tS _ subctrs) =
    if tS then concat $ map contrRan subctrs
    else contrRan (head subctrs)
contrRan (ProdContr   _ _ tP _ subctrs) =
    if tP then crossProd $ map contrRan subctrs
    else contrRan (head subctrs)

catObjects :: (Eq obj) => FuzContr e obj -> [[obj]]
catObjects fc@(SimpleContr h _ _ _ _) =
    nub $ [unitObj h,contrDom fc,contrRan fc]
catObjects fc =
    nub $ contrDom fc : contrRan fc : map (concat . catObjects) (subContrs fc)

contrCat :: (Eq e, Eq obj) => FuzContr e obj ->
                              Gog (Loos e) [obj] (FRel e obj obj)
contrCat fc = let eLat = entryLat $ header fc in
                fRelGogCat eLat (getAllFRels eLat $ catObjects fc)
```

Obviously, `contrDom` resp. `contrRan` deliver the domain and range of the controller, respectively. To do so in the `SimpleContr` case, they build up the direct sums by simple list concatenation. In the `ProdContr` case, the function `crossProd` is used to compute the relational products of the domains resp. ranges of all subcontrollers. The exact manner of function of this routine is explained below.
The necessary objects to instantiate the underlying Goguen category are delivered by `catObjects`. The simple case `SimpleContr` provides the unit object, the domain and range of the controller and the targets of all input and output entities, respectively. With controllers of type `ProdContr` we again have to add the products of the input resp. output spaces of the subcontrollers.

To understand the computation of the core of a product controller later on, it is very important to know how the function

```
crossProd :: [[obj]] -> [obj]
crossProd xs = concat $ replicate (product $ map length (init xs)) (last xs)
```

works. Obviously, `crossProd` is parametrized by the list `xs` of the objects whichs cross product shall be generated. But again, Haskell's strong typing turns into a demerit. It would be nice if we could represent the resulting cross product by a list of tuples. Thus, for instance, the product $\{1,2\} \times \{3,4\}$ could be represented by the list

```
[(1,2),(1,4),(2,3),(2,4)].
```

But, as shown above, we aim at an instance of a Goguen category (`Gog`) where all objects have to be of the same type. Thus, the lists `[1,2]`, `[3,4]` and `[(1,2),(1,4),(2,3),(2,4)]` cannot be objects of the same Goguen category. The escape we use is to simply replicate the elements of the last object of `xs` n times whereas n is the product of the lengths of all objects out of `xs` except the last one. The resulting list then is isomorphic to the cross product. In our example we get `[3,4,3,4]`.
The interpretability suffers from this construction. Hence, we provide the function

```
getLabel :: Int -> [[obj]] -> [Int]
getLabel i srcs = getLabel' i $ map length srcs
  where
    getLabel' i [l]    = [i]
    getLabel' i (l:ls) = div i (product ls) : getLabel' (mod i $ product ls) ls
```

which delivers the interpretation of a position in the product of several lists. To do so it has to get the position in question (`i`) and the sources `srcs` from which the cross product was created. The result then is a list of positions — one for each list out of `srcs`. In our example, typing `getLabel 2 [[1,2],[3,4]]` results in the output `[1,0]`. Notice that counting the position starts with zero. Notice furthermore, that `getLabel` presupposes that the cross product was created *right associative* (as shown with `crossProd`) whereas the arrangement of the elements of `srcs` is decisive. This convention comes due to the fact that it allows a relatively comfortable computation of the (crisp) projections, which is done by the following function.

```
projections :: Gog (Loos e) [obj] (FRel e obj obj) ->
               Lat e -> [[obj]] -> [obj] -> [FRel e obj obj]
projections g e srcs trg = projs' 1 srcs
  where
    projs' _ []     = []
    projs' n (s:ss) =
       let l1 = length s - 1
           l2 = product (map length (ss)) - 1
       in fUpd (gog_bottom g trg s) [( (k*(l1+1)*(l2+1)+j*(l2+1)+i,j)
                                     ,lat_topEl e) | k<-[0..n-1],
                                                     i<-[0..l2],
                                                     j<-[0..l1]]
          : projs' (n*(l1+1)) ss
```

Consider, for example, the three sets $\{1, 2, 3\}$, $\{4, 5\}$ and $\{6, 7\}$. The relational product of $\{1, 2, 3\} \times (\{5, 6\} \times \{8, 9\})$ with the convention above then is computed by the projections shown in Figure 4.5.

src/trg	**1**	**2**	**3**	**5**	**6**	**8**	**9**
(1,5,8)	1	0	0	1	0	1	0
(1,5,9)	1	0	0	1	0	0	1
(1,6,8)	1	0	0	0	1	1	0
(1,6,9)	1	0	0	0	1	0	1
(2,5,8)	0	1	0	1	0	1	0
(2,5,9)	0	1	0	1	0	0	1
(2,6,8)	0	1	0	0	1	1	0
(2,6,9)	0	1	0	0	1	0	1
(3,5,8)	0	0	1	1	0	1	0
(3,5,9)	0	0	1	1	0	0	1
(3,6,8)	0	0	1	0	1	1	0
(3,6,9)	0	0	1	0	1	0	1
proj.		ρ_1		ρ_2		ρ_3	

Figure 4.5: The projections for $\{1,2,3\} \times (\{5,6\} \times \{8,9\})$

With this, 1 and 0 are the greatest resp. least element of the underlying entry lattice of the $\mathcal{L}$-fuzzy relations. One can see the periodicity of the 1-entries which makes the computation quite comfortable. In ρ_1 we have period 4, i.e., we have four 1-entries below each other in every column. This comes due to the fact that $4 = \mid \{5,6\} \mid \cdot \mid \{8,9\} \mid$. Analogously, we have period $2 = \mid \{8,9\} \mid$ with ρ_2 and so on. With these remarks it should be clear how `projections` works. It takes the underlying Goguen category `g`, the underlying entry lattice `e` to have access to its least resp. greatest element, and the objects `srcs` from which the cross product `trg` was created. The computation of the projections then is straightforward. Notice that all terms are computed on the abstraction level of Goguen categories since `projections` shall support the determination of the core of a controller.

The function

```
injections :: Gog (Loos e) [obj] (FRel e obj obj) ->
              Lat e -> [[obj]] -> [obj] -> [FRel e obj obj]
injections g e srcs trg = inj' srcs 0
  where
    inj'[]     _ = []
    inj'(s:ss) n = fUpd (gog_bottom g s trg)
                        [((i,j),lat_topEl e) | i <- [0..length s-1],
                                               j <- [i..length trg-1], i+n==j]
                   : inj' ss (n+length s)
```

delivers the counterpart of `projections` by computing the (crisp) injections for a direct sum `trg` that was created out of the objects `srcs`. Again, the computation is straightforward whereas the arrangement of `srcs` is decisive.

Now, we are ready to compute the core of a controller. To do so, we make use of the underlying equations (cf. Section 4.2.1).

```
core :: (Eq obj, Eq e) => FuzContr e obj -> FRel e obj obj
core fc@(SimpleContr (Header _ _ _ _ _ _) agg rs lEIn lEOut) =
  let g = contrCat fc
  in foldl (gog_derOp g "Meet" (opLoos agg))
           (gog_bottom g (contrDom fc) (contrRan fc)) $
             map (\(intFun,(x,y)) -> gog_derOp g "Comp" (opLoos intFun)
                                     (gog_converse g x) y)
                 (sortEnt lEIn lEOut rs)
  where
    getMor les s = let ents = filter ((== s).entLabel) les
                   in if null ents then error "No such entity !"
                      else head ents
    sortEnt _    _    []            = []
    sortEnt lesI lesO ((iF,(s,[])):ss)   = sortEnt lesI lesO ss
    sortEnt lesI lesO ((iF,(s,t:ts)):ss) =
           (iF, (entRel $ getMor lesI s,
                 entRel $ getMor lesO t)) : sortEnt lesI lesO ((iF,(s,ts)):ss)
core fc@(SumContr _ sS tS derJoin subctrs) =
  let
    g        = contrCat fc;
    doms     = map contrDom subctrs;  dom = concat doms
    rans     = map contrRan subctrs;  ran = concat rans
    subCores = case (sS,tS) of
                 (True,True)   -> injR [] rans . injL [] doms
                 (True,False)  -> injL [] doms
                 (False,True)  -> injR [] rans
                 (False,False) -> id
  in
    foldl1 (gog_derOp g "Meet" (opLoos derJoin)) $ subCores $ map core subctrs
  where
    injL us [l]    (x:xs) = [fExtSrc us x []]
    injL us (l:ls) (x:xs) = fExtSrc us x ls : injL (us++[l]) ls xs
    injR us [l]    (x:xs) = [fExtTrg us x []]
    injR us (l:ls) (x:xs) = fExtTrg us x ls : injR (us++[l]) ls xs

core fc@(ProdContr h sP tP derMeet subctrs) =
  let
    g        = contrCat fc
    compG    = gog_comp g
    doms     = map contrDom subctrs;  dom = crossProd doms
    rans     = map contrRan subctrs;  ran = crossProd rans
    projL xs = zipWith compG (projections g (entryLat h) doms dom)xs
    projR xs = zipWith (\r p -> compG r $ gog_converse g p) xs $
                       projections g (entryLat h) rans ran
    subCores = case (sP,tP) of
                 (True,True)   -> projR . projL
                 (True,False)  -> projL
                 (False,True)  -> projR
                 (False,False) -> id
  in foldl1 (gog_derOp g "Meet" (opLoos derMeet)) $ subCores $ map core subctrs
```

We again have to differentiate between the three cases `SimpleContr`, `SumContr` and `ProdContr`. The computation for simple controllers is straightforward by reading out the rulebase and interpreting the corresponding rules. With sum and product controllers we have to treat the four different cases arising from the setting of the flags `sS` / `tS` resp. `sP` / `tP`. According to their state, the cores of the subcontrollers have to be extended by the (automatically generated) injections and projections, respectively. We want to mention a difference between the computations for sum controllers and product controllers. From Lemma 4.2.3 we know that a term $\iota_1^{\smile} R_1 \sqcup_{\oplus} \iota_2^{\smile} R_2$ is equivalent to $\iota_1^{\smile} R_1 \sqcup \iota_2^{\smile} R_2$ for relations $R_1 : A_1 \to B$, $R_2 : A_2 \to B$ and the induced crisp injections $\iota_i : A_i \to A_1 + A_2$ if $\sqcup_{\oplus}$ is a meet-based derived operation with neutral element $⊥\!\!⊥$. Thus, applying the injections in this term corresponds to simply extending the matrices R_1 resp. R_2 by zero rows such that the resulting matrices R_1' and R_2' have source $A_1 + A_2$ and target B. This means that the operations `fExtSrc` and `fExtTrg` of the `LFuzzyRel` module are applicable. An analogous computation for the relational product is obviously not to expect.

Combinators for fuzzy controllers

After the basic routines on fuzzy controllers, we now want to provide suitable combinators to construct them step by step. Furthermore, we then have the possibility to construct new fuzzy controllers in a component-free manner. Building up a fuzzy controller always starts with the empty controller.

```
emptyContr l unit eLat = let emptyLoos = Loos { loos_lat = eLat
                                               ,loos_op  = const $ id
                                               ,loos_e   = lat_botEl eLat
                                               ,loos_z   = lat_botEl eLat}
                             emptyDerOp = DerOp "" emptyLoos True True
                             emptyFunc  = UnFunc "" id
                             emptyBFunc = BinFunc "" (const $ id)
                         in if l=="" then error ("Empty label!")
                            else SimpleContr (Header l unit eLat  emptyFunc
                                                     emptyBFunc emptyFunc)
                                       emptyDerOp [] [] []
```

Obviously, `emptyContr` only takes a minimal set of parameters. The remaining part can be set to default values. Notice that a controller of type `SumContr` or `ProdContr` with an empty `subContrs` list is seen to be invalid. Furthermore, the empty label is not allowed for a controller. The reason for this becomes clear later on.

An empty controller can be updated by new rules and linguistic entities, respectively. Furthermore, the corresponding parts of the header can be set. This is covered by the following routines.

```
hUpdFuzz f   (Header l u e _ ap d) = Header l u e f ap d
hUpdAppl ap  (Header l u e f _ d)  = Header l u e f ap d
hUpdDefuzz d (Header l u e f ap _) = Header l u e f ap d

contrUpdFuzz f (SimpleContr h ag rs lEIn lEOut) =
   SimpleContr (hUpdFuzz f h) ag rs lEIn lEOut
contrUpdFuzz f (SumContr h sS tS dJ sC) =
   SumContr (hUpdFuzz f h) sS tS dJ sC
contrUpdFuzz f (ProdContr h sP tP dM sC) =
   ProdContr (hUpdFuzz f h) sP tP dM sC

contrUpdAppl ap (SimpleContr h ag rs lEIn lEOut) =
   SimpleContr (hUpdAppl ap h) ag rs lEIn lEOut
contrUpdAppl ap (SumContr h sS tS dJ sC) =
   SumContr (hUpdAppl ap h) sS tS dJ sC
contrUpdAppl ap (ProdContr h sP tP dM sC) =
   ProdContr (hUpdAppl ap h) sP tP dM sC

contrUpdAggr ag (SimpleContr h _ rs lEIn lEOut) =
    SimpleContr h ag rs lEIn lEOut

contrUpdDefuzz d (SimpleContr h ag rs lEIn lEOut) =
   SimpleContr (hUpdDefuzz d h) ag rs lEIn lEOut
contrUpdDefuzz d (SumContr h sS tS dJ sC) =
   SumContr (hUpdDefuzz d h) sS tS dJ sC
contrUpdDefuzz d (ProdContr h sP tP dM sC) =
   ProdContr (hUpdDefuzz d h) sP tP dM sC

contrUpdRuleIn, contrUpdRuleOut ::
    String -> [Rule e] -> FuzContr e obj -> FuzContr e obj
contrUpdRuleIn cLabel xs fc@(SimpleContr (Header l u e f ap d) ag rs lI lO) =
    if cLabel == "" || l == cLabel then
      SimpleContr (Header l u e f ap d) ag (xs++rs) lI lO
    else fc
contrUpdRuleIn cLabel xs (SumContr  (Header l u e f ap d) dJ sS tS subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a sum controller"++
             " or is invalid !")
    else SumContr (Header l u e f ap d) dJ sS tS $
                  map (contrUpdRuleIn cLabel xs) subctrs
contrUpdRuleIn cLabel xs (ProdContr (Header l u e f ap d) dM sP tP subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a product controller"++
             " or is invalid !")
    else ProdContr (Header l u e f ap d) dM sP tP $
         map (contrUpdRuleIn cLabel xs) subctrs

contrUpdRuleOut l xs =
    contrUpdRuleIn l (concat $ map (\(iF,(t,ss)) -> [(iF,(s,[t])) | s<-ss]) xs)

contrUpdEntIn, contrUpdEntOut :: (Eq obj) =>
      String -> [(DerOp e,
                 (LingEntity (FRel e obj obj),[String]))] ->
      FuzContr e obj -> FuzContr e obj
contrUpdEntIn cLabel lEs fc@(SimpleContr (Header l u e f ap d) ag rs lI lO) =
    if cLabel == "" || l == cLabel then
```

```
      SimpleContr (Header l u e f ap d) ag
                  (map (\(iF,(e,ts)) -> (iF, (entLabel e,ts))) lEs ++ rs)
                  (map (fst.snd) lEs ++ lI)
                  lO
    else fc
contrUpdEntIn cLabel lEs (SumContr (Header l u e f ap d) dJ sS tS subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a sum controller"++
             " or is invalid !")
    else SumContr (Header l u e f ap d) dJ sS tS $
              map (contrUpdEntIn cLabel lEs) subctrs
contrUpdEntIn cLabel lEs (ProdContr (Header l u e f ap d) dM sP tP subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a product controller"++
             " or is invalid !")
    else ProdContr (Header l u e f ap d) dM sP tP $
              map (contrUpdEntIn cLabel lEs) subctrs

contrUpdEntOut cLabel lEs fc@ (SimpleContr (Header l u e f ap d) ag rs lI lO) =
    if cLabel == "" || l == cLabel then
      SimpleContr (Header l u e f ap d) ag
                  (concat (map (\(iF,(e,ts)) -> [(iF,(t,[entLabel e]))
                                                | t<-ts]) lEs) ++ rs)
                  lI
                  (map (fst.snd) lEs ++ lO)
    else fc
contrUpdEntOut cLabel lEs (SumContr (Header l u e f ap d) dJ sS tS subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a sum controller"++
             " or is invalid !")
    else SumContr (Header l u e f ap d) dJ sS tS $
              map (contrUpdEntOut cLabel lEs) subctrs
contrUpdEntOut cLabel lEs (ProdContr (Header l u e f ap d) dM sP tP subctrs) =
    if cLabel == "" || l == cLabel then
      error ("Label \""++cLabel++"\" specifies a product controller"++
             " or is invalid !")
    else ProdContr (Header l u e f ap d) dM sP tP $
              map (contrUpdEntOut cLabel lEs) subctrs
```

The functions for the header should be clear. For updating the rulebase, we have `contrUpdRuleIn` resp. `contrUpdRuleOut`. For the case that the rulebase of a subcontroller within a sum or product controller is to update, we introduce the parameter `cLabel`. As shown here, the `SumContr` and `ProdContr` parts of every function shown above are recursive. The operation in question is then applied to *every* subcontroller having the name `cLabel`. If `cLabel` specifies a sum or product controller, the functions above fail with an error message. Notice again, that we demand the controller labels to be unique.
For simple controllers `cLabel` plays no role and, hence, the user can set it to `""`. Here we see the reason why controllers with an empty label are not allowed.
Back to the functions for updating rules, the parameter `xs` specifies the new rules carrying the list of new target entities a source entity shall be connected with. Notice that we have a difference here between updating rules for input entities and updating them for output entities. With `contrUpdRuleIn` the source entity is an input entity and the targets are output entities. This exactly corresponds to our interpretation of the data type `Rule`. But, with `contrUpdRuleOut` the source is an output entity. Hence, we have to modify the list in the way shown above such that we can use `contrUpdRuleIn` to compute the resulting controller.
The functions for updating linguistic entities work analogously. The only interesting thing is the structure of the new parameter `lEs`. It consists of a list of tuples each carrying the new linguistic entity and a list of entities it shall be connected with in the rulebase. Hence, we are able to update the rulebase simultaneously without having to perform a separate function call.

All these operations above are still somehow componentwise. The next constructions are based on ready created fuzzy controllers. The product of two controllers can be build by the function

```
contrProd, contrSum :: (Eq e, Eq obj) => Bool -> Bool -> FuzContr e obj ->
                                        FuzContr e obj ->  DerOp e -> FuzContr e obj
contrProd sP tP c1 c2 derMeet  =
    let dummy         = ProdContr (header c1) sP tP derMeet [c1,c2]
        eLat          = entryLat $ header c1;  g     = contrCat dummy
        compG         = gog_comp g;            convG = gog_converse g
        dom1          = contrDom c1;           dom2  = contrDom c2
        ran1          = contrRan c1;           ran2  = contrRan c2
        pI            = projections g eLat [dom1,dom2] $ crossProd [dom1,dom2]
        pO            = projections g eLat [ran1,ran2] $ crossProd [ran1,ran2]
        (newF,newApp) =
          let f1  = fuzz (header c1); f2  = fuzz (header c2)
          ap1 = appl (header c1); ap2 = appl (header c1)
          a1  = binFunc ap1;      a2  = binFunc ap2
      in
       if not sP then (f1,ap1)
       else
        (UnFunc (unLabel f1++"**"++unLabel f2)
          (\x -> foldl1 (gog_derOp g "Meet" (opLoos derMeet)) $
                 zipWith (\p f -> compG (f $ compG x p) $ convG p)
                 pI [unFunc f1,unFunc f2]),
         BinFunc (binLabel ap1++"**"++binLabel ap2)
          (\x y -> gog_derOp g "Meet" (opLoos derMeet)
                   (compG (a1 (compG x (head pI)) $
                               (compG (compG (convG $ head pI) y) $ head pO))
                          $ convG (head pO) )
                   (compG (a2 (compG x (last pI)) $
                               (compG (compG (convG $ last pI) y) $ last pO))
                          $ convG (last pO) ) ) )
    newDF         =
      let df1 = defuzz (header c1); df2 = defuzz (header c2) in
        if not tP then df1
        else UnFunc (unLabel df1++"**"++unLabel df2)
                   (\x -> foldl1 (gog_derOp g "Meet" (opLoos derMeet)) $
                          zipWith (\p f -> compG (f $ compG x p) $ convG p)
                          pO [unFunc df1,unFunc df2])
in  ProdContr (Header
                 (contrLabel (header c1)++"**"++contrLabel (header c2))
                 (unitObj $ header c1) (entryLat $ header c1)
                 newF newApp newDF )
              sP tP
              derMeet
              [c1,c2]
```

which is straightforward due to the explanations in Section 4.2.1. The only things that have to be computed are the fuzzification, application and defuzzification functions in conformance with the flags `sP` and `tP`. This is done by `newF`, `newAppl` and `newDF`, respectively. The label of the resulting controller is simply created by connecting the labels of `c1` and `c2` with an additional `**`. Notice that we do not have to differentiate between the three variants of `FuzContr` since we only need parameters of the header of `c1` and `c2`. But, notice that the function is *not* associative in the sense that the resulting data structures are equal. But, for the mathematical interpretation (the core) this plays no role since the relational product is associative. Furthermore, we use the labels of two controllers as the equality criteria in our `Eq` instance. As one can see, the computation of the label of the resulting controller in `contrProd` is associative. Notice that the operation above preserves only the properties both `c1` and `c2` have, i.e., a rule of the product controller can only fire if this rule would cause `c1` and `c2` to fire. This is immediately seen by the fact that the loos `derMeet` is restricted to have $\mathbb{I}$ as neutral element and, hence, $\perp\!\!\!\perp$ as zero.

The next construction is the relational sum of two controllers.

```
contrSum sS tS c1 c2 derJoin  =
    let dummy          = SumContr (header c1) sS tS derJoin [c1,c2]
        eLat           = entryLat $ header c1;  g     = contrCat dummy
        compG          = gog_comp g;            convG = gog_converse g
        dom1           = contrDom c1;           dom2  = contrDom c2
        ran1           = contrRan c1;           ran2  = contrRan c2
        iI             = injections g eLat [dom1,dom2] $ dom1++dom2
        iO             = injections g eLat [ran1,ran2] $ ran1++ran2
        (newF,newApp) =
          let f1  = fuzz (header c1); f2  = fuzz (header c2)
              ap1 = appl (header c1); ap2 = appl (header c1)
              a1  = binFunc ap1;      a2  = binFunc ap2
          in
           if not sS then (f1,ap1)
           else
            (UnFunc (unLabel f1++"++"++unLabel f2)
               (\x -> foldl1 (gog_derOp g "Meet" (opLoos derJoin)) $
                      zipWith (\i f -> compG (f $ compG x (convG i)) i)
                      iI [unFunc f1,unFunc f2]),
             BinFunc (binLabel ap1++"++"++binLabel ap2)
               (\x y -> gog_derOp g "Meet" (opLoos derJoin)
                        (compG (a1 (compG x $ convG (head iI)) $
                                   (compG (compG (head iI) y) $ convG (head iO)))
                               $ head iO )
                        (compG (a2 (compG x $ convG (last iI)) $
                                   (compG (compG (last iI) y) $ convG (last iO)))
                               $ last iO ) ) )
        newDF          =
          let df1 = defuzz (header c1); df2 = defuzz (header c2) in
          if not tS then df1
          else  UnFunc (unLabel df1++"++"++unLabel df2)
                       (\x -> foldl1 (gog_derOp g "Meet" (opLoos derJoin)) $
                              zipWith (\i f -> compG (f $ compG x (convG i)) i)
                              iO [unFunc df1,unFunc df2])
    in  SumContr (Header
                     (contrLabel (header c1)++"++"++contrLabel (header c2))
                     (unitObj $ header c1) (entryLat $ header c1)
                     newF newApp newDF )
                   sS tS
                   derJoin
                   [c1,c2]
```

The implementation is analogous and a direct realization of the underlying operation. The following two functions mark the derived meet resp. derived join operation on two fuzzy controllers as special cases for the functions above.

```
contrJoin, contrMeet :: (Eq e, Eq obj) => FuzContr e obj -> FuzContr e obj ->
                                          DerOp e -> FuzContr e obj
contrJoin = contrSum  False False
contrMeet = contrProd False False
```

Now, we want to switch to `contrMelt`. The main motivation for this function is the reduction of the underlying data structure `FuzContr` and a gain in efficiency for several computations (e.g., `contrSum`). The resulting controller shall be of type `SimpleContr` and contain at maximum one kind of linguistic entities for each input and output, i.e., all input/output entities shall have the same target. But, notice that the history of creation of the underlying controller is automatically deleted when using `contrMelt` and cannot be reconstructed. We start with sum controllers.

```
contrMelt :: (Eq e, Eq obj) => FuzContr e obj -> FuzContr e obj
contrMelt fc@(SumContr h sS tS derJoin subctrs) =
  let
    subms  = map contrMelt subctrs;   ag    = aggr $ head subms
    doms   = map contrDom subms;      rans  = map contrRan subms
    lEIn   = map lingEntIn subms;     lEOut = map lingEntOut subms
    rs     = concat $ map rules subms
    extLab = case (sS,tS) of
              (True,True)  -> map (\(iF,(s,ts))->(iF,(s++"++",map (++"++") ts)))
              (True,False) -> map (\(iF,(s,ts))->(iF,(s++"++",ts)))
              (False,True) -> map (\(iF,(s,ts))->(iF,(s,map(++"++") ts)))
              (False,False)-> id
  in SimpleContr h ag
               (extLab $ meltRs (comm ag) (idem ag) (head rs) [] $ tail rs)
               (nub $ concat $ if sS then injR [] doms lEIn  else lEIn)
               (nub $ concat $ if tS then injR [] rans lEOut else lEOut)
  where
    injR us [l]    (x:xs) = [map (\le -> LingEntity (entLabel le++"++")
                                                   (fExtTrg us (entRel le)[]))x]
    injR us (l:ls) (x:xs) =  map (\le -> LingEntity (entLabel le++"++")
                                                   (fExtTrg us (entRel le)ls)) x
                            : injR (us++[l]) ls xs
    meltRs _ _ x              []     []  = [x]
    meltRs c i x              (r:rs) []  = x:meltRs c i r [] rs
    meltRs c i x@(iF,(l,trgs)) rs     (t@(iF2,(l2,trgs2)):ts)=
         if (l==l2) && (opLabel iF==opLabel iF2) && c then
             meltRs c i (iF,(l,if i then nub $ trgs++trgs2
                              else trgs++trgs2))         rs ts
         else meltRs c i x (t:rs) ts
```

The header need not be computed again. This has already been done with the creation of the data structure using `contrSum`. The aggregation function is delivered by the melted subcontrollers (`subms`). Notice that we choose the first element of this list. Indeed, we could choose an arbitrary element since we implicitly presuppose that the subcontrollers are combinable.
The computation of the new linguistic entities and rulebase is a straightforward realization of Definitions 4.2.5 and 4.2.6. Notice that the resulting input and output entities get new labels if they are extended to new targets. For the computation of the new rulebase we explicitly consider whether the aggregation function is commutative (`c`) or idempotent (`i`). Idempotency implies that we can omit duplicate output entities within a rule, and commutativity allows, presupposed equal interpretation functions, to put two rules together, which have the same input entity. Finally, we again want to mention that this construction does not affect the input-output behavior of the resulting controller (cf. Theorem 4.2.1).
Melting two product controllers is done analogously.

```
contrMelt fc@(ProdContr h sP tP derMeet subctrs) =
  let g        = contrCat fc;          subms = map contrMelt subctrs
      lEIn     = map lingEntIn subms;  lEOut = map lingEntOut subms
      compG    = gog_comp g;           convG = gog_converse g
      concProj = zipWith (\p (LingEntity l r) ->
                            (LingEntity l (compG r $ convG p)))
      extLab   = case (sP,tP) of
```

```
              (True,True)   -> map (\(s,ts) -> (s++"&&",map (++"&&") ts))
              (True,False)  -> map (\(s,ts) -> (s++"&&",ts))
              (False,True)  -> map (\(s,ts) -> (s,map(++"&&") ts))
              (False,False) -> id
 in SimpleContr h (aggr $ head subms)
               (newRules $ map rules subms)
               (nub $ prodEnts  concProj g
                                lEIn (map contrDom subms) sP)
               (nub $ prodEnts  concProj g
                                lEOut (map contrRan subms) tP)
 where
   newRules []         = []
   newRules [rs]       = rs
   newRules (rs:ruls)  = [(iF,(l ++ (if sP then "**" else "&&") ++ l2,
                                [n++ (if tP then "**" else "&&") ++ n2
                                 | n<-ls,n2<-ls2]))
                            |(iF,(l,ls)) <- rs, (iF2,(l2,ls2)) <- newRules ruls]
   prodEnts _ _      [] _  _    = error ("Empty entity list while "++
                                         "computing the relational products!")
   prodEnts f g ents trgs cP =
       let es          = prodEnts' ents
           cp          = crossProd trgs
           pr          = projections g (entryLat h) trgs cp
           (f',t,conc) = if cP then (f pr,cp,"**")
                         else (id,head trgs,"&&")
       in map (\en -> LingEntity (tail $ tail $ foldl (\x y -> x ++ conc ++
                                                         entLabel y) "" en)
                                 (foldl (\x y -> gog_derOp g "Meet"
                                                 (opLoos derMeet) x (entRel y))
                                                 (gog_top g (unitObj h) t)
                                                 $ f' en)  ) es
   prodEnts' []        = error ("Empty entity list while computing " ++
                                "the relational products!")
   prodEnts' [es]      = map (:[]) es
   prodEnts' (es:ents) = let en = prodEnts' ents
                              in  concat [[e1:es2 | es2<-en] | e1<-es]
```

The key functionality is established by `prodEnts` and `prodEnts'`, respectively. The second function creates all pairs of linguistic entities arising from the melted subcontrollers. These pairs are represented by lists. The lists are the base for `prodEnts` to generate the projections,

if necessary, and then compute the resulting linguistic entities (cf. Definitions 4.2.8, 4.2.9). The entities of the melted controller also get new labels. If they are extended using the projections (i.e., `sP` or `tP` are `True`), they are concatenated by `**`. Otherwise, we use `&&`. This labeling is a good indicator if one wants to list the resulting controller using the `show` function.
With product controllers we have to remember that the input-output behavior of the melted controller is affected. This immediately follows from Theorem 4.2.2. But, we have in every case that calling

```
core $ contrProd sProd tProd c1 c2 derM
```

and

```
core (contrMelt $ contrProd sProd tProd c1 c2 derM)
```

deliver the same result for given parameters `sProd`, `tProd`, `c1`, `c2` and `derM`. The same is true for `contrSum`.

The standard case of `contrMelt` (for simple controllers) is trivial.

```
contrMelt f = f
```

Finally, we want to introduce predefined functions to infer the output for a given controller and a given input.

```
fuzzify fc x   = unFunc (fuzz $ header fc) x
apply fc x     = binFunc (appl $ header fc) x $ core fc
defuzzify fc y = unFunc (defuzz $ header fc) y
infer fc       = defuzzify fc . apply fc . fuzzify fc
```

They can be used quite intuitively.

With these combinators a convenient use of the module should be possible. The user now has the chance to manually and (in wide areas) automatically construct and manipulate fuzzy controllers.

4.4 Example controller

In this section we want to provide the module

```
module FContrTest where
```

```
import FContr
import Lattice
import LFuzzyRel
import LFuzzyRelLattices
import List (elemIndex)
```

in which we develop a little example controller to show how to use `FContr`. With the example we do not aim at a sophisticated controller, but try to involve as many operations as possible.

Suppose we have the crossroad shown in Figure 4.6 where the traffic is regulated by two traffic lights "L1" and "L2". The signal flow of both traffic lights shall be dependent from

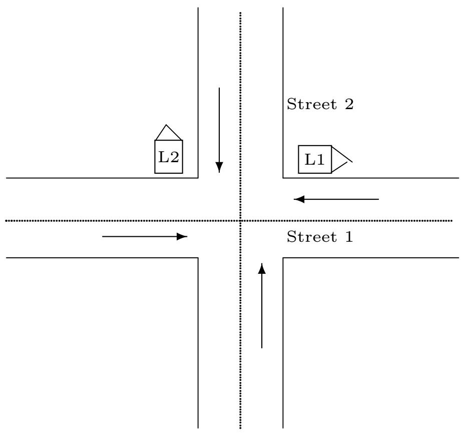

Figure 4.6: A crossroad controlled by a fuzzy controller

the amount of traffic on both streets. Our aim is to develop a controller steering both "L1" and "L2". This can be achieved by creating two separate controllers and then applying the cross product operation to them. This corresponds to a kind of "concurrent" modeling.

A first simple approach to control a single traffic light is the following. We want to make the controller dependent from the amount of traffic on the two streets. Hence, we have to get an input from two sensors each measuring the traffic of its street. We denote the output domain of a sensor by A and, thus, have the input space $A \times A$ for the controllers. We provide the linguistic input entities

$LT : I \to A \times A$	- low traffic,
MT	- medium traffic,
HT	- high traffic.

The output entities are the following :

$TR : I \to B$	- turn red,
S	- stay unchanged,
TG	- turn green.

They obviously directly correspond to the possible actions a traffic light can do. The range B includes the possible states of a traffic light. Normally, we would have *red*, *yellow* and *green*. But, *yellow* is a time-dependent (and not traffic-dependent) state. Since we do not have modeled time in our approach, we omit this state so that we have $B := \{red, green\}$.

Now, the question arises which entry lattice to choose. An entry in the input entities shall express to which degree the traffic on street one and two, respectively, is low, medium or high. Hence, we choose $[0,1] \times [0,1]$ as the underlying entry lattice.

At this point we have enough information to model the single controllers "L1" and "L2". First, we provide the entry lattice. It is clear that we have to use an approximation of $[0,1] \times [0,1]$. We use a granularity of $\frac{1}{100}$.

```
gran = (100::Int)
eLat :: Lat (Int,Int)
eLat = let els = [ (x,y) | x<-[0..gran], y<-[0..gran]]
       in Lat{lat_poSet = PoSet { poSet_isElem   = flip elem $ els
                                 ,poSet_elements = els
                                 ,poSet_lEq      = \(x1,y1) (x2,y2)->x1<=x2 &&
                                                                     y1<=y2 }
             ,lat_sup     = \(x1,y1) (x2,y2) -> (max x1 x2,max y1 y2)
             ,lat_inf     = \(x1,y1) (x2,y2) -> (min x1 x2,min y1 y2)
             ,lat_botEl   = (0,0)
             ,lat_topEl   = (gran,gran)
             ,lat_atomS   = [(1,0),(0,1)]
             ,lat_jIrredS = [(0,x)|x<-[1..gran]]++[(x,0)|x<-[1..gran]]
             ,lat_mIrredS = [(gran,x)|x<-[0..gran-1]]++[(x,gran)|x<-[0..gran-1]]}
```

The parameter `gran` sets the granularity. The lattice then is instantiated quite intuitively. The only thing to mention is that we use the componentwise ordering

$$(x_1, y_1) \leq (x_2, y_2) \;:\Leftrightarrow\; x_1 \leq x_2 \text{ and } y_1 \leq y_2$$

on $[0,1] \times [0,1]$ so that we cannot rely on the standard `Prelude` function `(<=)`. The definition above is consistent which is shown by the following session on Hugs or GHC.

```
FContrTest> lat_atomS eLat == atomSet (lat_lSemiLat eLat)
True
```

```
FContrTest> lat_mIrredS eLat == mIrredSet eLat
True
FContrTest> lat_jIrredS eLat == jIrredSet eLat
True
FContrTest> testLattice eLat []
[]
```

Notice that checking both the join-irreducible and meet-irreducible elements before calling `testLattice` is necessary since `testLattice` automatically reduces the underlying tests to these elements (cf. Section 3.1).

Now, we can start to model the linguistic entities. First, some parameters are introduced.

```
unit,trafficSrc,tLightTrg,trafficAB :: [Int]
unit       = [1]
trafficSrc = [0..9]
tLightTrg  = [20,21]
trafficAB = crossProd [trafficSrc,trafficSrc]
```

With this `unit` delivers the unit object which constitutes the source of the linguistic entities. Furthermore, `trafficSrc` gives the range of the measuring devices for the traffic and can be interpreted as the amount of cars at the crossroad on the respective street. Hence, it corresponds to the set A introduced above. The target of the output entities (B from above) is provided by the parameter `tLightTrg`. Since we have to type all lists equally, the entries have to be interpreted as *red* (`20`) and *green* (`21`), respectively. Finally, the cross product $A \times A$ is delivered by `trafficAB`.

We want to construct our linguistic input entities using the intensifying modifier "more or less". This corresponds to the application $F;_* \Xi$ for a given fuzzy set F, modifier Ξ and a derived operation $;_*$. We exemplary explain the procedure while constructing the linguistic entity HT. In the following we use $;$ for $;_*$. First, we introduce two given crisp fuzzy sets F_1 and F_2 over $A \times A$ expressing which amount of cars constitutes a high traffic on street one resp. two. With this we suppose that street one is the main street and, hence, shall be preferred with state *green*.

```
highA = fUpd (fBot eLat [1::Int] trafficAB) [((0,80+i),(gran,0))|i<-[0..19]]
highB = fUpd (fBot eLat [1::Int] trafficAB) [((0,i*10+9),(0,gran))|i<-[0..9]]
```

Notice that `trafficAB` was created using the `crossProd` operation. Together with the explanations of Section 4.3 it then should be clear that `highA` and `highB` are mathematically described by

$$F_1(1,(x,y)) := \begin{cases} (1,0) & \text{, if } x \geq 8 \\ (0,0) & \text{, otherwise} \end{cases}, \qquad F_2(1,(x,y)) := \begin{cases} (0,1) & \text{, if } y = 9 \\ (0,0) & \text{, otherwise} \end{cases}.$$

Obviously, already eight cars on street one are seen as high traffic whereas nine cars are needed on street two. The starting fuzzy sets F_1 and F_2 now have to be modified. Again, we have to provide a granularity how sensitive the controller shall react on little changes in the traffic amount. To avoid unnecessarily many state changes with the traffic lights, a change in the traffic amount lower than two cars on one street shall not affect the behavior. This is modeled by the fuzzy relation $\Xi : A \times A \to A \times A$

$$\begin{aligned} \Xi((x_1,y_1),(x_2,y_2)) \; := (\; & min(1, max(0, 1.2 - 0.2 \cdot |x1 - x2|)), \\ & min(1, max(0, 1.2 - 0.2 \cdot |y1 - y2|))). \end{aligned} \tag{4.6}$$

One can see that $\mathbb{I}_{A \times A} \sqsubseteq \Xi$ holds. If we now compute $F_1; \Xi$, the interpretation of "high traffic" is preserved within F_1. But, everything that is "nearly high traffic" gets a weakened entry. Thus, we, for example, have $(F_1;\Xi)(5,0) = (0.6,0)$. To achieve the overall fuzzy set HT, we finally have to compute $(F_1 \sqcup F_2);\Xi$. With these remarks the Haskell code should be understandable.

```
trAB :: [(Int,Int)]
trAB = [(x,y) | x<-trafficSrc,y<-trafficSrc]
mayBeVal (Just i) = i
mayBeVal Nothing  = 0
xi :: FRel (Int,Int) Int Int
xi = fUpd (fBot eLat trafficAB trafficAB)
      [((mayBeVal $ elemIndex (x1,y1) trAB, mayBeVal $ elemIndex (x2,y2) trAB),
        (min gran $ max 0 $ gran+20-20*abs (x1-x2),
         min gran $ max 0 $ gran+20-20*abs (y1-y2)) )
       | (x1,y1)<-trAB, (x2,y2)<-trAB]

highT   = LingEntity "HT" $ roughly xi (fJoin highA highB) 1
```

The function `trAB` represents $A \times A$. It differs from `trafficAB` because it consists of tuples and, hence, allows a comfortable creation of Ξ. Furthermore, `getMayBeValue` is an auxiliary function which is also needed with `xi`. The implementation of `xi` then is a straightforward realization of Formula 4.6. Finally, HT is computed by `highT` which makes use of the predefined function `roughly` (cf. Section 4.3).
The linguistic entities MT and LT for medium resp. low traffic are provided analogously.

```
upd1    = fUpd (fBot eLat unit trafficAB)
mediumA = upd1 [((0,30+i),(gran,0))|i<-[0..9]]
mediumB = upd1 [((0,i*10+4),(0,gran))|i<-[0..9]]

lowA = upd1 [((0,i),(gran,0))|i<-[0..9]]
lowB = upd1 [((0,i*10),(0,gran))|i<-[0..9]]

mediumT = LingEntity "MT" $ roughly xi (fJoin mediumA mediumB) 1
lowT    = LingEntity "LT" $ roughly xi (fJoin lowA lowB) 1
```

Now, we switch to the output entities. They are realized as follows.

```
upd2 = fUpd (fBot eLat unit tLightTrg)
tRed   = LingEntity "TR" $ upd2 [((0,0),lat_topEl eLat)]
tGreen = LingEntity "TG" $ upd2 [((0,1),lat_topEl eLat)]
stay   = LingEntity "S"  $ upd2 [((0,0),(div gran 2,div gran 2)),
                                 ((0,1),(div gran 2,div gran 2))]
```

The entities TR and TG, implemented by `tRed` resp. `tGreen`, are crisp whereas S is not. This representation is chosen arbitrarily.

The input and output entities have to be connected within the rulebase. With the model developed so far we choose a very intuitive variant.

If x is HT then y is TG
If x is MT then y is S
If x is LT then y is TR

The rulebase demands to turn red when there is only low traffic on the respective street. Analogously, turning green is demanded with high traffic. Medium traffic shall cause the traffic light to stay unchanged.

Now, we construct "L1" and "L2". Throughout this example we want all rules to be interpreted by ;. The application function of the resulting controllers also shall be the common composition operator. Furthermore, we use the identity operator as fuzzification, and join is used as aggregation function. Hence, we introduce the following abbreviations.

```
infOp, supOp :: DerOp (Int,Int)
infOp       = DerOp "Inf"    (lat_infLoos eLat) True True
supOp       = DerOp "Sup"    (lat_supLoos eLat) True True
```

```
compBinFunc :: BinFunc (FRel (Int,Int) Int Int)
compBinFunc = BinFunc "Comp" fComp
```

The controller for street one is computed as follows.

```
deFuzz sc = UnFunc "Res" (fDown . fRRes sc)
tLightCtrA =  contrUpdEntIn  "" [(infOp,(lowT,["TR"])),
                                 (infOp,(highT,["TG"])),
                                 (infOp,(mediumT,["S"]))] $
              contrUpdEntOut "" [(infOp,(tRed,[])),
                                 (infOp,(tGreen,[])),
                                 (infOp,(stay,[]))] $
              contrUpdDefuzz (deFuzz $ fScalar eLat (80,0) unit) $
              contrUpdAggr supOp $
              contrUpdAppl compBinFunc $
              emptyContr "TLA" unit eLat
```

The construction is delivered as expected. Notice that we do not explicitly have to provide the identity function as fuzzification since `emptyContr` uses it per definition (cf. Section 4.3). The only things we have to set are the aggregation and defuzzification function as well as the linguistic entities for input and output, respectively. As defuzzification we choose the cut approach with the function $\Delta(y) := (\alpha_I^{(0.8,0)} \backslash y)^{\downarrow}$. Of course, the parameter $(0.8, 0)$ strongly affects the behavior of the resulting controller and has to be investigated properly. The second component of this tuple is set to zero since we want to control street one.

The controller for the second street is implemented analogously.

```
tLightCtrB =  contrUpdEntIn  "" [(infOp,(lowT,["TR"])),
                                 (infOp,(highT,["TG"])),
                                 (infOp,(mediumT,["S"]))] $
              contrUpdEntOut "" [(infOp,(tRed,[])),
                                 (infOp,(tGreen,[])),
                                 (infOp,(stay,[]))] $
              contrUpdDefuzz (deFuzz $ fScalar eLat (0,80) unit) $
              contrUpdAggr supOp $
              contrUpdAppl compBinFunc $
              emptyContr "TLB" unit eLat
```

The only difference is the label and the defuzzification function. Here the first component of the scalar for the cut computation is set to zero since we want to control street two.

Finally, we have to put `tLightCtrA` and `tLightCtrB` together to one product controller. This is done by the function

```
tLightCtr' = contrProd False True tLightCtrA tLightCtrB infOp
```

which uses the standard cross product (indicated by `infOp`). Notice that only the range of the resulting controller has to be extended so that it steers two traffic lights. It is clear that now the critical states (*green*,*green*) and (*red*,*red*) can occur. Hence, we modify the defuzzification function of `tLightCtr'` such that these states are excluded. We choose the most simple variant by providing a function $f : A \times A \to A \times A$ that can be applied to the defuzzified output of `tLightCtr'`. It shall be of the following form :

$$\begin{aligned}
(green,green) &\mapsto (red,green),\\
(green,red) &\mapsto (green,red),\\
(red,green) &\mapsto (red,green),\\
(red,red) &\mapsto (green,red).
\end{aligned}$$

Obviously, the non-critical states are not changed. But, when both streets have green, we decide to turn red on the main street and let the cars of street two pass. This is done due to fairness considerations. Otherwise it would be possible that “L2” never turns green (e.g., during the peak time). The traffic flow on the main street should not be affected too strong with this regulation since “L1” and “L2” are programmed to prefer street one with state *green* if less than nine cars are waiting on street two.
Finally, the state (*red*,*red*) shall be turned into (*green*,*red*) which makes even clearer that we prefer the main street with state *green*.

Since $A \times A$ is represented by the list `[20,21,20,21]` which can be thought as

`[(red,red),(red,green),(green,red),(green,green)]`,

we can achieve the intended behavior by applying the matrix

$$R := \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

to the defuzzified output. If we denote the old defuzzification function of `tLightCtr'` by Δ', we have the new defuzzification function $\Delta(y) := \Delta'(y); R$.

The following Haskell code realizes our intent.

```
tLightCtr = let df' = defuzz $ header tLightCtr'
                tLT = crossProd [tLightTrg,tLightTrg]
                tEl = lat_topEl eLat
                r   = fUpd (fBot eLat tLT tLT)
                           [((0,2),tEl),((1,1),tEl),((2,2),tEl),((3,1),tEl)]
            in contrUpdDefuzz (UnFunc (unLabel df') $
                                     \x -> fComp (unFunc df' x) r) tLightCtr'
```

Now, we examplary want to have a look at the rulebase of `tLightCtr` to see the effect of the `contrProd` operation. We compute the rulebase as follows.

```
FContrTest> rules $ contrMelt tLightCtr
[("Inf",("LT&&LT",["TR**TR"])),("Inf",("LT&&HT",["TR**TG"])),
("Inf",("LT&&MT",["TR**S"])),("Inf",("HT&&LT",["TG**TR"])),
("Inf",("HT&&HT",["TG**TG"])),("Inf",("HT&&MT",["TG**S"])),
("Inf",("MT&&LT",["S**TR"])),("Inf",("MT&&HT",["S**TG"])),
("Inf",("MT&&MT",["S**S"]))]
```

Here we see that indeed only the output entities are extended (indicated by the concatenator `**`). This rulebase is the direct realization of our intended controller.
The effect of the extended defuzzification function of `tLightTrg` can be seen by the following computations.

```
FContrTest>infer tLightCtr' $ fUpd (fBot eLat unit $ contrDom tLightCtr')
                               [((0,89),(100,100))]
Source: [1]
Target: [20,21,20,21]
Rel   :
[[(0,0),(0,0),(0,0),(100,100)]]

FContrTest>infer tLightCtr $ fUpd (fBot eLat unit $ contrDom tLightCtr)
                              [((0,89),(100,100))]
Source: [1]
Target: [20,21,20,21]
Rel   :
[[(0,0),(100,100),(0,0),(0,0)]]
```

With the first call we compute the resulting state of the two traffic lights when we have eight cars on the main street and nine cars on street two. The result indeed indicates that this causes both “L1” and “L2” to turn green if they are controlled by `tLightCtr'`.

With the second call we use the modified controller `tLightCtr`. It computes state (*red*,*green*) for the same amount of traffic.

Now, suppose we want to introduce an additional signal which causes the traffic lights to go inactive (often indicated by a flashing yellow light). For this purpose we provide a new controller as follows.

```
offSrc   = unit
offCtr = let offEnt l = LingEntity l $ fTop eLat [1] offSrc
         in  contrUpdEntIn  "" [(infOp,(offEnt "OffI",["OffO"]))] $
             contrUpdEntOut "" [(infOp,(offEnt "OffO",[]))] $
             contrUpdAppl compBinFunc $
             emptyContr "Off" unit eLat
```

It does nothing more than to react on a special input signal and generate the corresponding output signal. If we combine this controller and `tLightCtr` by

```
extTLightCtr = contrSum True True tLightCtr offCtr supOp
```

the resulting controller can be turned off by a special signal. The following computation makes this clear.

```
FContrTest> apply extTLightCtr $ fUpd (fBot eLat unit $ contrDom extTLightCtr)
                                      [((0,100),(100,100))]
Source: [1]
Target: [20,21,20,21,1]
Rel   :
[[(0,0),(0,0),(0,0),(0,0),(100,100)]]
```

The domain of `extTLightCtr` is $(A \times A) + I$. Hence, we have 101 entries in the domain such that the last entry represents the *off* signal. Using this fact, we create the crisp input fuzzy set shown above and, hence, the lights are turned off after the application of this input to the core of `extTLightCtr`.

The computation

```
FContrTest> infer extTLightCtr $ fUpd (fBot eLat unit $ contrDom extTLightCtr)
                                      [((0,90),(100,100))]
Source: [1]
Target: [20,21,20,21,1]
Rel   :
[[(0,0),(0,0),(100,100),(0,0),(0,0)]]
```

shows that the controller does not generate the signal to turn off if we have an input representing the amount of traffic. We choose position 90 of the domain which means that we have nine cars on street one and zero cars on street two. Hence, the controller infers the output shown above. It corresponds to the tuple (*green*,*red*), i.e., we have *green* on street one and *red* on street two.

Chapter 5

Conclusion

In this thesis we, on the one hand, provided a collection of different Haskell modules to make it possible to explore Goguen categories using the RATH system. For this purpose we had to introduce suitable data structures and functions to create, manipulate and test different lattice-structures. How these functions can be used properly was shown with the module `StandardLattices` where we implemented some standard lattice constructions.
The `Lattice` module delivered the necessary auxiliary functions to implement the `Goguen` module which provides the key functionality to include Goguen categories into RATH and test their correct instantiation. The main effort with this module was to implement the test routines efficiently. Especially the antimorphism property of the underlying definition caused some problems which could be solved quite well.

On the other hand, this thesis covers a very important application of Goguen categories — fuzzy controllers. We first introduced the module `LFuzzyRel` which constitutes a framework for a comfortable handling of $\mathcal{L}$-fuzzy relations. With this framework we also support derived operations from lattice-ordered semigroups such that this important construction can be applied. Furthermore, `LFuzzyRel` is used to create the standard model of Goguen categories.
With these preparations we then were able to provide suitable combinators to create and test fuzzy controllers within the abstract theory of Goguen categories. To do so, we motivated why the linguistic model (thus, Mamdani inference) is good to handle by $\mathcal{L}$-fuzzy relations and, hence, by Goguen categories. After that, we defined when two controllers are combinable, introduced a set of operations on combinable controllers (e.g., derived direct sum, derived cross product) and examined their influence on the input-output behavior

of the resulting controller. We saw that the derived sum in the case that the controllers are combinable and the derived cross product in the case of Mamdani inference affects the input-output behavior in the expected way.
Thus, we had the mathematical base to provide the module `FContr` which implements these operations as well as all necessary functionality to create and test fuzzy controllers based on the linguistic model. In a final example where we implemented a fuzzy controller that steers two traffic lights at a crossroad, we demonstrated how to use this module.

But, there are still many things to do.
The auxiliary modules `Lattice` and `LFuzzyRel` only provide the most necessary functionality that was needed for the instantiation of Goguen categories and for the `FContr` module. In a future work, these frameworks could be extended such that they provide functionality that goes into deep with the respective topic. Thus, `LFuzzyRel` could, for example, support the construction of fuzzy negation and fuzzy implication operators.
The more important issue of future extensions should be to provide a GUI for the `FContr` module. Thus, the user would have the chance to create and test controllers in a comfortable and clear way.
Finally, the efficiency of the provided algorithms could be improved. Especially with `FContr` comprehensive computations are done. We think that a considerable gain in efficiency can be achieved there if further investigation is done.

List of Figures

Bibliography

[1] G. Grätzer : General Lattice Theory, Birkhäuser, 1978

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

[3] H. Gericke : Theorie der Verbände, Hochschultaschenbücher Verlag, Bibliographisches Institut AG, Mannheim, 1967

[4] Zadeh : Fuzzy Sets, Information and Control 8, pp. 338-353, 1965

[5] H. Thiele : Einführung in die Fuzzy Logik, University of Dortmund, 1993

[6] J. A. Goguen : $\mathcal{L}$-fuzzy Sets, J. Math. Anal. Appl. 18, pp. 145-157, 1967

[7] Robert Babuška : Fuzzy and Neural Control, Faculty of Information Technology and Systems, Control Engineering Laboratory, Delft University of Technology, Delft, Netherlands, 2001

[8] R. Jager : Fuzzy Logic in Control, PhD Thesis, Delft University of Technology, Delft, Netherlands, 1995

[9] P. Freyd, A. Scedrov : Categories, Allegories, North-Holland, Amsterdam, 1990

[10] Y. Kawahara, H. Furusawa : An Algebraic Formalisation of Fuzzy Relations, Fuzzy Sets and Systems 101, pp. 125-135, 1999

[11] M. Winter : A New Algebraic Approach to $\mathcal{L}$-fuzzy Relations Convenient to Study Crispness, INS Information Sciences 139/3-4, pp. 233-252, 2001

[12] M. Winter : Goguen Categories: An Algebraic Approach to $\mathcal{L}$-fuzzy Relations, PhD Habilitation, Department of Computer Science, Institute for Mathematics and Theoretical Computer Science, University of the Federal Armed Forces, Munich, 2000

[13] M. Winter : Derived Operations in Goguen Categories, TAC Theory and Applications, of Categories, Vol.10, No. 11, pp. 220-247, 2002

[14] G. Schmidt, T. Ströhlein : Relationen und Graphen, Springer, 1989

[15] G. Schmidt, T. Ströhlein : Relations and Graphs, Discrete Mathematics for Computer Scientists, EATCS Monographs on Theoretical Computer Science, Springer, 1993

[16] H. Furusawa, W. Kahl : A Study on Symmetric Quotients, University of the Federal Armed Forces Munich, 1998

[17] W. Kahl, G. Schmidt : Exploring (Finite) Relation Algebras Using Tools Written in Haskell, Technical Report No. 2000-02, Department of Computer Science, Institute for Software Technology, University of the Federal Armed Forces, Munich, 2000

[18] M. Winter : Strukturtheorie heterogener Relationenalgebren mit Anwendung auf Nichtdeterminismus in Programmiersprachen, Dissertationsverlag NG Kopierladen GmbH, München, 1998

[19] S. Thompson : The Craft of Functional Programming, second edition, Addison-Wesley, 1999

[20] A. J. T. Davie : An Introduction to Functional Programming Systems using Haskell, Cambridge University Press, 1992

Appendix A

Lattice Instances and Export of the Functions

In the following we want to connect the type class view and the record data structures of our lattice module provided in Section 3.1. We start by instantiating the type classes.

```
module LatticeInstances where
import Lattice
import LatticeClass

instance POrderedSet (PoSet el) el where
    isElem   = poSet_isElem
    elements = poSet_elements
    lEq      = poSet_lEq

instance POrderedSet (LSemiLat el) el where
    isElem   = lSemiLat_isElem
    elements = lSemiLat_elements
    lEq      = lSemiLat_lEq

instance POrderedSet (USemiLat el) el where
    isElem   = uSemiLat_isElem
    elements = uSemiLat_elements
    lEq      = uSemiLat_lEq

instance POrderedSet (RelCompLat el) el where
    isElem   = relCompLat_isElem
    elements = relCompLat_elements
    lEq      = relCompLat_lEq

instance POrderedSet (Lat el) el where
    isElem   = lat_isElem
    elements = lat_elements
    lEq      = lat_lEq

instance POrderedSet (CompLat el) el where
    isElem   = compLat_isElem
    elements = compLat_elements
    lEq      = compLat_lEq

instance LoSemiLattice (LSemiLat el) el where
    inf   = lSemiLat_inf
    botEl = lSemiLat_botEl
    atomS = lSemiLat_atomS

instance LoSemiLattice (RelCompLat el) el where
    inf   = relCompLat_inf
    botEl = relCompLat_botEl
    atomS = relCompLat_atomS

instance LoSemiLattice (Lat el) el where
    inf   = lat_inf
    botEl = lat_botEl
    atomS = lat_atomS

instance LoSemiLattice (CompLat el) el where
    inf   = compLat_inf
    botEl = compLat_botEl
    atomS = compLat_atomS

instance UpSemiLattice (USemiLat el) el where
    sup   = uSemiLat_sup
    topEl = uSemiLat_topEl

instance UpSemiLattice (Lat el) el where
    sup   = lat_sup
    topEl = lat_topEl
```

```
instance UpSemiLattice (CompLat el) el where
    sup   = compLat_sup
    topEl = compLat_topEl

instance RelCompLattice (RelCompLat el) el where
    relComplem = relCompLat_relComplem

instance Lattice (Lat el) el where
    jIrredS = lat_jIrredS
    mIrredS = lat_mIrredS

instance Lattice (CompLat el) el where
    jIrredS = compLat_jIrredS
    mIrredS = compLat_mIrredS

instance CompLattice (CompLat el) el where
    complem = compLat_complem
```

Furthermore, we need the reversed instances to export the tests and functions.

```
revPoSet :: (POrderedSet p el) => p -> PoSet el
revPoSet p = PoSet { poSet_isElem   = isElem p
                    ,poSet_elements = elements p
                    ,poSet_lEq      = lEq p      }

revLSemiLat :: (LoSemiLattice l el) => l -> LSemiLat el
revLSemiLat l = LSemiLat { lSemiLat_poSet = revPoSet l
                          ,lSemiLat_inf   = inf l
                          ,lSemiLat_botEl = botEl l
                          ,lSemiLat_atomS = atomS l }

revUSemiLat :: (UpSemiLattice l el) => l -> USemiLat el
revUSemiLat l = USemiLat { uSemiLat_poSet = revPoSet l
                          ,uSemiLat_sup   = sup l
                          ,uSemiLat_topEl = topEl l }

revRelCompLat :: (RelCompLattice r el) => r -> RelCompLat el
revRelCompLat r = RelCompLat { relCompLat_lSemiLat   = revLSemiLat r
                              ,relCompLat_relComplem = relComplem r }
```

```
revLat :: (Lattice l el) => l -> Lat el
revLat l = Lat { lat_poSet   = revPoSet l
                ,lat_sup     = sup l
                ,lat_inf     = inf l
                ,lat_topEl   = topEl l
                ,lat_botEl   = botEl l
                ,lat_atomS   = atomS l
                ,lat_jIrredS = jIrredS l
                ,lat_mIrredS = mIrredS l }

revCompLat :: (CompLattice c el) => c -> CompLat el
revCompLat c = CompLat { compLat_lat     = revLat c
                        ,compLat_complem = complem c }
```

Hence, we are able to transfer the functions to our type class view. To avoid name space conflicts, we start all function names by `t_`.

```
t_leastFPA ::  (Eq a, Lattice l a) => a -> l -> (a -> a) -> a
t_leastFPA a l = leastFPA a (revLat l)

t_leastFP ::  (Eq a, Lattice l a) => l -> (a -> a) -> a
t_leastFP l = t_leastFPA (botEl l) l

t_latFPs :: (Eq a, Ord a, Lattice l a) => l -> (a -> a) -> (Lat a)
t_latFPs l = latFPs (revLat l)

t_atomSetBy :: (POrderedSet p el) => p -> el -> (el -> el -> el) -> [el] -> [el]
t_atomSetBy p = atomSetBy (revPoSet p)

t_redSetBy :: (Eq el,POrderedSet p el) => p -> (el -> el -> el) -> [el] -> [el]
t_redSetBy p = redSetBy (revPoSet p)

t_irredSetBy :: (Eq el,POrderedSet p el)=>p->el->(el -> el -> el)->[el]->[el]
t_irredSetBy p = irredSetBy (revPoSet p)

t_jRedSet,t_jIrredSet,t_mRedSet,t_mIrredSet :: (Eq el, Lattice l el) => l -> [el]
t_jRedSet   l = redSetBy   (revPoSet l) (sup l) []
t_jIrredSet l = irredSetBy (revPoSet l) (botEl l) (sup l) []
t_mRedSet   l = redSetBy   (revPoSet l) (inf l)           []
```

```
t_mIrredSet l = irredSetBy (revPoSet l) (topEl l) (inf l) []

t_testRefl, t_testTrans :: (POrderedSet p el) => p -> TestRes el
t_testRefl     = testRefl  . revPoSet
t_testTrans    = testTrans . revPoSet

t_testAntiSymm,t_testPoSet :: (Eq el, POrderedSet p el) => p -> TestRes el
t_testAntiSymm = testAntiSymm . revPoSet
t_testPoSet    = testPoSet    . revPoSet

t_testConsBy :: (Eq el,POrderedSet p el) => p -> (el -> el -> el) ->
                [el] -> [el] -> TestRes el
t_testConsBy p = testConsBy (revPoSet p)

t_testJCons :: (Eq el,UpSemiLattice u el) => u -> TestRes el
t_testJCons = testJCons . revUSemiLat

t_testMCons :: (Eq el,LoSemiLattice l el) => l -> TestRes el
t_testMCons = testMCons . revLSemiLat

t_testRelCompCons :: (Eq el,RelCompLattice r el) => r -> TestRes el
t_testRelCompCons = testRelCompCons . revRelCompLat

t_testConsUnBy :: (Eq el,POrderedSet p el) => p -> (el -> el) ->
                  [el] -> [el] -> TestRes el
t_testConsUnBy p = testConsUnBy (revPoSet p)

t_testCompCons :: (Eq el,CompLattice c el) => c -> TestRes el
t_testCompCons = testCompCons . revCompLat

t_testCommBy,t_testIdemBy,t_testAssBy :: (Eq el,POrderedSet p el) =>
                              p -> (el -> el -> el) -> [el] -> TestRes el
t_testCommBy p = testCommBy (revPoSet p)
t_testIdemBy p = testIdemBy (revPoSet p)
t_testAssBy  p = testAssBy  (revPoSet p)

t_testJComm,t_testJIdem,t_testJAss :: (Eq el,UpSemiLattice u el)=>u->TestRes el
t_testJComm = testJComm . revUSemiLat
t_testJIdem = testJIdem . revUSemiLat
t_testJAss  = testJAss  . revUSemiLat
```

```
t_testMComm,t_testMIdem,t_testMAss :: (Eq el,LoSemiLattice l el)=>l->TestRes el
t_testMComm = testMComm . revLSemiLat
t_testMIdem = testMIdem . revLSemiLat
t_testMAss  = testMAss  . revLSemiLat

t_testTopElBy :: (UpSemiLattice u el) => u -> [el] -> TestRes el
t_testTopElBy u = testTopElBy (revUSemiLat u)

t_testBotElBy :: (LoSemiLattice l el) => l -> [el] -> TestRes el
t_testBotElBy l = testBotElBy (revLSemiLat l)

t_testTopEl :: (UpSemiLattice u el) => u -> TestRes el
t_testTopEl = testTopEl . revUSemiLat

t_testBotEl :: (LoSemiLattice l el) => l -> TestRes el
t_testBotEl = testBotEl . revLSemiLat

t_testUpSemiLatticeBy :: (Eq el,UpSemiLattice u el) =>
                         u -> [el] -> [el] -> TestRes el
t_testUpSemiLatticeBy  u = testUpSemiLatticeBy (revUSemiLat u)

t_testLoSemiLatticeBy :: (Eq el,LoSemiLattice l el) =>
                         l -> [el] -> [el] -> TestRes el
t_testLoSemiLatticeBy  l = testLoSemiLatticeBy (revLSemiLat l)

t_testLatticeBy       :: (Eq el,Lattice l el) => l -> [el] -> [el] -> TestRes el
t_testLatticeBy        l = testLatticeBy (revLat l)

t_testUpSemiLattice :: (Eq el,UpSemiLattice u el) => u -> TestRes el
t_testUpSemiLattice = testUpSemiLattice . revUSemiLat

t_testLoSemiLattice :: (Eq el,LoSemiLattice l el) => l -> TestRes el
t_testLoSemiLattice = testLoSemiLattice . revLSemiLat

t_testLattice       :: (Eq el,Lattice l el) => l -> TestRes el
t_testLattice       = testLattice . revLat

t_testModularBy :: (Eq el,Lattice l el) => l -> [el] -> TestRes el
t_testModularBy l = testModularBy (revLat l)
```

```
t_testModular :: (Eq el,Lattice l el) => l -> TestRes el
t_testModular = testModular . revLat

t_testDistrBy :: (Eq el,POrderedSet p el) => p -> (el -> el -> el) ->
                 (el -> el -> el) -> [el] -> TestRes el
t_testDistrBy p = testDistrBy (revPoSet p)

t_testJDistr,t_testMDistr :: (Eq el,Lattice l el) => l -> TestRes el
t_testJDistr = testJDistr . revLat
t_testMDistr = testMDistr . revLat

t_testRelComplBy :: (Eq el,RelCompLattice r el) => r -> [el] -> TestRes el
t_testRelComplBy r = testRelComplBy (revRelCompLat r)

t_testRelCompl :: (Eq el,RelCompLattice r el) => r -> TestRes el
t_testRelCompl = testRelCompl . revRelCompLat

t_testAtomicIrred :: (Eq el,Lattice l el) => l -> TestRes el
t_testAtomicIrred = testAtomicIrred . revLat

t_testAtomicBy :: (Eq el,Lattice l el) => l -> [el] -> TestRes el
t_testAtomicBy l = testAtomicBy (revLat l)

t_testAtomic :: (Eq el,Lattice l el) => l -> TestRes el
t_testAtomic = testAtomic . revLat

t_testComplBy :: (CompLattice c el) => c -> [el] -> TestRes el
t_testComplBy c = testComplBy (revCompLat c)

t_testCompl :: (CompLattice c el) => c -> TestRes el
t_testCompl = testCompl . revCompLat

t_testBoolLatticeBy :: (Eq el,CompLattice c el)=>c -> [el] -> [el] -> TestRes el
t_testBoolLatticeBy c = testBoolLatticeBy (revCompLat c)

t_testBoolLattice,t_testBoolLatticeIrred :: (Eq el,CompLattice c el) =>
                                            c -> TestRes el
t_testBoolLattice      = testBoolLattice . revCompLat
t_testBoolLatticeIrred = testBoolLatticeIrred . revCompLat
```

```
t_testMorphBy :: (Eq el2,POrderedSet p1 el1, POrderedSet p2 el2) => p1 -> p2 ->
                 (el1 -> el1 -> el1) ->
                 (el2 -> el2 -> el2) -> String -> (el1 -> el2) -> TestRes el1
t_testMorphBy p1 p2 = testMorphBy (revPoSet p1) (revPoSet p2)

t_testMorphUnBy :: (Eq el2,POrderedSet p1 el1, POrderedSet p2 el2) => p1 -> p2 ->
                   (el1 -> el1) -> (el2 -> el2) ->
                   String -> (el1 -> el2) -> TestRes el1
t_testMorphUnBy p1 p2 = testMorphUnBy (revPoSet p1) (revPoSet p2)

t_testUpSemiLatMorph :: (Eq el2,UpSemiLattice u1 el1, UpSemiLattice u2 el2) =>
                        u1 -> u2 -> (el1 -> el2) -> TestRes el1
t_testUpSemiLatMorph   u1 u2   = testUpSemiLatMorph (revUSemiLat u1)
                                                    (revUSemiLat u2)

t_testLoSemiLatMorph :: (Eq el2,LoSemiLattice l1 el1, LoSemiLattice l2 el2) =>
                        l1 -> l2 -> (el1 -> el2) -> TestRes el1
t_testLoSemiLatMorph   l1 l2   = testLoSemiLatMorph (revLSemiLat l1)
                                                    (revLSemiLat l2)

t_testLatMorph       :: (Eq el2,Lattice l1 el1, Lattice l2 el2) =>
                        l1 -> l2 -> (el1 -> el2) -> TestRes el1
t_testLatMorph         l1 l2 = testLatMorph (revLat l1) (revLat l2)

t_testUpCoSemiLatMorph :: (Eq el2,UpSemiLattice l1 el1, LoSemiLattice l2 el2) =>
                          l1 -> l2 -> (el1 -> el2) -> TestRes el1
t_testUpCoSemiLatMorph l1 l2   = testUpCoSemiLatMorph (revUSemiLat l1)
                                                      (revLSemiLat l2)

t_testLoCoSemiLatMorph :: (Eq el2,LoSemiLattice l1 el1, UpSemiLattice l2 el2) =>
                          l1 -> l2 -> (el1 -> el2) -> TestRes el1
t_testLoCoSemiLatMorph l1 l2   = testLoCoSemiLatMorph (revLSemiLat l1)
                                                      (revUSemiLat l2)

t_testCoLatMorph :: (Eq el2,Lattice l1 el1 ,Lattice l2 el2) =>
                    l1 -> l2 -> (el1 -> el2) -> TestRes el1
t_testCoLatMorph       l1 l2 = testCoLatMorph (revLat l1) (revLat l2)

t_testBoolLatMorph :: (Eq el2,CompLattice c1 el1, CompLattice c2 el2) =>
```

```
                     c1 -> c2 -> (el1 -> el2) -> TestRes el1
t_testBoolLatMorph     c1 c2 = testBoolLatMorph (revCompLat c1) (revCompLat c2)

t_testMonoFunc, t_testAntiFunc :: (POrderedSet p el)=>p->(el -> el)->TestRes el
t_testMonoFunc p = testMonoFunc (revPoSet p)
t_testAntiFunc p = testAntiFunc (revPoSet p)
```

Appendix B

Goguen Instances and Export of the Functions

In the following we want to provide instances for the class `GoguenCat` provided in Section 3.3. Furthermore, the according test routines shall be exported. Thus, we connect our `Goguen` module and the RATH system.

```
module GoguenInstances where


import RelAlgClasses
import RelAlgInstances (dedDict)
import GoguenClass
import Goguen
import RelAlg


instance Category         (Gog loos obj mor) obj mor where
    isObj   = gog_isObj
    isMor   = gog_isMor
    objects = gog_objects
    homset  = gog_homset
    source  = gog_source
    target  = gog_target
    idmor   = gog_idmor
    comp    = gog_comp


instance Allegory         (Gog loos obj mor) obj mor where
    converse = gog_converse
    meet     = gog_meet
    incl     = gog_incl

instance DistribAllegory  (Gog loos obj mor) obj mor where
    join   = gog_join
    bottom = gog_bottom

instance DivisionAllegory (Gog loos obj mor) obj mor where
    rres = gog_rres
    lres = gog_lres
    syq  = gog_syq

instance DedCat           (Gog loos obj mor) obj mor where
    top = gog_top

instance GoguenCat        (Gog loos obj mor) loos obj mor where
    up    = gog_up
    down  = gog_down
    derOp = gog_derOp
```

Before we can export the tests, we need the reverse instance of the `Goguen` class.

```
gogDict :: GoguenCat g l obj mor => g -> Gog l obj mor
gogDict g = Gog { gog_up    = up g
                ,gog_down  = down g
                ,gog_derOp = derOp g }
```

Hence, the consistency tests and necessary functions are exported to be available within the type class view. Notice that we cannot export those functions that are parametrized by given functions to determine the scalars or crisp relations (e.g., `antiMorphBy`). To do so, we would, for example, have to transform a function `f` of type

```
f::(DedCat d obj mor) => d -> obj -> [mor]
```

into a new function

```
f'::  Ded obj mor -> obj -> [mor]
```

which is, in general, not possible. Thus, we only export the tests using the standard functions for the determination of the scalars and crisp relations, respectively.

```
t_antiMorph :: (Eq mor,GoguenCat g l obj mor)  => g -> obj -> obj -> [mor -> mor]
t_antiMorph g = antiMorph $ gogDict g
```

```
t_testAntiMorph    :: (Eq obj, Eq mor, GoguenCat g l obj mor) =>
                        g -> obj -> obj -> TestResult obj mor
t_testAntiMorph    g = testAntiMorph $ gogDict g

t_testAntiMorphAll :: (Eq obj,Eq mor,GoguenCat g l obj mor) =>
                        g -> TestResult obj mor
t_testAntiMorphAll g = testAntiMorphAll $ gogDict g

t_testUpDown :: (Eq obj, Eq mor, GoguenCat g l obj mor) =>
                g -> obj -> obj -> obj -> TestResult obj mor
t_testUpDown g = testUpDown $ gogDict g

t_testUpDownAll :: (Eq obj, Eq mor, GoguenCat g l obj mor) =>
                   g -> TestResult obj mor
t_testUpDownAll g = testUpDownAll $ gogDict g

goguen_TEST :: (Eq obj, Eq mor, GoguenCat g l obj mor) => g -> TestResult obj mor
goguen_TEST g = gog_TEST $ gogDict g

t_testLinear :: (Eq mor, GoguenCat g l obj mor) => g -> TestResult obj mor
t_testLinear g = testLinear $ gogDict g

goguen_fun_TEST :: (Eq mor2,GoguenCat g1 l obj1 mor1,GoguenCat g2 l obj2 mor2)=>
                   g1 -> g2 -> Fun obj2 mor2 obj1 mor1 -> TestResult obj1 mor1
goguen_fun_TEST g1 g2 = gog_fun_TEST (gogDict g1) $ gogDict g2
```