Chapter 2

Preliminaries

In this chapter we introduce some preliminary notions from order theory, answer set programming and fuzzy logic. Note that all notations that are frequently used in this book are included in a list of symbols at the back of the book.

2.1 Order Theory

Definition 2.1 (from [Birkhoff (1973)]). *A binary relation $\leqslant$ on a set P is a **preorder** iff for each x, y, z in P the relation obeys the following conditions:*

(1) Reflexivity: $x \leqslant x$

(2) Transitivity: $(x \leqslant y) \wedge (y \leqslant z) \Rightarrow (x \leqslant z)$

If the binary relation $\leqslant$ furthermore satisfies the anti-symmetry condition

$$(x \leqslant y) \wedge (y \leqslant x) \Rightarrow (x = y)$$

*then $\leqslant$ is called an **order relation**.*

A set P together with a preorder $\leqslant$ on P is called a **preordered set** and is denoted as $(P, \leqslant)$. For a given preorder $\mathscr{P} = (P, \leqslant_{\mathscr{P}})$ we denote $\leqslant_{\mathscr{P}}$ by $\leqslant$ if no confusion is possible. Furthermore for a preordered set $(P, \leqslant)$ the notation $y \geqslant x$ is equivalent to $x \leqslant y$. Last, a preordered set $(P, \leqslant)$ is called finite if P consists of a finite number of elements.
A set P together with an order relation $\leqslant$ is called a **partially ordered set** (short: **poset**). For an order relation $\leqslant$ we abbreviate $(x \leqslant y) \wedge (x \neq y)$ as $x < y$ (or sometimes $y > x$).

Definition 2.2 (from [Birkhoff (1973)]). *Let $(P, \leqslant)$ be a poset, let A be a subset of P and let x be an element of P. Then we define:*

x is a **lower bound** for A iff $\forall y \in A \cdot x \leqslant y$

x is an **upper bound** for A iff $\forall y \in A \cdot y \leqslant x$

Definition 2.3 (from [Birkhoff (1973)]). *Let $(P,\leqslant)$ be a poset, let A be a subset of P and let x be an element of P. Then we define:*

x is the **least element** of A iff x is a lower bound for A and $x \in A$

x is the **greatest element** of A iff x is an upper bound for A and $x \in A$

The least element and greatest element of A are commonly referred to as the **minimum**, resp. **maximum** of A. Note that if they exist, they must be unique [Birkhoff (1973)]. A poset $\mathscr{P}$ is called **bounded** if it contains a minimum and a maximum. We denote these elements with $0_{\mathscr{P}}$, resp. $1_{\mathscr{P}}$. If the poset used is clear from the context we denote them with 0, respectively 1. For a given set A we denote the minimum and maximum as $\min A$, resp. $\max A$, if they exist.

Definition 2.4 (from [Birkhoff (1973)]). *Let $(P,\leqslant)$ be a poset, let A be a subset of P and let x be an element of P. Then we define:*

x is the **infimum** of A iff x is the greatest lower bound for A

x is the **supremum** of A iff x is the least upper bound for A

If the infimum of A exists we denote this with $\inf A$*. Likewise we denote the supremum of A with* $\sup A$*, if it exists.*

Definition 2.5 (from [Birkhoff (1973)]). *A poset $(P,\leqslant)$ is called a **lattice** iff each pair $\{x,y\} \subseteq P$ has an infimum and supremum. If every subset of P has an infimum and supremum we call P a **complete lattice**.*

Note that every finite lattice must necessarily be complete. Furthermore every complete lattice is bounded. Last we would like to remark that $\inf\{x,y\}$ and $\sup\{x,y\}$ are commonly denoted as $x \sqcap y$, respectively $x \sqcup y$. The operation $\sqcap$ is called the **meet** operator and $\sqcup$ the **join** operator.

Definition 2.6 (from [Tarski (1955)]). *Let $(P_1,\leqslant_1)$ and $(P_2,\leqslant_2)$ be two posets and let f be a $P_1 \to P_2$ mapping. Then we define*

f is increasing iff $\forall x,y \in P_1 \cdot x \leqslant_1 y \Rightarrow f(x) \leqslant_2 f(y)$

f is decreasing iff $\forall x,y \in P_2 \cdot x \leqslant_2 y \Rightarrow f(x) \geqslant_2 f(y)$

Note that increasing functions are also commonly called **monotonic**. Tarski proved the following theorem on fixpoints of increasing functions on complete lattices.

Proposition 2.1 (from [Tarski (1955)]). *Let $\mathscr{L}$ be a complete lattice and let $f : \mathscr{L} \to \mathscr{L}$ be an increasing function. Then f has a least fixpoint, i.e. there is an $x \in \mathscr{L}$ such that*

$f(x) = x$ and for all $y \in \mathscr{L}$ such that $f(y) = y$ we have that $x \leqslant y$. We denote the least fixpoint of f as f^*.

It turns out that the least fixpoint of an increasing function f on a complete lattice $\mathscr{L}$ can be computed by iteratively applying f, starting from the least element in the lattice, until a fixpoint is found. We call this an **iterated fixpoint computation**.

Definition 2.7 (from [Baral (2003)]). *Let $\mathscr{L}$ be a complete lattice and let $f : \mathscr{L} \to \mathscr{L}$ be an increasing function. Then*

$$f^0 = 0_{\mathscr{L}}$$
$$f^n = f(f^{n-1}) \text{ if } n \text{ is a successor ordinal}$$
$$f^\alpha = \sup\{f^\beta \mid \beta < \alpha\} \text{ if } \alpha \text{ is a limit ordinal}$$

Proposition 2.2 (from [Baral (2003)]). *Let $\mathscr{L}$ be a complete lattice and let $f : \mathscr{L} \to \mathscr{L}$ be an increasing function. Then $f^* = f^\alpha$, where α is a limit ordinal.*

2.2 Answer Set Programming

In this section we introduce answer set programming. The section is structured as follows. First we begin by introducing the syntax & semantics in 2.2.1. This is followed by a discussion on the the two types of negation that can be considered in answer set programming in 2.2.2. Finally, in 2.2.3, we conclude by discussing the relations between answer set programming and the satisfiability problem in propositional logic.

2.2.1 *Definitions*

In this section we define the syntax and semantics of ASP. The terminology is based on material from [Van Nieuwenborgh (2005)].

2.2.1.1 *Language*

Answer set programming (**ASP**) is built from a language containing terms, atoms and (extended) literals as basic building blocks. A **term** is a **variable** or a **constant**. In this book we adopt the usual convention that variables (constants) are denoted by a symbol starting with an upper-case (lower-case) character. An **atom** is an expression of the form $p(t_1,\ldots,t_n)$, where p is a **predicate** of arity n and $t_1,\ldots,t_n$ are terms. An atom is called **grounded** if it does not contain any variables.

A **literal** is either an atom a or a negated atom $\neg a$ (called a classically or strongly negated literal). An **extended literal** is either a literal or an expression of the form not$l$ (called a **negation-as-failure literal** or **naf-literal**), where l is a literal. An (extended) literal is called **grounded** if its underlying atom is grounded.
For a set of literals L we use not$L$ to denote the set $\{\text{not}\, l \mid l \in L\}$ and $\neg L$ to denote the set $\{\neg l \mid l \in L\}$, where $\neg(\neg l) = l$. With L^+ we denote the positive part of L, i.e. $L^+ = \{a \in L \mid a \text{ is an atom}\}$. Furthermore, for a set of extended literals L we denote with L^- the set of literals underlying the naf-literals in L, i.e. $L^- = \{l \mid \text{not}\, l \in L\}$. For a set of grounded literals L, we say that L is **consistent** iff $L \cap \neg L = \emptyset$. Last, for a set of grounded atoms A, we denote the set of all literals over A as Lit_A, i.e. $Lit_A = A \cup \neg A$.

Definition 2.8. A **normal rule** r is an expression of the form

$$a \leftarrow \beta$$

where a is either the empty set or a singleton containing a literal and β is a set of extended literals. The left-hand side a is called the **head** of the rule, denoted as r_h, whereas the right-hand side β is called the **body** of the rule, and is denoted as r_b.

Rules can be divided in certain classes, depending on conditions satisfied by a and/or β:

(1) A **constraint** is a rule where a is empty.
(2) A **fact** is a rule where β is empty.
(3) A **simple rule** is a rule where $\beta^- = \emptyset$, i.e. a rule with no negation-as-failure literals.
(4) A **positive rule** is a rule where $\beta^- = \emptyset$, $\beta^+ = \beta$ and a is either an atom or empty, i.e. a rule containing only atoms.

Definition 2.9. An **answer set program** (**ASP program**) is a countable set of rules.

Example 2.1. Consider the following program $P_{ex2.1}$.

$$rightOf(john, chris) \leftarrow \tag{2.1}$$

$$rightOf(chris, cathy) \leftarrow \tag{2.2}$$

$$rightOf(X,Y) \leftarrow rightOf(X,Z), rightOf(Z,Y) \tag{2.3}$$

In this program rules (2.1)–(2.2) are facts stating who is sitting immediately on the right of whom. For example rule (2.1) states that *john* is sitting immediately on the right of *chris*. Rule (2.3) then describes that the relation "right of" is transitive. Note that all rules in this program are positive (and thus also simple).

Using the types of rules introduced above we can consider the following types of programs:

(1) A **positive program** contains only positive rules.
(2) A **simple program** contains only simple rules.
(3) A **normal program** can contain any rule.

A positive, simple or normal program is called **constraint-free** if it does not contain any constraints.

2.2.1.2 *Grounding*

In the formulation of the semantics of ASP programs we assume that programs do not contain variables. In the following we explain how we can obtain a grounded version, $gnd(P)$, from a normal program P that contains variables.

Definition 2.10 (from [Van Nieuwenborgh (2005)]). *Let P be a program. The set of all constants appearing in P is called the **Herbrand universe**, denoted $\mathscr{U}_P$. The **Herbrand base** $\mathscr{B}_P$ of P is the set containing all grounded atoms that can be constructed from the predicates in P and the terms in $\mathscr{U}_P$.*

Consider now a rule r in a program P. A **grounded instance** of r is any rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a mapping from the variables occurring in r to the terms in $\mathscr{U}_P$. We denote the set of all ground instances of a rule $r \in P$ by $gnd_{\mathscr{U}_P}(r)$. The **grounded program** P is then defined as

$$gnd(P) = \bigcup_{r \in P} gnd_{\mathscr{U}_P}(r)$$

Example 2.2. Consider the following program $P_{ex2.2}$:

$$p(X,Y) \leftarrow q(X), r(Y)$$
$$q(a) \leftarrow$$
$$r(b) \leftarrow$$

Its grounding is the following program $gnd(P_{ex2.2})$:

$$p(a,a) \leftarrow q(a), r(a)$$
$$p(a,b) \leftarrow q(a), r(b)$$
$$p(b,a) \leftarrow q(b), r(a)$$
$$p(b,b) \leftarrow q(b), r(b)$$

$$q(a) \leftarrow$$
$$r(b) \leftarrow$$

Note that the grounding process can be exponential in the size of the program. Therefore researchers have recently started to devote their attention to the study of more efficient grounding methods (see e.g. [Syrjänen (2001, 2004); Leone *et al.* (2004); Gebser *et al.* (2007)]). In the remainder of this book we assume that all programs are grounded, unless stated otherwise.

2.2.1.3 *Semantics*

In this section we define the *meaning* of ASP programs constructed in the language introduced above. Intuitively, if we model certain knowledge or a certain problem with an ASP program, we want the semantics of the program to capture the knowledge that can be derived.

Formally, the meaning of a program is represented by *interpretations*. If P is a program, then any consistent set $I \subseteq Lit_{\mathscr{B}_P}$ is an **interpretation** of P. For programs without classical negation interpretations are subsets of $\mathscr{B}_P$. An interpretation I is **total** if $\mathscr{B}_P = I \cup \neg I$. An extended literal is true w.r.t. an interpretation I, denoted $I \models l$, iff $l \in I$ if l is not a naf-literal and $I \not\models a$ if l is a naf-literal of the form not$a$. If L is a set of (extended) literals we define $I \models L$ iff $\forall l \in L \cdot I \models l$.

For a rule $r \in P$ of the form $a \leftarrow \beta$ we say that r is **satisfied** by I, denoted $I \models r$, iff $I \not\models \beta$ or $I \models a$.

Definition 2.11. Let P be a program. An interpretation I of P is called a **model** of P iff $\forall r \in P \cdot I \models r$. Furthermore, I is a **minimal model** of P iff I is a model of P and no model J of P exists such that $J \subset I$.

For constraint-free positive programs the minimal model is guaranteed to exist and can be computed using the following monotonic operator.

Definition 2.12 (from [Van Emden and Kowalski (1976)]). *Let P be a constraint-free positive program and let I be an interpretation of P. The* ***immediate consequence operator*** Π_P *of P is a $\mathscr{P}(\mathscr{B}_P) \rightarrow \mathscr{P}(\mathscr{B}_P)$ function defined as*

$$\Pi_P(I) = \{a \mid a \leftarrow \beta \in P \wedge \beta \subseteq I\}$$

It is easy to see that this operator is monotonic, and due to Proposition 2.1 and Proposition 2.2, that it has a least fixpoint that can be computed using an iterated fixpoint compu-

tation. Define $\Pi_P^0 = \emptyset$ and $\Pi_P^i = \Pi_P(\Pi_P^{i-1})$ for $i \geqslant 1$, then Π_P^j is the least fixpoint of Π_P, denoted as $\Pi_P^\star$, iff $\Pi_P(\Pi_P^j) = \Pi_P^j$. In other words, the least fixpoint of Π_P can be computed by iteratively applying Π_P, starting from the empty interpretation, until a fixpoint is found. Note that this computation always ends if $\mathscr{B}_P$ is finite, which we will assume in the remainder of this book. The following proposition shows that this least fixpoint coincides with the minimal model of P, hence the procedure explained above gives us a procedural method for computing the minimal model of a program.

Proposition 2.3 (from [Van Nieuwenborgh (2005)]). *Let P be a constraint-free positive program. Then I is a model of P iff $\Pi_P(I) \subseteq I$. Furthermore, the unique minimal model of P equals the least fixpoint of Π_P.*

Note that from the former it follows that for positive programs the minimal model, if it exists, must be unique. The above definitions can easily be extended to simple programs with constraints. Let us denote by P' the program consisting of the rules in P where we (i) consider classically negated atoms $\neg a$ as fresh atoms and (ii) replace constraint rules of the form $\leftarrow \beta$ by rules of the form $\bot \leftarrow \beta$. Furthermore we extend the definition of inconsistency by saying that a set $I \subseteq \mathscr{B}_P$ is inconsistent iff $\{a, \neg a\} \subseteq I$ for some $a \in \mathscr{B}_P$ or $\bot \in I$. Note that since interpretations (and thus also models) are by definition consistent, they can never contain $\bot$.

Proposition 2.4 (from [Van Nieuwenborgh (2005)]). *Let P be a simple program. An interpretation I is the unique minimal model of P iff I is the unique minimal model of P'.*

Now we can introduce the semantics of answer set programs. We do this in two steps. First we define the answer sets for programs without negation-as-failure.

Definition 2.13. Let P be a simple program. An interpretation A is called an **answer set** of P iff A is the minimal model of P.

For normal programs, i.e. programs containing negation-as-failure, we have to define the semantics differently because the minimal models of these programs do not always correspond to our intuition regarding negation-as-failure.

Example 2.3. Consider the following program $P_{ex2.3}$:

$$person(john) \leftarrow$$
$$suitable\_for\_job(X) \leftarrow person(X), \text{not}\, criminal\_record(X)$$

It is easy to verify that this program has two minimal models, namely $M_1 = \{person(john), criminal\_record(john)\}$ and $M_2 = \{person(john), suitable\_for\_job(john)\}$. Both of these minimal models contain knowledge that was not explicitly stated in our program, i.e. in M_1 we assume that John has a criminal record, whereas in M_2 we suppose the opposite. Only M_2 is intuitively acceptable, however, since the extra knowledge it includes can be inferred using negation-as-failure: due to our failure to deduct that John has a criminal record, we assume that he doesn't.

The above example illustrates that we need a way of selecting the right minimal models of a program. Formally this can be done by starting from a candidate answer set A of P and construct a reduct program P^A that does not contain negation-as-failure. Then, the candidate answer set is a real answer set if it is an answer set of the reduct (i.e. if it is the minimal model of the reduct).

Definition 2.14 (from [Gelfond and Lifschitz (1988)]). *Let P be a normal program and let I be an interpretation of P. The **reduct** of P w.r.t. I, denoted as P^I, is the program*

$$\{a \leftarrow (\beta \setminus \text{not}\,\beta^-) \mid a \leftarrow \beta \in P \wedge (I \models \text{not}\,\beta^-)\}$$

In other words, P^I is obtained by removing all naf-literals not$l$ for which $l \notin I$ from the bodies of the rules in P and removing all rules containing not$l$ for which $l \in I$. Intuitively this means we remove the naf-literals of rules that could be satisfied by I, judged only by looking at the negative information in I, and discard the rules that can never be satisfied by I.

Definition 2.15 (from [Van Nieuwenborgh (2005)]). *Let P be a normal program. An interpretation A is called an **answer set** of P iff A is the minimal model of P^A.*

The following example illustrates that this definition indeed eliminates the unintuitive minimal models.

Example 2.4. Consider again program $P_{ex2.3}$ from Example 2.3 together with its two minimal models M_1 and M_2. Computing the reductions gives us the program $P_{ex2.3}^{M_1}$:

$$person(john) \leftarrow$$

and $P_{ex2.3}^{M_2}$:

$$person(john) \leftarrow$$
$$suitable\_for\_job(john) \leftarrow person(john)$$

It is easy to see that the minimal model of $P_{ex2.3}^{M_1}$ is $\{person(john)\}$, which is not equal to M_1, hence M_1 is not an answer set of $P_{ex2.3}$. The minimal model M_2 is an answer set of $P_{ex2.3}$, however, as M_2 is the minimal model of $P_{ex2.3}^{M_2}$.

In general, the answer sets of a program will be a subset of the minimal models.

Proposition 2.5 (from [Baral (2003)]). *Let P be a normal program. Any answer set of P is a minimal model of P.*

The reverse does not hold, as Example 2.4 shows. Note that a program can have multiple answer sets or even no answer sets, as shown in the following examples.

Example 2.5. Consider program P_{nondet}:

$$a \leftarrow \text{not}\, b$$

$$b \leftarrow \text{not}\, a$$

This program has two answer sets: $A_1 = \{a\}$ and $A_2 = \{b\}$.

Example 2.6. Consider program P_{empty}:

$$p \leftarrow \text{not}\, p$$

This program has no answer sets. Indeed, its only minimal model is $M = \{p\}$, but $P^M = \emptyset$, which has $\emptyset \neq M$ as its answer set.

The fact that programs can have multiple answer sets or no answer sets forms the basis for the **answer set programming paradigm**[Marek and Truszczyński (1999); Niemelä (1999)]. In this paradigm, we solve a certain combinatorial problem by writing an ASP program such that the answer sets of the program correspond to the solutions of the problem. Most often this is done by writing a program in a generate-define-test style, as shown in the following example:

Example 2.7. Consider the problem of coloring the vertices of a graph in either black or white, such that adjacent nodes are colored differently. We can model this problem using the ASP program P_{gc}:

$$black(X) \leftarrow \text{not}\, white(X) \tag{2.4}$$

$$white(X) \leftarrow \text{not}\, black(X) \tag{2.5}$$

$$sim(X,Y) \leftarrow white(X), white(Y) \tag{2.6}$$

$$sim(X,Y) \leftarrow black(X), black(Y) \tag{2.7}$$

$$\leftarrow edge(X,Y), sim(X,Y) \tag{2.8}$$

In this program rules (2.4) and (2.5) are the so-called **generate part**, which generate an arbitrary graph coloring. One can see that the possibility of having two answer sets thus allows us to state non-deterministic choice in our program. Rules (2.6) and (2.7) form the **defining part** of our program, which defines certain concepts that will be used in the **constraint part** consisting of rule (2.8). The latter rule eliminates solutions (i.e. answer sets) in which adjacent nodes are similarly colored.

Note that in the above program there are no rules defining what $edge(X,Y)$ means. This is because an ASP program consists of two parts: a general part describing a solution, as above, and an input part defining a specific instance of the problem. For our graph coloring we can e.g. describe the graph depicted in Fig. 2.1a on the facing page by the following set of facts $F_{2.1a}$:

$$\begin{aligned}
&node(a) \leftarrow \\
&node(b) \leftarrow \\
&node(c) \leftarrow \\
&edge(a,b) \leftarrow \\
&edge(b,a) \leftarrow \\
&edge(a,c) \leftarrow \\
&edge(c,a) \leftarrow
\end{aligned}$$

It is easy to verify that the answer sets of $P_{gc} \cup F_{2.1a}$ are

$$A_1 = I \cup \{black(a), white(b), white(c), sim(b,c)\}$$

and

$$A_2 = I \cup \{white(a), black(b), black(c), sim(b,c)\}$$

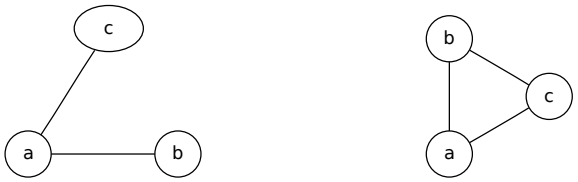
where

$$\begin{aligned}
I = \{&node(a), node(b), node(c), edge(a,b), edge(b,a), edge(a,c), edge(c,a), \\
&edge(b,c), edge(c,b), sim(a,a), sim(b,b), sim(c,c)\}
\end{aligned}$$

Hence the answer sets correspond to the two admissible graph colorings.

If we consider the graph depicted in Fig. 2.1b, however, we find that $P_{gc} \cup F_{2.1b}$ has no answer sets, where $F_{2.1b}$ consists of the input facts encoding this graph.

The above example shows that ASP can be used as a problem solving tool, much in the same vein as constraint satisfaction solvers.



(a) Graph $G_{2.1a}$ (b) Graph $G_{2.1b}$

Fig. 2.1: Input Graphs

2.2.2 *Classical negation vs negation-as-failure*

As mentioned before, in ASP we have two types of negation: classical negation and negation-as-failure. The former states that the negation of an atom a can be explicitly derived, whereas not a is true if we cannot derive a. The important difference between these two constructs is perhaps best illustrated by an example:

Example 2.8. Suppose we are building an ASP program for controlling a car. To ensure safety this car must abide by the traffic rules and so we need to express the usual rule of giving way to the right, i.e. that we have to yield to cars coming from the right. Doing this with negation-as-failure we obtain the rule

$$driveon \leftarrow \text{not}\, carOnRight$$

However, this rule states that if we fail to derive that there is a car on the right, we can drive on. So if it is foggy and our sensors cannot determine whether there is a car on the right, we will drive on, which can have some fatal consequences. Writing this rule with classical negation we obtain

$$driveon \leftarrow \neg carOnRight$$

This rule states that we only drive on when the sensors on the car have derived that no car is coming. Hence in the case of foggy weather we stay safe, though it can take a while until we get to our destination.

Classical negation can be eliminated from the program by introducing for each literal $\neg a$ a new atom a' and adding the constraint $\leftarrow a, a'$ to the program. The constraint ensures that any model will be consistent, and thus ensures that the semantics are preserved. For more details see [Baral (2003)]. Unless stated otherwise, for all programs in the remainder of this chapter we assume that classical negation has been eliminated in this way.

2.2.3 *Links to SAT*

There exist important links between ASP and the boolean satisfiability problem (SAT), which we highlight in this section. We begin by illustrating that the graph coloring program introduced above can be translated into an equivalent and equally concise SAT problem.

Example 2.9. Consider program $P_{gc} \cup F_{2.1a}$ from Example 2.7. Its corresponding SAT problem, denoted as $comp(P_{gc} \cup F_{2.1a})$, is obtained by replacing $\leftarrow$ by $\equiv$, replacing not l by $\neg l$ for any literal l, replacing empty bodies by *True*, replacing empty heads by *False* and grouping rule bodies of rules with the same heads by disjunction:

$$
\begin{aligned}
node(a) &\equiv True \\
node(b) &\equiv True \\
node(c) &\equiv True \\
edge(a,b) &\equiv True \\
edge(b,a) &\equiv True \\
edge(a,c) &\equiv True \\
edge(c,a) &\equiv True \\
black(a) &\equiv \neg white(a) \\
white(a) &\equiv \neg black(a) \\
black(b) &\equiv \neg white(b) \\
white(b) &\equiv \neg black(b) \\
black(c) &\equiv \neg white(c) \\
white(c) &\equiv \neg black(c) \\
sim(a,a) &\equiv (white(a) \wedge white(a)) \vee (black(a) \wedge black(a)) \\
sim(a,b) &\equiv (white(a) \wedge white(b)) \vee (black(a) \wedge black(b)) \\
sim(a,c) &\equiv (white(a) \wedge white(c)) \vee (black(a) \wedge black(c)) \\
sim(b,a) &\equiv (white(b) \wedge white(a)) \vee (black(b) \wedge black(a)) \\
sim(b,b) &\equiv (white(b) \wedge white(b)) \vee (black(b) \wedge black(b)) \\
sim(b,c) &\equiv (white(b) \wedge white(c)) \vee (black(b) \wedge black(c)) \\
sim(c,a) &\equiv (white(c) \wedge white(a)) \vee (black(c) \wedge black(a)) \\
sim(c,b) &\equiv (white(c) \wedge white(b)) \vee (black(c) \wedge black(b)) \\
sim(c,c) &\equiv (white(c) \wedge white(c)) \vee (black(c) \wedge black(c))
\end{aligned}
$$

$$False \equiv edge(a,a) \wedge sim(a,a)$$
$$False \equiv edge(a,b) \wedge sim(a,b)$$
$$False \equiv edge(a,c) \wedge sim(a,c)$$
$$False \equiv edge(b,a) \wedge sim(b,a)$$
$$False \equiv edge(b,b) \wedge sim(b,b)$$
$$False \equiv edge(b,c) \wedge sim(b,c)$$
$$False \equiv edge(c,a) \wedge sim(c,a)$$
$$False \equiv edge(c,b) \wedge sim(c,b)$$
$$False \equiv edge(c,c) \wedge sim(c,c)$$

One can easily verify that answer sets A_1 and A_2 from Example 2.7 are models of the above translation to SAT.

Formally this translation is called the **completion** of an ASP program. It is also commonly called **Clark's completion** after Keith Clark, who originally proposed this correspondence as a method for describing the semantics of negation-as-failure [Clark (1977)].

Definition 2.16 (from [Clark (1977)]). *Let P be a normal program. The **completion** of P, denoted $comp(P)$, is defined as the following set of propositions:*

$$\begin{aligned} comp(P) = \{a \equiv \bigvee\{comp(\beta) \mid a \leftarrow \beta \in P\} \mid a \in \mathscr{B}_P, \beta \neq \emptyset\} \\ \cup \{a \equiv True \mid (a \leftarrow) \in \mathscr{B}_P\} \\ \cup \{False \equiv comp(\beta) \mid (\leftarrow \beta) \in \mathscr{B}_P\} \end{aligned}$$

where $\bigvee(\emptyset) = False$ and $comp(\beta) = b_1 \wedge \ldots \wedge b_n \wedge \neg c_1 \wedge \ldots \wedge \neg c_m$ for $\beta = \{b_1, \ldots, b_n, \text{not}\, c_1, \ldots, \text{not}\, c_m\}$.

In [Fages (1994)], Fages showed that under certain conditions the answer sets of an ASP program and the models of its completion coincide. The question then arises why we need ASP at all and why we cannot just write our problems directly as their SAT encodings? After all, the completion of the graph coloring program introduced above is almost as concise as the grounded program. There are ASP programs for which the models of the completion do not coincide with the answer sets, however. This occurs when there are

atoms that positively depend on other atoms, as shown in the following example.

Example 2.10. Consider the following program $P_{ex2.10}$:

$$a \leftarrow a$$

The answer set of $P_{ex2.10}$ is $\emptyset$. However, its completion $comp(P_{ex2.10})$ has two models: $\emptyset$ and $\{a\}$.

From the above example one might think that answer sets correspond to the minimal models of the completion. This turns out to be false, however, as one can see from the following example.

Example 2.11. Consider the following program $P_{ex2.11}$:

$$a \leftarrow a$$
$$p \leftarrow \text{not}\, p, \text{not}\, a$$

One can easily verify that $P_{ex2.11}$ has no answer sets. However, its completion has the single (and therefore trivially minimal) model $\{a\}$.

While the above shows that in general minimal models of the completion are not answer sets, the reverse does hold: answer sets are minimal models of the completion.

Proposition 2.6 (from [Gelfond and Lifschitz (1988)]). *Let P be a normal answer set program. Then any answer set A of P is a minimal model of the completion $comp(P)$ of P.*

We can now answer the question why we need ASP. While the graph coloring example could be concisely encoded in SAT, this does not hold in general. For example, programs incorporating recursion require a more involved translation [Lin and Zhao (2004)]. However, in many application domains it is quite convenient to define predicates recursively, such as the transitive closure defined by the *rightOf* predicate in Example 2.1. The following program illustrates the use of recursion on the problem of finding Hamilton cycles in a graph.

Example 2.12. Consider the problem of determining Hamilton cycles of a graph, i.e. finding a path in the graph that visits every vertex exactly once. In [Marek and Truszczyński (1999)] the authors propose to encode this problem with the following program P_{hc}:

$$in(U,V) \leftarrow edge(U,V), \text{not}\, out(U,V) \tag{2.9}$$

$$out(U,V) \leftarrow edge(U,V), \text{not}\, in(U,V) \tag{2.10}$$

$$reachable(V) \leftarrow in(v_0,V) \tag{2.11}$$

$$reachable(V) \leftarrow reachable(U), in(U,V) \tag{2.12}$$

$$\leftarrow in(U,V), in(U,W), V \neq W \tag{2.13}$$

$$\leftarrow in(U,W), in(V,W), U \neq V \tag{2.14}$$

$$\leftarrow vertex(U), \text{not}\, reachable(U), in(U,V) \tag{2.15}$$

where in rules (2.13) and (2.14) $V \neq W$ and $U \neq V$ are extensions of the ungrounded ASP language which denote that the grounded instances of these rules where $V = W$, resp. $U = V$ holds should not be included in the grounding of the program. Rules (2.9) and (2.10) are the generate rules, where $in(U,V)$ means edge (U,V) is included in the cycle, and $out(U,V)$ means the edge (U,V) is *not* included in the cycle. Rules (2.11) and (2.12) are the defining part, encoding when a vertex is reachable. Note that we have to explicitly state the starting vertex v_0 for this program to work. Furthermore note that we do not state that $reachable(v_0)$ is necessarily true. This is to ensure that the program adds an edge to v_0, and thus creates a cycle rather than a path. Last, rules (2.13)–(2.15) are the constraints eliminating answer sets in which a vertex is visited twice or a certain node is never visited.
Note that the *reachable* predicate is defined recursively. This can lead to counterintuitive results for the models of the completion. Consider the following input rules I from [Babovich *et al.* (2000)]:

$$vertex(v_0) \leftarrow$$
$$vertex(v_1) \leftarrow$$
$$edge(v_0,v_0) \leftarrow$$
$$edge(v_1,v_1) \leftarrow$$

The grounded program $gnd(P_{gc} \cup I)$ will have no answer sets, as no Hamilton cycle exists in this graph. The set $\{vertex(v_0), vertex(v_1), edge(v_0,v_0), edge(v_1,v_1), in(v_0,v_0), in(v_1,v_1), reachable(v_0), reachable(v_1)\}$ is a model of $comp(gnd(P_{gc}) \cup I)$, however.

2.3 Fuzzy Logic

In this section we introduce fuzzy sets and fuzzy logic. We begin by introducing fuzzy sets in Section 2.3.1, then turn our attention to the common operators of fuzzy logic in Section 2.3.2 and conclude by discussing formal fuzzy logics in Section 2.3.3.

2.3.1 *Fuzzy Sets*

We briefly introduce the concepts from fuzzy set theory that we will use throughout this book.

Definition 2.17. Consider a complete lattice $\mathscr{L}$. An **$\mathscr{L}$-fuzzy set** in a universe X is a mapping from X to $\mathscr{L}$.

We will refer to $([0,1],\leqslant)$-fuzzy sets as just **fuzzy sets**. The *inclusion* of two fuzzy sets is defined as follows:

Definition 2.18. Given two $\mathscr{L}$-fuzzy sets A and B in a universe X we define the **Zadeh inclusion** $A \subseteq B$ of A and B as follows:

$$A \subseteq B \equiv \forall x \in X \cdot A(x) \leqslant B(x)$$

It is often important to denote the elements of a fuzzy set that are contained to a degree that is higher than 0. The *support* of a fuzzy set is the set of elements that have this property.

Definition 2.19. Consider an $\mathscr{L}$-fuzzy set A in a universe X. The **support** of A is the set $supp(A)$ that is defined by $supp(A) = \{x \mid x \in X, A(x) > 0\}$.

2.3.2 *Logical Operators on Bounded Lattices*

In this section we recall the generalizations of classical logical operators in fuzzy logic.

Negators, Triangular Norms and Triangular Conorms

The negation $\neg$ of classical logic can be generalized as follows:

Definition 2.20. A **negator** $\mathscr{N}$ on a bounded lattice $\mathscr{L}$ is a decreasing $\mathscr{L} \to \mathscr{L}$ mapping that satisfies $\mathscr{N}(0) = 1$ and $\mathscr{N}(1) = 0$. The negator $\mathscr{N}$ is called **involutive** iff for each $x \in \mathscr{L}$ we have $\mathscr{N}(\mathscr{N}(x)) = x$.

Note that if we take $True = 1$ and $False = 0$, the boundary conditions $\mathscr{N}(0) = 1$ and $\mathscr{N}(1) = 0$ ensure that any negator behaves as the negation $\neg$ of classical logic on the lattice $(\{0,1\},\leqslant)$. The generalizations of other logical operators will similarly need certain boundary conditions to ensure that the classical behavior is recovered for the lattice $(\{0,1\},\leqslant)$.

Example 2.13. We introduce two common negators over $([0,1],\leqslant)$, the lattice most commonly associated with fuzzy logic.

(1) The **Gödel negator** $\mathscr{N}_M$ (also known as **drastic negator**) on a bounded lattice $\mathscr{L}$ is the $\mathscr{L} \to \mathscr{L}$ mapping defined as

$$\mathscr{N}_M(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

(2) The **standard negator** $\mathscr{N}_W$ (also known as **Łukasiewicz negator**) is the $[0,1] \to [0,1]$ mapping defined as $\mathscr{N}_W(x) = 1 - x$. Note that this negator is involutive.

Conjunction is usually generalized by *t-norms* and disjunction by *t-conorms*.

Definition 2.21 (from [Klement *et al.* (2002)]). *A **triangular norm** $\mathscr{T}$ (short: **t-norm**) on a bounded lattice $\mathscr{L}$ is an increasing, associative and commutative $\mathscr{L}^2 \to \mathscr{L}$ mapping that satisfies the boundary condition $\mathscr{T}(1,x) = x$ for any $x \in \mathscr{L}$.*

Definition 2.22 (from [Klement *et al.* (2002)]). *A **triangular conorm** $\mathscr{S}$ (short: **t-conorm**) on a bounded lattice $\mathscr{L}$ is an increasing, associative and commutative $\mathscr{L}^2 \to \mathscr{L}$ mapping that satisfies the boundary condition $\mathscr{S}(0,x) = x$ for any $x \in \mathscr{L}$.*

Hence t-norms and t-conorms only differ in their boundary conditions. Due to the associativity and commutativity we can extend them to an arbitrary number of arguments, i.e. $\mathscr{T}(x_1,\ldots,x_n) = \mathscr{T}(x_1,\mathscr{T}(x_2,\mathscr{T}(\ldots,x_n)))$ and likewise $\mathscr{S}(x_1,\ldots,x_n) = \mathscr{S}(x_1,\mathscr{S}(x_2,\mathscr{S}(\ldots,x_n)))$. From the above definitions we also obtain two other boundary conditions: $\mathscr{T}(0,x) = 0$ and $\mathscr{S}(1,x) = 1$ for any $x \in \mathscr{L}$.

Example 2.14 (from [De Cooman and Kerre (1994)]). *The following two t-norms and t-conorms are well-known.*

(1) Consider a bounded lattice $\mathscr{L} = (L, \leqslant)$. One can immediately see that $\sqcap$ is a triangular norm on $\mathscr{L}$, which we will denote as $\mathscr{T}_M$. Likewise $\sqcup$ is a triangular t-conorm on $\mathscr{L}$ which we will denote as $\mathscr{S}_M$.

*(2) Consider a bounded lattice $\mathscr{L} = (L, \leqslant)$. The **drastic t-norm** $\mathscr{T}_Z$ is a $\mathscr{L}^2 \to \mathscr{L}$ mapping defined by*

$$\mathscr{T}_Z(x,y) = \begin{cases} x & \textit{if } y = 1 \\ y & \textit{if } x = 1 \\ 0 & \textit{otherwise} \end{cases}$$

t-norm	**t-conorm**
$\mathscr{T}_M(x,y) = \min(x,y)$	$\mathscr{S}_M(x,y) = \max(x,y)$
$\mathscr{T}_W(x,y) = \max(0,x+y-1)$	$\mathscr{S}_W(x,y) = \min(1,x+y)$
$\mathscr{T}_P(x,y) = x \cdot y$	$\mathscr{S}_P(x,y) = x+y-x \cdot y$

Table 2.1: T-norms and t-conorms on $([0,1],\leqslant)$

Likewise we can define the ***drastic t-conorm*** $\mathscr{S}_Z$ *as the following* $\mathscr{L}^2 \to \mathscr{L}$

$$\mathscr{S}_Z(x,y) = \begin{cases} x & \text{if } y = 0 \\ y & \text{if } x = 0 \\ 1 & \text{otherwise} \end{cases}$$

The above example shows that for any bounded lattice $\mathscr{L}$ we can construct at least two t-norms and t-conorms. It is interesting to note that any other t-norm (t-conorm) that can be constructed on $\mathscr{L}$ must necessarily be in between the drastic and minimum (maximum) t-norms (t-conorms).

Proposition 2.7 (from [De Cooman and Kerre (1994)]). *For any t-norm* $\mathscr{T}$ *and t-conorm* $\mathscr{S}$ *on a bounded lattice* $\mathscr{L} = (L,\leqslant)$ *we have that for any* $x,y \in \mathscr{L}$.

$$\mathscr{T}_Z(x,y) \leqslant \mathscr{T}(x,y) \leqslant \mathscr{T}_M(x,y)$$

$$\mathscr{S}_M(x,y) \leqslant \mathscr{S}(x,y) \leqslant \mathscr{S}_Z(x,y)$$

It is well-known that in classical logic $\wedge$ and $\vee$ satisfy the De Morgan properties, hence they are often called *dual*. The following definition generalizes this property to arbitrary negators and binary functions on a bounded lattice.

Definition 2.23. Let $\mathscr{N}$ be a negator on a bounded lattice $\mathscr{L}$. The **dual image** of a $\mathscr{L}^2 \to \mathscr{L}$ mapping f w.r.t. $\mathscr{N}$ is the $\mathscr{L}^2 \to \mathscr{L}$ mapping $f^{\leftrightarrow_{\mathscr{N}}}$ defined as

$$f^{\leftrightarrow_{\mathscr{N}}}(x,y) = \mathscr{N}(f(\mathscr{N}(x),\mathscr{N}(y)))$$

It turns out that if $\mathscr{N}$ is involutive the dual of a t-norm (resp. t-conorm) is a t-conorm (resp. t-norm) and vice versa [De Cooman and Kerre (1994)].

Example 2.15. In Table 2.1 we have defined the well known t-norms $\mathscr{T}_M$, $\mathscr{T}_W$ and $\mathscr{T}_P$ on the complete lattice $([0,1],\leqslant)$. They are respectively called the **minimum t-norm**, **Łukasiewicz t-norm** and **product t-norm**. The t-conorms $\mathscr{S}_M$, $\mathscr{S}_W$ and $\mathscr{S}_P$ defined in

this same table are respectively called the **maximum t-conorm**, **Łukasiewicz t-conorm** (also known as the **bounded sum**) and **product t-conorm** (also known as the **probabilistic sum**).
Each t-norm is dual with the t-conorm on the same line, w.r.t. the standard negator $\mathscr{N}_W$. Specifically we have that $\mathscr{T}_M^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{S}_M$, $\mathscr{T}_W^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{S}_W$ and $\mathscr{T}_P^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{S}_P$ and $\mathscr{S}_M^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{T}_M$, $\mathscr{S}_W^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{T}_W$ and $\mathscr{S}_P^{\leftrightarrow_{\mathscr{N}_W}} = \mathscr{T}_P$.

Note that different t-norms have different properties. For example, consider the t-norms introduced in Example 2.15 on the preceding page. The minimum t-norm $\mathscr{T}_M$ is the only t-norm that satisfies idempotency (i.e. $\mathscr{T}_M(x,x) = x$ for every x [Klement *et al.* (2002)]), whereas the Łukasiewicz t-norm $\mathscr{T}_W$ is the only t-norm in Table 2.1 that satisfies the law of contradiction[1] w.r.t. the standard negator $\mathscr{N}_W$. Which t-norm to use is thus greatly dependent on the application and especially on the properties of classical conjunction that are important in the problem at hand. A thorough discussion of the logical properties that remain valid for the three t-norms $\mathscr{T}_M$, $\mathscr{T}_W$ and $\mathscr{T}_P$ can be found in [Kerre (1993)].

Implicators

Definition 2.24. An **implicator** $\mathscr{I}$ on a bounded lattice $\mathscr{L}$ is a $\mathscr{L}^2 \to \mathscr{L}$ mapping that is increasing in its first partial mapping and decreasing in its second partial mapping and furthermore satisfies the boundary conditions $\mathscr{I}(0,0) = 0$ and $\mathscr{I}(1,x) = x$ for each $x \in \mathscr{L}$.

It turns out that any implicator $\mathscr{I}$ on a bounded lattice $\mathscr{L}$ also satisfies the boundary conditions $\mathscr{I}(0,x) = 0$ and $\mathscr{I}(x,1) = 1$, for any $x \in \mathscr{L}$ [De Cock (2002)].
Now the question arises how implicators can be constructed. Since we already constructed t-conorms and negators, a natural idea is to start from a generalization of the classical logic tautology $p \Rightarrow q \equiv \neg p \vee q$.

Definition 2.25. Let $\mathscr{L}$ be a bounded lattice, let $\mathscr{S}$ be a t-conorm on $\mathscr{L}$ and let $\mathscr{N}$ be a negator on $\mathscr{L}$. The $\mathscr{L}^2 \to \mathscr{L}$ mapping $\mathscr{I}_{\mathscr{S},\mathscr{N}}$ defined as $\mathscr{I}_{\mathscr{S},\mathscr{N}}(x,y) = \mathscr{S}(\mathscr{N}(x),y)$ is called the **S-implicator** induced by $\mathscr{S}$ and $\mathscr{N}$.

Using an involutive negator and the dual of the t-conorm we can also define S-implicators w.r.t. a t-norm and a negator.

Definition 2.26. Let $\mathscr{L}$ be a bounded lattice, let $\mathscr{T}$ be a t-norm on $\mathscr{L}$ and let $\mathscr{N}$ be an involutive negator on $\mathscr{L}$. The $\mathscr{L}^2 \to \mathscr{L}$ mapping $\mathscr{I}_{\mathscr{T},\mathscr{N}}$ defined as $\mathscr{I}_{\mathscr{T},\mathscr{N}}(x,y) =$

[1] In classical logic the law of contradiction states that $p \wedge \neg p = \mathit{False}$.

S-implicator	Residual implicator
$\mathscr{I}_{\mathscr{T}_M,\mathscr{N}_W}(x,y) = \max(1-x,y)$	$\mathscr{I}_M(x,y) = \begin{cases} 1 & \text{if } x \leqslant y \\ y & \text{otherwise} \end{cases}$
$\mathscr{I}_{\mathscr{T}_W,\mathscr{N}_W}(x,y) = \min(1-x+y,1)$	$\mathscr{I}_W(x,y) = \min(1-x+y,1)$
$\mathscr{I}_{\mathscr{T}_P,\mathscr{N}_W}(x,y) = 1-x+x\cdot y$	$\mathscr{I}_P(x,y) = \begin{cases} 1 & \text{if } x \leqslant y \\ \frac{y}{x} & \text{otherwise} \end{cases}$

Table 2.2: Implicators on $([0,1],\leqslant)$

$\mathscr{N}(\mathscr{T}(x,\mathscr{N}(y)))$ is called the **S-implicator** induced by $\mathscr{T}$ and $\mathscr{N}$.

Example 2.16. In the first column of Table 2.2 one can find the S-implicators on $[0,1]$ that are induced by the t-norms from Table 2.1 and the involutive negator $\mathscr{N}_W$. The implicator $\mathscr{I}_{\mathscr{T}_M,\mathscr{N}_W}$ is called the **Kleene-Dienes implicator**, $\mathscr{I}_{\mathscr{T}_W,\mathscr{N}_W}$ the **Łukasiewicz implicator** and $\mathscr{I}_{\mathscr{T}_P,\mathscr{N}_W}$ the **Reichenbach implicator**.

While S-implicators are useful, they do not preserve some important properties of classical implication such as modus ponens (i.e. $p \wedge (p \Rightarrow q) \Rightarrow q$), transitivity (i.e. $(p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r)$ and shunting (i.e. $p \Rightarrow (q \Rightarrow r) \equiv p \wedge q \Rightarrow r$). For example, generalizing the modus ponens formula $p \wedge (p \Rightarrow q) \Rightarrow q$ can be done by stating that for any $x,y \in [0,1]$ we must have that $\mathscr{T}(x,\mathscr{I}(x,y)) \leqslant y$. However, one can easily see that some S-implicators violate this requirement:

$$\mathscr{T}_M(0.5,\mathscr{I}_{\mathscr{T}_M,\mathscr{N}_W}(0.5,0.3)) = 0.5 > 0.3$$

It turns out that for certain t-norms one can construct implicators that do satisfy these properties.

Definition 2.27. Let $\mathscr{L}$ be a complete lattice and let $\mathscr{T}$ be a t-norm on $\mathscr{L}$. The **residual implicator** (short: **R-implicator**) of $\mathscr{T}$ is the $\mathscr{L}^2 \to \mathscr{L}$ mapping

$$\mathscr{I}_{\mathscr{T}}(x,y) = \sup\{\lambda \mid \lambda \in \mathscr{L} \wedge \mathscr{T}(x,\lambda) \leqslant y\}$$

The following proposition shows that R-implicators are indeed implicators as defined by Definition 2.24.

Proposition 2.8 (from [De Cock (2002)]). *Let $\mathscr{L}$ be a complete lattice and let $\mathscr{T}$ be a t-norm on $\mathscr{L}$. Then the residual implicator of $\mathscr{T}$ is an implicator on $\mathscr{L}$.*

For a specific class of t-norms we also obtain the following important property.

Proposition 2.9 (from [De Cock (2002)]). *Let $\mathscr{L}$ be a complete lattice and let $\mathscr{T}$ be a t-norm on $\mathscr{L}$. If for each $\lambda \in \mathscr{L}$ and family $(x_i)_{i\in I}$ we have that $\mathscr{T}(\sup_{i\in I} x_i, \lambda) = \sup_{i\in I} \mathscr{T}(x_i, \lambda)$, i.e. all partial mappings of $\mathscr{T}$ are supmorphisms, it holds that*

$$\mathscr{T}(x,y) \leqslant z \text{ iff } x \leqslant \mathscr{I}(y,z)$$

*for all $x,y,z \in \mathscr{L}$. This property is called the **residuation principle**.*

The residuation principle is also commonly referred to as the **Galois connection** or **adjoint property**. In [De Baets (1995)] it is shown that for a t-norm $\mathscr{T}$ that satisfies the condition in Proposition 2.9, we have that $\mathscr{T}(x, \mathscr{I}_{\mathscr{T}}(y,z)) \leqslant y$ for each $x,y,z \in \mathscr{L}$, i.e. the generalization of the modus ponens, introduced above, holds. For this reason we will limit our attention to t-norms satisfying this condition in the remainder of this book.

Example 2.17. In the second column of Table 2.2 we listed some common residual implicators on $([0,1],\leqslant)$. The implicator $\mathscr{I}_M$ is called the **Gödel implicator**, $\mathscr{I}_W$ the **Łukasiewicz implicator**, $\mathscr{I}_P$ the **Goguen implicator**. Note that the Łukasiewicz implicator is both a residual implicator and an S-implicator.

An important property of residual implicators is the following.

Proposition 2.10 (from [De Baets (1995)]). *Let $\mathscr{I}$ be a residual implicator on $\mathscr{L}$. For any $x,y \in \mathscr{L}$ it holds that $\mathscr{I}(x,y) = 1$ iff $x \leqslant y$.*

In classical logic it is well-known that $a \wedge (a \Rightarrow b) \equiv a \wedge b$. For t-norms and implicators this does not hold in general, but for specific t-norms on $([0,1],\leqslant)$ a similar property can be shown to hold.

Proposition 2.11 (from [Klement *et al.* (2002)]). *Let $\mathscr{T}$ be a continuous t-norm on $([0,1],\leqslant)$. Then for any $x,y \in [0,1]$ it holds that $\mathscr{T}(x, \mathscr{I}_{\mathscr{T}}(x,y)) = \min(x,y)$. This property is called **divisibility**.*

Usually the equivalence $a \equiv b$ in classical logic is defined as $((a \Rightarrow b) \wedge (b \Rightarrow a))$. A similar concept can be defined in fuzzy logic using a residual implicator and t-norm.

Definition 2.28. Consider a residual implicator $\mathscr{I}$ and t-norm $\mathscr{T}$ on $\mathscr{L}$. The **biresiduum** of $\mathscr{I}$ and $\mathscr{T}$ is the $\mathscr{L}^2 \to \mathscr{L}$ mapping $\approx$ defined for all $x,y \in \mathscr{L}$ as:

$$x \approx y = \mathscr{T}(\mathscr{I}(x,y), \mathscr{I}(y,x))$$

Finally, one can show that for any implicator $\mathscr{I}$ on a bounded lattice $\mathscr{L}$, the partial mapping $\mathscr{I}(.,0)$ is a negator on $\mathscr{L}$.

Definition 2.29. Let $\mathscr{L}$ be a bounded lattice and let $\mathscr{I}$ be an implicator on $\mathscr{L}$. The **induced negator** of $\mathscr{I}$ is then the $\mathscr{L} \to \mathscr{L}$ mapping $\mathscr{N}_{\mathscr{I}}$ defined as $\mathscr{I}(x) = \mathscr{I}(x,0)$, for each $x \in \mathscr{L}$.

Example 2.18. The Gödel negator $\mathscr{N}_M$ is the induced negator of the Gödel implicator; the Łukasiewicz negator $\mathscr{N}_W$ is the induced negator of the Łukasiewicz implicator.

2.3.3 *Formal Fuzzy Logics*

In [Hájek (1998)] it is shown that for the minimum t-norm, Łukasiewicz t-norm and product t-norm a propositional calculus capable of describing their properties can be constructed. The author of [Hájek (1998)] furthermore identifies a propositional calculus that captures the properties shared by all continuous t-norms on $([0,1],\leqslant)$. We briefly discuss this in this section.

Definition 2.30 (from [Hájek (1998)]). *The **propositional calculus** $\mathbf{PC}(\mathscr{T})$ given by the continuous t-norm $\mathscr{T}$ on $([0,1],\leqslant)$ has propositional variables $p_1,\ldots,p_n$, the connectives $\wedge$ and $\to$ and the truth constant $\overline{0}$ for 0. **Formulas** are defined as usual: each propositional variable is a formula; $\overline{0}$ is a formula; if φ,ψ, are formulas, then $\varphi \wedge \psi$ and $\varphi \to \psi$ are formulas. Further connectives are defined as follows:*

(1) $\varphi \triangle \psi = \varphi \wedge (\varphi \to \psi)$
(2) $\varphi \triangledown \psi = ((\varphi \to \psi) \to \psi) \wedge ((\psi \to \varphi) \to \varphi)$
(3) $\neg\varphi = \varphi \to \overline{0}$
(4) $\varphi \leftrightarrow \psi = (\varphi \to \psi) \wedge (\psi \to \varphi)$

*An **evaluation** of propositional variables is a mapping e assigning to each propositional variable p a truth value $e(p) \in [0,1]$. This evaluation is extended to formulas as follows:*

(1) $e(\overline{0}) = 0$
(2) $e(\varphi \to \psi) = \mathscr{I}_{\mathscr{T}}(e(\varphi), e(\psi))$
(3) $e(\varphi \wedge \psi) = \mathscr{T}(e(\varphi), e(\psi))$

*A formula φ is a 1-**tautology** of $PC(\mathscr{T})$ iff $e(\varphi) = 1$ for each evaluation e. A set of formulas is called a **theory**. Given a theory Θ we say that an evaluation e is a **model** of Θ, denoted $e \models \Theta$, if and only if $e(\theta) = 1$ for each $\theta \in \Theta$.*

One can see that 1-tautologies are formulas that are absolutely true under any evaluation. Furthermore note that from Proposition 2.11 we know that the definition of $\triangle$ corresponds to the minimum, which is one of the reasons why the calculus is restricted to continuous t-norms. Now a logic can be constructed that captures the properties that are common to all continuous t-norms on $([0,1],\leqslant)$.

Definition 2.31 (from [Hájek (1998)]). *The following formulas are axioms of the* ***basic logic*** **BL***:*

(A1) $(\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))$
(A2) $(\varphi \wedge \psi) \to \varphi$
(A3) $(\varphi \wedge \psi) \to (\psi \wedge \varphi)$
(A4) $(\varphi \wedge (\varphi \to \psi)) \to (\psi \wedge (\psi \to \varphi))$
(A5) $(\varphi \to (\psi \to \chi)) \to ((\varphi \wedge \psi) \to \chi)$
(A6) $((\varphi \wedge \psi) \to \chi) \to (\varphi \to (\psi \to \chi))$
(A7) $((\varphi \to \psi) \to \chi) \to (((\psi \to \varphi) \to \chi) \to \chi)$
(A8) $\overline{0} \to \varphi$

The deduction rule of **BL** *is modus ponens. Proof and a provable formula in* **BL** *are defined similar to proof, respectively provable formula in classical logic.*

The above axioms all have intuitive meanings and correspond to properties in classical logic. Axiom 1 is transitivity of implication. Axiom 2 is sometimes called *weakening* of conjunction in classical logic. Axioms 3 and 4 express the commutativity of $\wedge$, respectively $\triangle$. Axioms 5 and 6 are commonly called the *shunting* rules in classical logic. Axiom 7 is a variant of proof by cases: if χ follows from $\varphi \to \psi$ then if χ also follows from $\psi \to \varphi$ we have χ is true. Finally Axiom 8 states that anything can be proven from a false proposition. The axioms of **BL** are all 1-tautologies in each $\mathbf{PC}(\mathscr{T})$, where $\mathscr{T}$ is a continuous t-norm on $([0,1],\leqslant)$. Hence these properties are true for all continuous t-norms on $([0,1],\leqslant)$. It can be shown that **BL** is *sound*, i.e. that each provable formula in **BL** is a 1-tautology in each $\mathbf{PC}(\mathscr{T})$. Two types of completeness are distinguished: *standard completeness* and *general completeness*. The former states that every 1-tautology in each $\mathbf{PC}(\mathscr{T})$ can be proven in **BL** and has been shown to hold in [Cignoli *et al.* (2000)]. The latter defines completeness with respect to a more general semantics for **BL**, called BL-algebras. The general completeness theorem states that a formula φ is provable in **BL** if it is a general BL-tautology, i.e. a tautology for each BL-algebra [Hájek (1998)]. We can thus conclude

t-norm	logic name	extra axiom(s)
$\mathscr{T}_M$	Gödel logic	$\varphi \to (\varphi \wedge \varphi)$
$\mathscr{T}_W$	Łukasiewicz logic	$\neg\neg\varphi \to \varphi$
$\mathscr{T}_P$	product logic	$\neg\neg\chi \to ((\varphi \wedge \chi \to \psi \wedge \chi) \to (\varphi \to \psi))$
		$\varphi\triangle\neg\varphi \to \overline{0}$

Table 2.3: Propositional logics for the common continuous t-norms on $([0,1],\leqslant)$ (from [Hájek (1998)])

that the logic **BL** captures all properties that are common to the continuous t-norms on $([0,1],\leqslant)$.

If we now consider specific t-norms, it is also possible to construct a logic that exactly captures the 1-tautologies for this specific t-norm. Note that these logics can be characterized by extending **BL** with certain axioms. In Table 2.3 we illustrate the logics one obtains by considering the minimum, Łukasiewicz and product t-norm, as well as the axiom(s) that need to be added to **BL** to create this logic.

An interesting extension of Łukasiewicz logic is **Rational Pavelka Logic** (short: **RPL**). The language of RPL is constructed by extending the language of $\mathbf{PC}(\mathscr{T}_W)$ with truth constants $\overline{r}$ for each *rational number* $r \in [0,1]\cap\mathbb{Q}$. The formulas in this logic are defined as for $\mathbf{PC}(\mathscr{T}_W)$, where we also consider the truth constants as formulas. Evaluations e are extended such that for any rational $r \in [0,1]\cap\mathbb{Q}$ we have that $e(\overline{r}) = r$. The axioms of RPL are the axioms of Łukasiewicz logic plus two bookkeeping axioms for the truth constants: $(\overline{r} \to \overline{s}) \leftrightarrow \overline{\mathscr{I}_W(r,s)}$ and $\neg\overline{r} \leftrightarrow \overline{1-r}$ for all r and s in $[0,1]\cap\mathbb{Q}$. Hence the set of axioms is countable, but not finite. The deduction rule, theories, proofs, provability and models are defined as for Łukasiewicz logic.

Finally, reasoning in the above logics can be done using existing methods. For Gödel logic we can use boolean SAT solvers, for Łukasiewicz and rational Pavelka logic we can use mixed integer programming (MIP) [Hähnle (1994)] or constraint satisfaction [Schockaert *et al.* (2009)] and for product logic we can use bounded mixed integer quadratically constrained programming (bMICQP), as is used for fuzzy description logics [Bobillo and Straccia (2007)]. Similar to the boolean case, checking whether a set of formulas Θ is **satisfiable**, i.e. whether some model exists for Θ, is NP-complete [Hájek (1998)].

http://www.springer.com/978-94-91216-58-9

Answer Set Programming for Continuous Domains: A Fuzzy Logic Approach

Janssen, J.; Schockaert, S.; Vermeir, D.; De Cock, M.

2012, X, 174 p., Hardcover
ISBN: 978-94-91216-58-9
A product of Atlantis Press