Implementing Belnap-Logical Conflation and Implication Operators in Answer Set Programming

by

Zhao Yi Han

B.Sc., University of British Columbia, 2016

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Zhao Yi Han 2020
SIMON FRASER UNIVERSITY
Spring 2020

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name: **Zhao Yi Han**

Degree: **Master of Science (Computing Science)**

Title: **Implementing Belnap-Logical Conflation and Implication Operators in Answer Set Programming**

Examining Committee: **Chair:** David Mitchell
Associate Professor

James Delgrande
Senior Supervisor
Professor

Fred Popowich
Supervisor
Professor

Aaron Hunter
External Examiner
Instructor
School of Computing
British Columbia Institute of Technology

Date Defended: **March 19, 2020**

Abstract

Two types of negation are allowed in answer set programming (ASP), default negation and classical negation. When using two-valued logic as its basis, the presence of classical negation in ASP can lead to gluts (both true and false) and gaps (neither true nor false), which are handled in unintuitive ways. Belnap's four-valued logic, with gluts and gaps as truth values, is a more intuitive basis for ASP. This thesis examines the intuition behind Belnap logic, showing that the conflation operator, which has no obvious intuitive meaning, is central to the representation of default negation in Belnap logic. There is no single correct implication operator in Belnap logic that can be used in ASP rules, so we examine a number of different implication operators in Belnap logic, before presenting a new implication operator that generalizes them and showing how this implication operator can be implemented in ASP without changing its specifications.

Keywords: Logic programming; answer set programming; paraconsistent logic; Belnap logic

Table of Contents

Approval **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**

2 Belnap's Four-Valued Logic **4**

2.1 Truth, Falsity, and Truth Values 4

2.2 Operators in Belnap Logic 7

3 Default Negation in Four-Valued Answer Set Programming **14**

3.1 Logic Programming and Stable Model Semantics 14

3.2 Default Negation in Four-Valued ASP 20

4 Implications in Four-Valued Answer Set Programming **23**

4.1 The Basic Implication Operator 23

4.2 Other Implication Operators in Belnap Logic 26

4.3 General Unidirectional Implication Operators in Belnap Logic 31

4.4 Implementation of General Unidirectional Implications and Contraposition in ASP 36

5 Conclusion and Further Research **44**

Bibliography **46**

List of Tables

Table 2.1 The symbols and names for the infimum, supremum, and inversion operators of the truth and information partial orderings in Belnap logic. 8
Table 2.2 The truth tables of conjunction and its decomposition. 9
Table 2.3 The truth tables of disjunction and its decomposition. 10
Table 2.4 The truth tables of negation and its decomposition. 10
Table 2.5 The truth tables of consensus and its decomposition. 11
Table 2.6 The truth tables of gullibility and its decomposition. 11
Table 2.7 The truth tables of conflation and its decomposition. 11

Table 3.1 The truth tables of default negation and its decomposition. 21

Table 4.1 The truth tables of $\rightarrow_t$ and its decomposition. 26
Table 4.2 The truth tables of $\rightarrow_{cmi}$ and its decomposition. 27
Table 4.3 The truth tables of $\rightarrow_{le}$ and its decomposition. 29
Table 4.4 The truth tables of $\rightarrow_{si}$ and its decomposition. 30

List of Figures

Figure 2.1 A double Hasse diagram of **4** and its two partial orderings. 7

Chapter 1

Introduction

Logic programming is a highly robust field today, useful in artificial intelligence and many other fields where the domain can be represented by a set of yes/no values. One of its main advantages is that it satisfies the knowledge representation hypothesis, where the logic program contains symbols that transparently and explicitly represent the knowledge that it reasons about, compared to approaches like neural networks that are more opaque and "black box"-like. Most forms of logic programming use classical (propositional or first-order) logic as a base, and are effectively classical logic with specific constraints that limit its expressiveness (mainly based on Horn clauses by Horn [12]), because fully general classical logic can be computationally intractable (exponential or worse) to solve, whereas more restricted logic programming is guaranteed to be solvable in more reasonable amounts of time (such as NP and co-NP); the computational complexity of logic programming is beyond the scope of this thesis and can instead be found in works like Faber and Leone [5]. Some forms of logic programming are also non-monotonic, meaning that they are able to draw tentative conclusions that can be later retracted upon obtaining some counter-evidence, a capability that classical logic itself lacks.

Classical logic suffers from the principle of explosion, where it becomes possible to prove every single possible logical formula if the premises are inconsistent; a single contradiction causes the entire domain to collapse into incoherent triviality and uselessness. This is especially problematic because it is often not possible to tell whether a set of premises is inconsistent, as they may not appear to directly contradict one another at the first glance, but can have further logical consequences that turn out to be inconsistent with each other. The premises could have long since been integrated into a knowledge base and used to make conclusions, without knowing that all conclusions made after the inclusion of said premises are meaningless due to the premises being secretly inconsistent. Furthermore, the knowledge base may accept input from multiple different sources, necessitating mechanisms to handle inconsistencies one way or another. Paraconsistent logic is a field of logic that attempts to

avoid the principle of explosion, allowing the logic to prove certain contradictions without necessitating that everything else in the domain collapses into triviality; an overview of this field can be found in Priest [16]. Since logic programming is usually based on classical logic, it usually also suffers from the principle of explosion, so it is beneficial to have a form of logic programming that uses some paraconsistent logic as a basis instead of classical logic. Ideally, a contradiction should be detected and potentially given some specific treatment, but it should not be able to affect other parts of the program that are not related to the contradiction itself.

Answer set programming (ASP) is a non-monotonic logic programming paradigm, based on stable model semantics originally proposed by Gelfond and Lifschitz [8], widely used in many fields today, some of which are summarized by Falkner et al. [6]. The most basic version of ASP contains default negation as a major feature, representing the lack of knowledge or information on a subject. Most versions of ASP are usually extended by the ability to encode classical negation separately from default negation [9], representing explicit knowledge that something is false, as opposed to the lack of knowledge that something is true. The original formulation of ASP with classical negation was intended to agree with classical logic as much as possible, such that the principle of explosion is explicitly enforced by a separate clause in every program. But aside from this explicit explosion clause, there is in fact nothing else in ASP that causes explosive behavior even if a knowledge base contains both a formula and its classical negation, due to the specific, limited way in which it can draw conclusions. And in the absence of additional clauses to restrict this kind of behavior, it is also possible for ASP to derive nothing at all about a formula, neither its explicit truth nor its explicit falsity, another departure from classical logic where something must be either true or false. The potential existence of gluts (both true and false) and gaps (neither true nor false) in ASP suggests that it has much in common with Belnap's four-valued logic [2], which will be referred to as "Belnap logic" for the remainder of this thesis, a paraconsistent logic where the glut and the gap are both truth values in their own right in addition to true and false; it is used in some fields today, like an IT access control system by Bruns and Huth [3]. In Belnap logic, the truth and falsity of any formula can be handled separately [4], which bears great similarity to how ASP handles classical negation as well. A major part of this thesis will be examining Belnap logic in detail and showing how it is an intuitive basis to be used for ASP, with emphasis on the qualitative meanings of certain important nuances that, from my knowledge, receive relatively little attention from other works involving Belnap logic.

This thesis will be structured as follows:

- Chapter 2 introduces Belnap logic and examines it in detail, focusing on the qualitative meanings of truth and falsity in knowledge representation, and the relations those

meanings have with the various operators in Belnap logic. It ends with remarks about the lack of any obvious intuitive meaning for the "conflation" operator in Belnap logic, and the lack of a single perfect Belnap logic generalization of classical two-valued implication [11], two points that are central to the new findings of this thesis, presented in the next two chapters.

- Chapter 3 introduces ASP, defining program rules, models, and stable models. Two forms of negation in logic programming, default negation and classical negation, are introduced. And the first of the two findings in this thesis will be presented, that the conflation operator in Belnap logic is closely related to the concept of default negation.
- Chapter 4 focuses on the implication operator that is the backbone of every program rule, and the second major finding of my thesis is presented. Past attempts of combing ASP with Belnap logic exist, like Chen and Lin in 2016 [4], but they do not focus on what type of implication in Belnap logic that is to be used, whereas we show how several different types of implication operators in Belnap logic can all be used in tandem in ASP. We also present a new generalized implication operator that subsumes all implication operators examined in this chapter, and show how this generalized implication can be easily implemented in ASP.
- Chapter 5 summarizes the findings and proposes some potential directions for further research.

Before we begin, we should note that despite the focus on Belnap logic, the words "if", "then", "iff", "and", "or", and "not" in this thesis are to be interpreted as per classical logic. Whenever these words themselves are used, they apply to binary mathematical conditions that are either satisfied or not satisfied, where paraconsistency is not necessary or relevant, as opposed to actual Belnap logical constructs that will be denoted with symbols like $\wedge$ and $\rightarrow$. This is important as some of the conditions in this thesis are more intuitively defined via constructing their complements, such as "P is not satisfied iff Q is satisfied", where Q is a condition that has already been defined, such that P is satisfied iff Q is not satisfied, exactly like how biconditionals behave in classical logic.

Additionally, a distinction is made between what this thesis refers to as "propositions" versus "theorems". A proposition is a truth obtained from previous works in the field, while a theorem is one of the new findings of this thesis.

Chapter 2

Belnap's Four-Valued Logic

Unless stated otherwise, all information on Belnap logic in this chapter can be assumed to be from Belnap [2].

2.1 Truth, Falsity, and Truth Values

Belnap's four-valued logic centers around the set of four truth values $\mathbf{4} = \{\mathbf{T}, \mathbf{F}, \mathbf{B}, \mathbf{N}\}$. These truth values can be thought of as subsets of the set of classical truth values $\mathbf{2} = \{\mathbf{t}, \mathbf{f}\}$, where $\mathbf{T} = \{\mathbf{t}\}$, $\mathbf{F} = \{\mathbf{f}\}$, $\mathbf{B} = \{\mathbf{t}, \mathbf{f}\}$, and $\mathbf{N} = \emptyset$; their meanings can be thought of respectively as "true and not false", "false and not true", "both true and false", and "neither true nor false". Immediately, we notice that the words "true" and "false" are used here as if they are independent qualities, as opposed to negations of each other. All constructs in Belnap logic are defined in terms of the qualities of "is true" and "is false".

Definition 1. A *literal* is a positive literal or a negative literal. A *positive literal* is an atom a, and a *negative literal* is an atom's classical negation $\neg a$. An *interpretation* I is a set of literals. A literal l is:

- *true* in I iff $l \in I$;
- *false* in I iff $\neg l \in I$.

Again, we notice that truth and falsity are completely independent conditions; being true is not the same as being not false, and being false is not the same as being not true. There is nothing stating that an interpretation cannot contain both, or neither, of an atom and its negation. It is possible to be simultaneously true and false, but it is not possible to be simultaneously true and not true, since it is not possible for I to simultaneously contain and not contain a; it is similarly impossible to be simultaneously false and not false.

Another thing we emphasize is that, as originally formulated by Belnap, truth and falsity in Belnap logic are *epistemic*, representing the user's *knowledge*. The positive literals in an interpretation I represents the atoms in I that the user "has been told" of as true, and the negative literals being the set of atoms the user "has been told" of as false. A typical use case is that the user receives information from a number of different sources that are each individually trusted to be mostly reliable, but are not guaranteed to be consistent with each other. For example, suppose that the user has two sources, Alice and Bob. Alice tells the user that Microsoft stocks have risen today, while Bob tells the user that Microsoft stocks have *not* risen today; the user then records in her database that she has been told conflicting information about whether Microsoft stocks have risen today [1]. On the other hand, neither Alice nor Bob tells the user anything about Google stocks, so the user's database contains no information on whether Google stocks have risen today. Belnap logic is not concerned with the *ontological*, i.e. "inherent" or "metaphysical" truth or falsity of any logical formula, and therefore is not concerned with whether it is "possible" for a formula to be simultaneously true and false, or neither true nor false; it is only concerned with potentially conflicting or incomplete information about a formula's truth and falsity. Belnap logic is completely independent on the user's views on dialetheism.

Definition 2. Let $\mathbf{2} = \{\mathbf{t}, \mathbf{f}\}$ be the set of classical truth values, and $V_I(a) \subseteq \mathbf{2}$ be the *truth value* of atom a in an interpretation I. Then,

- $\mathbf{t} \in V_I(a)$ iff $a \in I$;
- $\mathbf{f} \in V_I(a)$ iff $\neg a \in I$.

The possible values of $V_I(a)$ are given the following names:

- $\mathbf{T} = \{\mathbf{t}\}$;
- $\mathbf{F} = \{\mathbf{f}\}$;
- $\mathbf{B} = \{\mathbf{t}, \mathbf{f}\}$;
- $\mathbf{N} = \emptyset$.

Definition 3. An *operator* Op is a function of arity n, domain $\mathbf{4}^n$, and codomain $\mathbf{4}$. A $formula$ is of the form $Op(A_1, ..., A_n)$, where each of $A_1, ..., A_n$ is a formula; the most basic formula is a single atom. As syntactic sugar, a unary operator can be written as $Op\ a$, while a binary operator can be written as $a\ Op\ b$. In an interpretation I, $V_I(Op(A_1, ..., A_n)) = Op(V_I(A_1), ..., V_I(A_n))$, evaluated recursively until atoms are evaluated. A formula A is:

[1] Note that nothing precludes a single source from providing conflicting information to the user; it is entirely possible for Alice to tell the user that Microsoft stocks have risen today but also that Microsoft stocks have not risen today.

- *true* in I iff $\mathbf{t} \in V_I(A)$;
- *false* in I iff $\mathbf{f} \in V_I(A)$.

Definitions 2 and 3 form the basis of how Belnap truth values relate to truth, falsity, and classical truth values.

Proposition 1. *For a formula A in interpretation I:*

- $V_I(A) = \mathbf{T}$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{f} \notin V_I(A)$ *(i.e.* $\mathbf{T}$ *is true and not false);*
- $V_I(A) = \mathbf{F}$ *iff* $\mathbf{t} \notin V_I(A)$ *and* $\mathbf{f} \in V_I(A)$ *(i.e.* $\mathbf{F}$ *is not true and false);*
- $V_I(A) = \mathbf{B}$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{f} \in V_I(A)$ *(i.e.* $\mathbf{T}$ *is true and false);*
- $V_I(A) = \mathbf{N}$ *iff* $\mathbf{t} \notin V_I(A)$ *and* $\mathbf{f} \notin V_I(A)$ *(i.e.* $\mathbf{T}$ *is not true and not false);*
- $\mathbf{t} \in V_I(A)$ *iff* $V_I(A) \in \{\mathbf{T}, \mathbf{B}\}$ *(i.e.* $\mathbf{T}$ *and* $\mathbf{B}$ *are true);*
- $\mathbf{f} \in V_I(A)$ *iff* $V_I(A) \in \{\mathbf{F}, \mathbf{B}\}$ *(i.e.* $\mathbf{F}$ *and* $\mathbf{B}$ *are false);*
- $\mathbf{t} \notin V_I(A)$ *iff* $V_I(A) \in \{\mathbf{F}, \mathbf{N}\}$ *(i.e.* $\mathbf{F}$ *and* $\mathbf{N}$ *are not true);*
- $\mathbf{f} \notin V_I(A)$ *iff* $V_I(A) \in \{\mathbf{T}, \mathbf{N}\}$ *(i.e.* $\mathbf{T}$ *and* $\mathbf{N}$ *are true).*

Proposition 1 is evident from Definitions 2 and 3. It provides a complete translation from truth and falsity to truth values, and vice versa.

The Belnap truth values form a *bilattice*, a set with two partial orderings, each of which is equipped with its own infimum and supremum operators [10]. For the purpose of this thesis, we do not need to examine bilattices in general; the only parts of bilattices that are made use of in the findings of this thesis are the two partial orderings of **4** (defined below) and their respective operators (defined later in this chapter).

Definition 4. The set $\{\mathbf{T}, \mathbf{F}, \mathbf{B}, \mathbf{N}\}$ has two partial orderings, $\leq_t$ and $\leq_i$:

- In the *truth ordering* $\leq_t$, $\mathbf{T}$ is the maximal element and $\mathbf{F}$ the minimal element. $\mathbf{B}$ and $\mathbf{N}$ are each greater than $\mathbf{F}$, less than $\mathbf{T}$, and incomparable with each other.
- In the *information ordering* $\leq_i$, $\mathbf{B}$ is the maximal element and $\mathbf{N}$ the minimal element. $\mathbf{T}$ and $\mathbf{F}$ are each greater than $\mathbf{N}$, less than $\mathbf{B}$, and incomparable with each other.

The two partial orderings are illustrated in Figure 2.1, a double Hasse diagram where the horizontal axis represents the truth ordering and the vertical axis represents the information ordering.

The truth ordering is more accurately described as the "more true and less false" ordering. For two atoms a and b, $a \leq_t b$ holds iff neither of the following two conditions hold: 1) a is

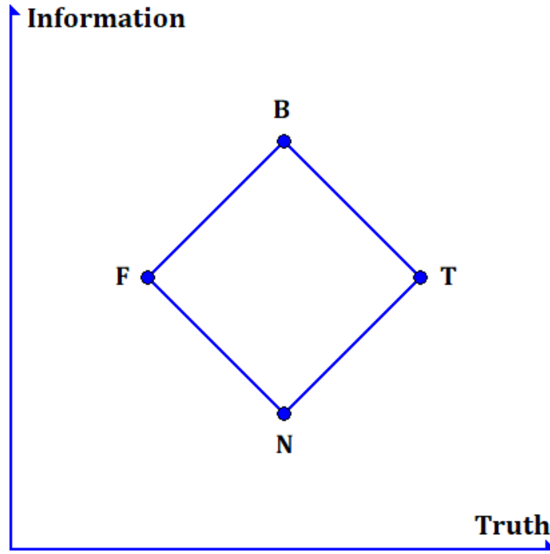

Figure 2.1: A double Hasse diagram of **4** and its two partial orderings.

true and b is not true; 2) a is not false and b is false. $a \leq_t b$ means that a is simultaneously more false or equal to b, and less true or equal to b. Truth and falsity thus cannot be decoupled from each other in this ordering.

The information ordering represents how much information an interpretation has about an atom. The minimal element has no information, the two middle elements have only positive information and only negative information respectively, and the maximal element has both positive and negative information. According to Definition 2 where the truth values are defined as subsets of $\{\mathbf{t}, \mathbf{f}\}$, the information ordering is equivalent to the subset relation; we will also see later that the supremum and infimum operators of the information ordering are equivalent to set union and intersection respectively.

2.2 Operators in Belnap Logic

In Ginsberg's original definition of bilattices [10], he requires that a bilattice has a supremum and infimum operator for each partial ordering, as well as an unary operator that inverts the first ordering, preserves the second ordering, and is its own inverse. For **4**, this operator is classical negation, which inverts the truth ordering while preserving the information ordering. However, we can easily add a second, analogous inversion operator that inverts the information ordering but preserves the truth ordering, and many extensions of Belnap logic do so. Details will be discussed in their relevant sections below, and we will eventually see that this second inversion operator is in fact vital to logic programming using Belnap logic.

The precise mathematical definitions of infimum, supremum, and inversion for bilattices are not relevant to this thesis, and interested readers can instead refer to Ginsberg [10]; here we will simply describe them in informal language. Let a and b be Belnap truth values. Then for either of the two partial orderings:

- If a and b are comparable, their infimum is the lesser (or equal) of the two; otherwise the infimum is the truth value that is less than them both.
- If a and b are comparable, their supremum is the greater (or equal) of the two; otherwise the supremum is the truth value that is greater than them both.
- If a is the maximal element, its inversion is the minimal element, and vice versa. If it is neither maximal nor minimal, its inversion is itself.

	Ordering	**Infimum**	**Supremum**	**Inversion**
Truth	$\leq_t$	$\wedge$ (conjunction)	$\vee$ (disjunction)	$\neg$ (inversion)
Information	$\leq_i$	$\otimes$ (consensus)	$\oplus$ (gullibility)	$-$ (conflation)

Table 2.1: The symbols and names for the infimum, supremum, and inversion operators of the truth and information partial orderings in Belnap logic.

Each operator in Table 2.1 has its own intuitive meaning, which will be covered in the remainder of this chapter. Before getting into those, we take note of some general identities of these operators.

Proposition 2. *For both partial orderings, the infimum, supremum, and inversion operators have the following identities:*

- *The inversion operator is its own inverse. For example,* $-(-A) = A$.
- *The two inversion operators are commutative with each other. For example,* $-\neg A = \neg -A$.
- *Each partial ordering has its own version of De Morgan's laws. For example,* $-(A \oplus B) = -A \otimes -B$.
- *Each negation operator is distributive toward the infimum and supremum operators of the other partial ordering. For example,* $-(A \vee B) = -A \vee -B$.

Proposition 2 can easily be verified by manually checking the truth tables of the formulas involved, using the truth tables provided later in this chapter.

Omori and Sano [15] developed a procedure to decompose any Belnap logical operator into a separate truth condition and falsity condition, possible because truth and falsity are conditions that are independent from each other by Definition 3. A truth condition is a function with domain **4** and codomain $\{\mathbf{T}, \mathbf{N}\}$, a binary condition that can only be true or

not true; similarly for a falsity condition with codomain $\{\mathbf{F}, \mathbf{N}\}$. This procedure splits an operator into two independent functions, one only containing information on truth and the other information on falsity. Unless stated otherwise, all information on decompositions can be assumed to have been taken from Omori and Sano [15].

Definition 5. The *decomposition* of an n-arity operator Op is a tuple $(Op_{\mathbf{t}}, Op_{\mathbf{f}})$, where $Op_{\mathbf{t}}$ is a function with domain $\mathbf{4}$ and codomain $\{\mathbf{T}, \mathbf{N}\}$, and $Op_{\mathbf{f}}$ a function with domain $\mathbf{4}$ and codomain $\{\mathbf{F}, \mathbf{N}\}$. For formulas $A_1, ..., A_n$, their values are as follows:

- If $Op(A_1, ..., A_n) = \mathbf{T}$, then $Op_{\mathbf{t}}(A_1, ..., A_n) = \mathbf{T}$ and $Op_{\mathbf{f}}(A_1, ..., A_n) = \mathbf{N}$.
- If $Op(A_1, ..., A_n) = \mathbf{F}$, then $Op_{\mathbf{t}}(A_1, ..., A_n) = \mathbf{N}$ and $Op_{\mathbf{f}}(A_1, ..., A_n) = \mathbf{F}$.
- If $Op(A_1, ..., A_n) = \mathbf{B}$, then $Op_{\mathbf{t}}(A_1, ..., A_n) = \mathbf{T}$ and $Op_{\mathbf{f}}(A_1, ..., A_n) = \mathbf{F}$.
- If $Op(A_1, ..., A_n) = \mathbf{N}$, then $Op_{\mathbf{t}}(A_1, ..., A_n) = \mathbf{N}$ and $Op_{\mathbf{f}}(A_1, ..., A_n) = \mathbf{N}$.

By this definition, $Op(A_1, ..., A_n) = Op_{\mathbf{t}}(A_1, ..., A_n) \oplus Op_{\mathbf{f}}(A_1, ..., A_n)$.

Definition 5 is a purely mechanical procedure for generating the decomposition of an operator. But as noted by Omori and Sano, for some operators it is possible to simplify their decompositions further by qualitatively examining their truth tables. In fact, such qualitative simplifications exist for all operators relevant to this thesis, as we will show below.

We will now show the decompositions of all the operators in Table 2.1. Blank entries in the truth and falsity condition tables indicate $\mathbf{N}$; they have been left blank to more easily emphasize the conditions that make the original operator true and false.

$\wedge$	**T**	**F**	**B**	**N**	$\wedge_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$\wedge_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**F**	**B**	**N**	**T**	**T**		**T**		**T**		**F**	**F**	
F	**F**	**F**	**F**	**F**	**F**					**F**	**F**	**F**	**F**	**F**
B	**B**	**F**	**B**	**F**	**B**	**T**		**T**		**B**	**F**	**F**	**F**	**F**
N	**N**	**F**	**F**	**N**	**N**					**N**		**F**	**F**	

Table 2.2: The truth tables of conjunction and its decomposition.

Proposition 3. *For formulas A and B in interpretation I,*

- $\mathbf{t} \in V_I(A \wedge B)$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \in V_I(B)$*;*
- $\mathbf{f} \in V_I(A \wedge B)$ *iff* $\mathbf{f} \in V_I(A)$ *or* $\mathbf{f} \in V_I(B)$*.*

Proposition 3, obtainable as a qualitative simplification of the truth and falsity conditions from Table 2.2, is a four-valued generalization of the behavior of conjunction in two-valued logic, where a conjunction is true if both arguments are true, and false if either of the arguments is false. In fact, this was the criteria from which Belnap developed the four-valued truth table of conjunction in the first place, as a generalization of two-valued conjunction.

This is the reason for some of the seemingly unintuitive entries in the truth table, like $\mathbf{B} \wedge \mathbf{N} = \mathbf{F}$; the more useful intuition is to think of conjunction as combining the truths and falsities of its arguments, rather than their truth values directly.

All similar propositions in the remainder of this chapter will be qualitative simplifications of the truth and falsity conditions of the rest of the Belnap operators. Their validity can be verified by checking the appropriate truth tables.

$\vee$	**T**	**F**	**B**	**N**	$\vee_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$\vee_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**	**T**				
F	**T**	**F**	**B**	**N**	**F**	**T**		**T**		**F**		**F**	**F**	
B	**T**	**B**	**B**	**T**	**B**	**T**	**T**	**T**	**T**	**B**		**F**	**F**	
N	**T**	**N**	**T**	**N**	**N**	**T**		**T**		**N**				

Table 2.3: The truth tables of disjunction and its decomposition.

Proposition 4. *For formulas A and B in interpretation I,*

- $\mathbf{t} \in V_I(A \vee B)$ *iff* $\mathbf{t} \in V_I(A)$ *or* $\mathbf{t} \in V_I(B)$*;*
- $\mathbf{f} \in V_I(A \vee B)$ *iff* $\mathbf{f} \in V_I(A)$ *and* $\mathbf{f} \in V_I(B)$*.*

Similarly to conjunction, Proposition 4 is a four-valued generalization of two-valued disjunction, where it is true if either argument is true, and false if both arguments are false. It explains some unintuitive truth table entries like $\mathbf{B} \vee \mathbf{N} = \mathbf{T}$.

	$\neg$		$\neg_{\mathbf{t}}$		$\neg_{\mathbf{f}}$
T	**F**	**T**		**T**	**F**
F	**T**	**F**	**T**	**F**	
B	**B**	**B**	**T**	**B**	**F**
N	**N**	**N**		**N**	

Table 2.4: The truth tables of negation and its decomposition.

Proposition 5. *For a formula A and interpretation I,*

- $\mathbf{t} \in V_I(\neg A)$ *iff* $\mathbf{f} \in V_I(A)$*;*
- $\mathbf{f} \in V_I(\neg A)$ *iff* $\mathbf{t} \in V_I(A)$*.*

Four-valued negation is also a generalization of two-valued negation, where it is true if its argument is false and false if its argument is true.

Proposition 6. *For formulas A and B in interpretation I,*

- $\mathbf{t} \in V_I(A \otimes B)$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \in V_I(B)$*;*
- $\mathbf{f} \in V_I(A \otimes B)$ *iff* $\mathbf{f} \in V_I(A)$ *and* $\mathbf{f} \in V_I(B)$*.*

$\otimes$	**T**	**F**	**B**	**N**	$\otimes_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$\otimes_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**N**	**T**	**N**	**T**	**T**		**T**		**T**				
F	**N**	**F**	**F**	**N**	**F**					**F**		**F**	**F**	
B	**T**	**F**	**B**	**N**	**B**	**T**		**T**		**B**		**F**	**F**	
N	**N**	**N**	**N**	**N**	**N**					**N**				

Table 2.5: The truth tables of consensus and its decomposition.

The consensus operator is meant to represent the information that two truth values can agree on; it is only true if both arguments are true, and only false if both arguments are false. As we can see from its truth table, and from its decomposition, the consensus operator behaves exactly like set intersection for Belnap truth values.

$\oplus$	**T**	**F**	**B**	**N**	$\oplus_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$\oplus_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**B**	**B**	**T**	**T**	**T**	**T**	**T**	**T**	**T**		**F**	**F**	
F	**B**	**F**	**B**	**F**	**F**	**T**		**T**		**F**	**F**	**F**	**F**	**F**
B	**B**	**B**	**B**	**B**	**B**	**T**	**T**	**T**	**T**	**B**	**F**	**F**	**F**	**F**
N	**T**	**F**	**B**	**N**	**N**	**T**		**T**		**N**		**F**	**F**	

Table 2.6: The truth tables of gullibility and its decomposition.

Proposition 7. *For formulas A and B in interpretation I,*

- $\mathbf{t} \in V_I(A \oplus B)$ *iff* $\mathbf{t} \in V_I(A)$ *or* $\mathbf{t} \in V_I(B)$*;*
- $\mathbf{f} \in V_I(A \oplus B)$ *iff* $\mathbf{f} \in V_I(A)$ *or* $\mathbf{f} \in V_I(B)$*.*

The gullibility operator is meant to represent combining all the information contained in two truth values, believing all of it regardless of any contradictions; it is true if either argument is true, and is false if either argument is false. Similarly to consensus, we can see from its truth table and decomposition that the gullibility operator behaves exactly like set union for Belnap truth values.

	$-$		$-_{\mathbf{t}}$		$-_{\mathbf{f}}$
T	**T**	**T**	**T**	**T**	
F	**F**	**F**		**F**	**F**
B	**N**	**B**		**B**	
N	**B**	**N**	**T**	**N**	**F**

Table 2.7: The truth tables of conflation and its decomposition.

Proposition 8. *For a formula A and interpretation I,*

- $\mathbf{t} \in V_I(-A)$ *iff* $\mathbf{f} \notin V_I(A)$*;*
- $\mathbf{f} \in V_I(-A)$ *iff* $\mathbf{t} \notin V_I(A)$*.*

From simply examining Proposition 8, it is unclear what the conflation operator is meant to represent, aside from a simple mechanical inversion of the information ordering. The intuitive meaning of this operator will become apparent in the context of logic programming involving both classical negation and default negation, as we will show in the next chapter of this thesis.

We conclude this chapter with a few final remarks.

One, when we look at the truth and falsity conditions of the Belnap bilattice operators as logical operators in their own right, we can see that some of them are equivalent to each other, and others can be converted into each other by composing them with some other operators. This can be verified via truth tables.

Proposition 9. *For formulas A and B in interpretation I, the following identities hold:*

- $A \wedge_{\mathbf{t}} B = A \otimes_{\mathbf{t}} B$; $A \vee_{\mathbf{t}} B = A \oplus_{\mathbf{t}} B$; $A \wedge_{\mathbf{f}} B = A \oplus_{\mathbf{f}} B$; $A \vee_{\mathbf{f}} B = A \otimes_{\mathbf{f}} B$;
- $\neg(\neg A \vee_{\mathbf{f}} \neg B) = A \wedge_{\mathbf{t}} B$; $\neg(\neg A \wedge_{\mathbf{t}} \neg B) = A \vee_{\mathbf{f}} B$;
- $\neg(\neg A \vee_{\mathbf{t}} \neg B) = A \wedge_{\mathbf{f}} B$; $\neg(\neg A \wedge_{\mathbf{f}} \neg B) = A \vee_{\mathbf{t}} B$;
- $\neg\neg_{\mathbf{t}}\neg A = \neg_{\mathbf{f}} A$; $\neg\neg_{\mathbf{f}}\neg A = \neg_{\mathbf{t}} A$; $\neg -_{\mathbf{t}}\neg A = -_{\mathbf{f}} A$; $\neg -_{\mathbf{f}}\neg A = -_{\mathbf{t}} A$;
- $\neg_{\mathbf{t}} - \neg A = -_{\mathbf{t}} A$; $-_{\mathbf{t}} - \neg A = \neg_{\mathbf{t}} A$; $\neg_{\mathbf{f}} - \neg A = -_{\mathbf{f}} A$; $-_{\mathbf{f}} - \neg A = \neg_{\mathbf{f}} A$.

Two, Belnap's original formulation of his logic [2] does not contain consensus, gullibility, or conflation; the only defined operators are conjunction, disjunction, and negation. Without the three information ordering operators, Belnap logic is not functionally complete, as in it cannot represent all possible functions that map from $\mathbf{4}^n$ to $\mathbf{4}$. Most notably, it cannot represent tautologies or contradictions, as in no formula is $\mathbf{T}$ in every interpretation or $\mathbf{F}$ in every interpretation. However, if consensus, gullibility, and conflation are added to Belnap logic, then it does become functionally complete; Omori and Sano [15] examine this issue more deeply.

Three, Belnap's original formulation of his logic does not actually contain any implication operator. Simply using $\neg A \vee B$ is undesirable, because it does not follow the modus ponens property of "if A is true then B is true" for some of the truth values. He defines that formula A entails B iff $V_I(A) \leq_t V_I(B)$ for all interpretations I. This definition avoids the principle of explosion, $A \wedge \neg A \to B$, and also invalidates disjunctive syllogism, $(A \vee B) \wedge \neg A \to B$, but does not allow for any formula to explicitly contain an implication in itself; we cannot have a formula like $P \equiv A \to B$. There are works like Hazen, Pelletier, and Sutcliffe 2018 [11] that examine the topic of adding explicit implication operators to Belnap logic, noting that there are many possible generalizations of the classical two-valued implication. We will hold off on defining implication operators in Belnap logic until Chapter 4, which focuses on

how several different Belnap implication operators can be implemented and used in ASP; the topic of modus ponens and $\neg A \vee B$ will also be revisited.

Chapter 3

Default Negation in Four-Valued Answer Set Programming

This chapter focuses on the behavior of default negation in logic programming that uses Belnap logic as a basis. A brief explanation of the concept of default negation itself will be provided.

3.1 Logic Programming and Stable Model Semantics

We will begin by introducing two-valued ASP without any extensions (like classical negation that will be covered later in this chapter). This is one of several variants of ASP that will be covered in this thesis, denoted as $\mathbf{ASP^2}$, the superscript indicating that it uses two-valued classical logic as a basis. Unless specified otherwise, all definitions and propositions on ASP can be assumed to have been taken from Gelfond and Lifschitz [8] [9].

Definition 6. Let $\mathbf{ASP^2}$ be a language with the following objects:

- An *interpretation* is a set of atoms.
- A *d-literal* (default literal) is either a positive d-literal or a negative d-literal. A *positive* d-literal is an atom a, and a *negative* d-literal is the *default negation* of an atom, $\sim a$.
- A *rule* is a formula of the form $(d_0 \leftarrow d_1, ..., d_n)$ [1] where the *head* d_0 is a positive d-literal or empty, and the *body* $(d_1, ..., d_n)$ is a conjunction of d-literals (positive

[1] Rules are written with brackets around them because they contain commas. Without brackets surrounding them, it would be difficult to tell apart the commas in a rule and the commas delimiting different rules in a set.

or negative) or empty. [2] $head(r)$ is the head of rule r, $body(r)$ is the body of rule r, $body^+(r)$ is the set of literals that appear as positive d-literals in $body(r)$, and $body^-(r)$ is the set of literals that appear as negative d-literals in $body(r)$.

- A *program* is a set of rules.

Definition 7. An object defined in Definition 6 may or may not be *satisfied* by an interpretation I. Different objects have different satisfaction conditions:

- A positive d-literal a is satisfied by I iff $a \in I$. A negative d-literal $\sim a$ is satisfied by I iff $a \notin I$.
- A conjunction of d-literals $(d_1, ..., d_n)$ is satisfied by I if all d-literals in it are satisfied by I.
- A rule r is not satisfied by I iff its body is satisfied but its head is not satisfied. An empty rule body is always satisfied, and an empty rule head is never satisfied. [3]
- A program P is satisfied by I iff all rules in it are satisfied. In this case, I is a *model* of P. If there exists no other model I' such that $I' \subset I$, then I is a *minimal model* of P.

Definition 8. The *reduct* of a program P relative to interpretation I, written as P^I, is the program containing the rules $\{(head(r) \leftarrow body^+(r)) \mid r \in P, body^-(r) \cap I = \emptyset\}$. I is a *stable model* of P iff I is a minimal model of P^I.

The only type of negation allowed in two-valued ASP without extensions is default negation, so-called because of its relation to the concept of defaults. Defaults, most prominently handled by Reiter's default logic [18], are statements about what to believe "by default" in the absence of information. For example, we may have a formula saying that we can assume by default that a swan is white, in the absence of information explicitly stating that the swan is black. Defaults give logic programming the ability to be non-monotonic, to retract previous conclusions when faced with new contrary evidence. We may have concluded q based on the rule "if we don't know p to be true, then conclude q", but we must retract this conclusion if we later discover that p is true. Every ASP program can be converted into a default logic program [9], though this is not something we need to be concerned about for this thesis. All we need to know is that default negation indicates an absence

[2] Logical programming rules of this form are based on Horn clauses by Horn [12], which can be thought of as an implication whose antecedent is a conjunction of atoms, and whose consequent is an atom, or equivalently a disjunction containing exactly one positive literal.

[3] A rule with an empty body, like $(d_0 \leftarrow)$, is called a *fact*, which forces d_0 to be satisfied in every model. A rule with an empty head, like $(\leftarrow d_1, ..., d_n)$, is called an *integrity constraint*, which prevents $d_1, ..., d_n$ from all being satisfied at the same time in any model.

of information, that a negative d-literal $\sim a$ means "we do not know a to be true". Seeing that default negation indicates the absence of information, we can already see its potential relation to the presence and absence of truth and falsity in Belnap logic.

A stable model, a.k.a. an "answer set" in Answer Set Programming, is the crux of stable model semantics by Gelfond and Lifschitz [8]. The reduct of a program P relative to interpretation I removes any rule that contains a negative d-literal not satisfied by I (since these rules are trivially satisfied by I), and removes all negative d-literals from any rules that remain (since these d-literals are satisfied by I), resulting in a simplified program that contains only positive d-literals, which is guaranteed to have a single unique minimal model; I is a stable model of P iff it is the unique minimal model of P^I. The intuition behind stable models is that a stable model is a rational, self-justifying set of beliefs, able to re-derive no more and no less information than itself based on its own information. Every stable model is a minimal model, and minimal models are desirable because we do not want to assume the truth of anything more than what is strictly necessary. Another intuitive meaning approximately relates stable model semantics to the well-founded semantics by Van Gelder, Ross, and Schlipf [21], where an atom is "unfounded" if it is either 1) unable to be derived from any rule in P, or 2) only derivable through circular reasoning (like $a \to a$); unfounded atoms are not in any stable model of a program.

Before we proceed further, we should examine a few examples of ASP to establish some intuitions.

Example 1. Let $P = \{(a \leftarrow \sim b)\}$, which has three models: $I_1 = \{a\}$, $I_2 = \{b\}$, and $I_3 = \{a, b\}$.
$P^{I_1} = \{(a \leftarrow)\}$, whose minimal model is $\{a\} = I_1$, so I_1 is a stable model.
$P^{I_2} = \emptyset$, whose minimal model is $\emptyset \neq I_2$, so I_2 is not a stable model.
$P^{I_3} = \emptyset$, whose minimal model is $\emptyset \neq I_3$, so I_3 is not a stable model.
This example shows that b, an atom that cannot be derived from any rule, is not in a stable model.

Example 2. Let $P = \{(a \leftarrow b), (b \leftarrow a)\}$, which has two models: $I_1 = \emptyset$, $I_2 = \{a, b\}$.
$P^{I_1} = \{(a \leftarrow b), (b \leftarrow a)\}$, whose minimal model is $\emptyset = I_1$, so I_1 is a stable model.
$P^{I_2} = \{(a \leftarrow b), (b \leftarrow a)\}$, whose minimal model is $\emptyset \neq I_2$, so I_2 is not a stable model.
This example shows that "circular reasoning" like $a \to b$ and $b \to a$ results in the involved atoms to not be in a stable model.

Example 3. Let $P = \{(a \leftarrow \sim a)\}$, which has one model: $I = \{a\}$.
$P^I = \emptyset$, whose minimal model is $\emptyset \neq I$, so I is not a stable model.
This example shows that "self-contradicting" rules can result in programs that have no stable models.

Example 4. Let $P = \{(a \leftarrow \sim b), (b \leftarrow \sim a)\}$, which has three models: $I_1 = \{a\}$, $I_2 = \{b\}$, $I_3 = \{a, b\}$.
$P^{I_1} = \{(a \leftarrow)\}$, whose minimal model is $\{a\} = I_1$, so I_1 is a stable model.
$P^{I_2} = \{(b \leftarrow)\}$, whose minimal model is $\{b\} = I_2$, so I_2 is a stable model.
$P^{I_3} = \emptyset$, whose minimal model is $\emptyset \neq I_3$, so I_3 is not a stable model.
This example shows that some programs can have more than one stable model.

Example 5. Let $P_1 = \{(a \leftarrow)\}$ and $P_2 = \{(\leftarrow \sim a)\}$. P_1 has one stable model, $\{a\}$, whereas P_2 has no stable models. This example shows that facts allow us to conclude new atoms, while integrity constraints by themselves do not let us make new conclusions; they only serve to remove existing unacceptable conclusions.

In $\mathbf{ASP^2}$, the only information we can have about an atom is information asserting its truth, and an atom a is (assumed by default to be) false in interpretation I iff I contains no information about the truth of a. This is in accordance with the closed-world assumption, also originally formulated by Reiter [17], stating that all true formulas are known to be true, and any formula not known to be true is false. This is a necessity in logic programming that uses strictly two-valued classical logic, where the only possible states for a formula are true and false. In the absence of information on a formula's truth, or in other words, if we fail to find any proof for a formula's truth, then by default we assume that formula to be false; this allows us to derive a formula's falsity by negation as failure (to prove the formula's truth). Falsity is the default state, and a minimal model has a minimal number of atoms that we assume to be true.

One major problem with $\mathbf{ASP^2}$ is that it does not allow us to express explicitly negative information. The most common extension to it allows it to do exactly that, adding classical negation to the language. However, this requires changing the definition of d-literals and interpretations, and causes a few other problems, as we will discuss below.

Definition 9. Let $\mathbf{ASP^2_{\mathit{CN}}}$ be an extension of $\mathbf{ASP^2}$ with the following modifications:

- A *literal* is either a positive literal or a negative literal. A *positive* literal is an atom a, while a *negative* literal is the classical negation of an atom, $\neg a$.
- A positive d-literal is now a literal l, while a negative d-literal is now a default-negated literal, $\sim l$; in either case l may be a positive or a negative literal.
- An interpretation is now a set of literals.

All other definitions remain unchanged.

We can see that positive and negative literals are effectively being treated as different atoms, a remark made explicit by Gelfond and Lifschitz [9] when they proved that $\mathbf{ASP^2_{\mathit{CN}}}$

programs can be converted into $\mathbf{ASP^2}$ programs by replacing each negative literal with a corresponding new atom. A positive d-literal is an explicit assertion of truth or falsity, and a negative d-literal is a statement on the absence of information on truth and falsity. The resemblance to Belnap logic starts to become more apparent here, but we are technically still working in the realm of classical, two-valued logic, which does not allow gluts (both true and false) or gaps (neither true nor false). To retain the behavior of classicality, a few additional clauses must be added to every logic program.

Proposition 10. *Let U be the set of all atoms that appear in $\mathbf{ASP^2_{\mathit{CN}}}$ program P. Then, for program $P' = P \cup \{(b \leftarrow a, \neg a) \mid a, b \in U\} \cup \{(\neg b \leftarrow a, \neg a) \mid a, b \in U\}$, the only model of P' that contains any pair of complementary literals a and $\neg a$ is the set of all possible literals that can be created from atoms in U. For program $P'' = P \cup \{(\leftarrow a, \neg a) \mid a \in U\}$, no model of P'' contains any pair of complementary literals a and $\neg a$.*

Program P' in Proposition 10 has rules that artificially enforce the principle of explosion, which can only be satisfied by an interpretation containing all possible literals if any pair of complementary literals is present at all. P'' instead simply has integrity constraints that cannot be satisfied by any interpretation that contains complementary literals. Both approaches enforce classicality by eliminating contradictions, and only one approach is needed; different implementations of ASP may not use the same approach. At least one of these two types of rules must be automatically added to every program to ensure that its models are not inconsistent.

Most modern ASP implementations used in industry do not disallow gaps. While they do not allow any model to have both an atom a and its classical negation $\neg a$, it is perfectly allowed to have neither a nor $\neg a$ in a model. This implicitly means that $\mathbf{ASP^2_{\mathit{CN}}}$ is a three-valued logic, where an atom can be true, false, or neither, but not both. It may depend on the specific problem being modelled whether a gap is interpreted as false, no information, or something else. We can try to enforce the closed-world assumption by adding rules like $(\neg a \leftarrow \sim a)$, but doing so makes us no longer able to conceptually distinguish between classical and default negation, and can change a program's stable models, [4] so doing so may not be a good idea.

Example 6. Let $P_1 = \{(b \leftarrow \sim a), (\leftarrow a, \neg a), (\leftarrow b, \neg b)\}$, and $P_2 = \{(b \leftarrow \neg a), (\leftarrow a, \neg a), (\leftarrow b, \neg b)\}$. The only stable model of P_1 is $\{b\}$, where the lack of information on the truth of a lets us conclude b. On the other hand, the only stable model of P_2 is $\emptyset$, where we cannot conclude b because we do not have explicit information on the falsity of a. In

[4] For example, the program $\{(b \leftarrow \neg a)\}$ has $\emptyset$ as its only stable model, but adding $(\neg a \leftarrow \sim a)$ to the program causes its only stable model to become $\{\neg a, b\}$.

neither stable model do we have any actual information on a; whether a counts as false or unknown here would depend on the specifications of the problem being modelled.

Example 7. Let $P = \{(b \leftarrow \sim a), (\neg b \leftarrow \sim a), (a \leftarrow \sim b, \sim\neg b), (\leftarrow a, \neg a), (\leftarrow b, \neg b)\}$. If the contradiction-removing rule $(\leftarrow b, \neg b)$ was not in P, then it would have two stable models, $\{a\}$ and $\{b, \neg b\}$, but the presence of this rule makes $\{a\}$ the only acceptable stable model.

Before we proceed further, we need to take a step back and examine how rules in $\mathbf{ASP^2}$ correspond to formulas in classical propositional logic.

Proposition 11. *Let I be a set of atoms that functions as both an $\mathbf{ASP^2}$ interpretation and a classical logic interpretation. Let $a_0, ..., a_n$ be atoms. Then I satisfies the $\mathbf{ASP^2}$ rule $(a_0 \leftarrow a_1, ..., a_m, \sim a_{m+1}, ..., \sim a_n)$ iff I is a model of the classical logical formula $a_1 \wedge ... \wedge a_m \wedge \neg a_{m+1} \wedge ... \wedge \neg a_n \rightarrow a_0$.*

Note that Proposition 11 simply refers to models of a rule, from Definition 7, not stable models. It is true because $\mathbf{ASP^2}$ d-literals have the same satisfaction conditions as classical logical literals, and similarly for $\mathbf{ASP^2}$ conjunctions versus classical logical conjunctions, and $\mathbf{ASP^2}$ rules versus classical logical implications; it shows that rules in $\mathbf{ASP^2}$ are meant to represent implications in classical logic. However, generalizing Proposition 11 to $\mathbf{ASP^2_{\mathit{CN}}}$ program P is awkward, as $\mathbf{ASP^2_{\mathit{CN}}}$ program P is actually a three-valued logic with the inclusion of Proposition 10, not to mention the difficulty of representing two different types of negation (classical and default) in classical logic. This thesis will show that Belnap logic provides an intuitive generalization of Proposition 11 into ASP with classical negation.

Definition 10. Let $\mathbf{ASP^4}$ (four-valued ASP) be a language with identical specifications to $\mathbf{ASP^2_{\mathit{CN}}}$. However, the classicality-enforcing rules in Proposition 10 are not automatically added to any programs.

As stated previously, when classical negation is involved, positive and negative literals can be effectively treated as separate atoms, and ASP based on four-valued logic has no need to artificially restrict the presence of gluts or gaps in an interpretation. Since an interpretation in $\mathbf{ASP^4}$ is a set of literals, it can also be treated as an interpretation in Belnap logic.

Example 8. Let $P = \{(a \leftarrow), (\leftarrow \neg a), (\neg b \leftarrow), (\leftarrow b), (c \leftarrow), (\neg c \leftarrow), (\leftarrow d), (\leftarrow \neg d)\}$. The only stable model of P is $I = \{a, \neg b, c, \neg c\}$, where $V_I(a) = \mathbf{T}$, $V_I(b) = \mathbf{F}$, $V_I(c) = \mathbf{B}$, and $V_I(d) = \mathbf{N}$, showing us that we can still specify the truth values of atoms exactly in $\mathbf{ASP^4}$.

Example 9. Let $P = \{(b \leftarrow a, \neg a), (a \leftarrow), (\neg a \leftarrow), (d \leftarrow c)\}$. The only stable model of P is $I = \{a, \neg a, b\}$, where $V_I(a) = \mathbf{B}$, $V_I(b) = \mathbf{T}$, and $V_I(c) = V_I(d) = \mathbf{N}$. This example shows that a rule can still require an atom to have a specific truth value to be able to make a conclusion, like $(b \leftarrow a, \neg a)$ requiring a to be $\mathbf{B}$ to conclude b. It also shows that, despite

the presence of a contradiction, the program does not explode into triviality and conclude the irrelevant atoms c and d.

To generalize Proposition 11 to $\mathbf{ASP^4}$, there are two main tasks that must be done:

1. representing default negation in Belnap logic;
2. finding a suitable four-valued implication that ASP rules represent.

Task 1 will be covered by the remainder of this chapter, while the next chapter is dedicated to Task 2.

3.2 Default Negation in Four-Valued ASP

Since we are leaving implications for the next chapter, for now we only need to consider individual d-literals that may be default-negated. First, we need to consider what satisfying a d-literal in $\mathbf{ASP^4}$ corresponds to in Belnap logic. For positive d-literals, it is possible for an interpretation I to satisfy both a d-literal d and its classical negation $\neg d$, or satisfy neither, independently of each other, like how a Belnap formula can be independently true, false, both, or neither, suggesting that satisfying a d-literal in $\mathbf{ASP^4}$ corresponds to truth in Belnap logic, and satisfying the d-literal's classical negation corresponds to falsity. Therefore, we can impose the following condition for conversion between $\mathbf{ASP^4}$ and Belnap logic.

Definition 11. Let I be an interpretation, and X an object in $\mathbf{ASP^4}$ for which the satisfaction of X and $\neg X$ by interpretations is defined, from Definitions 6 and 7. Let the *Belnap logical conversion* of X, $BLC(X)$, be a Belnap logical formula fulfilling the conditions that I satisfies X iff $\mathbf{t} \in V_I(BLC(X))$, and I satisfies $\neg X$ iff $\mathbf{f} \in V_I(BLC(X))$. Additionally, let BLC^{-1} be the inverse of BLC such that $BLC(X) = Y$ iff $X = BLC^{-1}(Y)$.

Definition 11 is not a constructive definition. It only provides the conditions that a Belnap formula must fulfill to be considered the BLC of X, and it is up to us to actually find what $BLC(X)$ is, necessitating much of the work in this thesis. Sometimes, the classical negation of X is not yet defined, in which case we must then define it ourselves in a way that is intuitively reasonable and useful to the problem we are modeling.

For a positive d-literal l: l as a d-literal is satisfied by I iff $l \in I$, and l as a Belnap literal is true in I iff $l \in I$; $\neg l$ as a d-literal is satisfied by I iff $\neg l \in I$, and l is false in I iff $\neg l \in I$. Therefore, the BLC of a positive d-literal is exactly what we intuitively expect.

Proposition 12. *For a positive d-literal l, where l is a literal, $BLC(l) = l$.*

To find the BLC of a negative d-literal $\sim l$, where l is a literal, we must first define its classical negation. We have already established in Proposition 12 that I satisfying a positive

d-literal l means l is true in I in Belnap logic. Since negative d-literals represent the absence of information, it is intuitive to guess that I satisfying a negative d-literal $\sim l$ means l is not true in I in Belnap logic. The classical negation of "l is true" is "$\neg l$ is true", which intuitively suggests that the classical negation of "l is not true" is "$\neg l$ is not true"; therefore it is intuitively reasonable to define that $\neg(\sim l) \equiv \sim\neg l$. This means we have the following conditions to fulfill:

- I satisfies $\sim l$ iff $\mathbf{t} \in V_I(BLC(\sim l))$;
- I satisfies $\sim\neg l$ iff $\mathbf{f} \in V_I(BLC(\sim l))$.

Which, according to the satisfaction conditions of negative d-literals, become:

- $l \notin I$ iff $\mathbf{t} \in V_I(BLC(\sim l))$;
- $\neg l \notin I$ iff $\mathbf{f} \in V_I(BLC(\sim l))$.

From Definition 11 and Proposition 12: $l \in I$ iff $\mathbf{t} \in V_I(l)$, so $l \notin I$ iff $\mathbf{t} \notin V_I(l)$; $\neg l \in I$ iff $\mathbf{f} \in V_I(l)$, so $\neg l \notin I$ iff $\mathbf{f} \notin V_I(l)$. Therefore, the conditions we need to fulfill become the following:

- $\mathbf{t} \notin V_I(l)$ iff $\mathbf{t} \in V_I(BLC(\sim l))$;
- $\mathbf{f} \notin V_I(l)$ iff $\mathbf{f} \in V_I(BLC(\sim l))$.

We have in fact arrived at a decomposition for an operator that is the Belnap logic version of default negation, which we will also denote by $\sim$ for consistency. This means we now have enough information to determine its complete truth table.

	$\sim$		$\sim_{\mathbf{t}}$		$\sim_{\mathbf{f}}$
T	**F**	**T**		**T**	**F**
F	**T**	**F**	**T**	**F**	
B	**N**	**B**		**B**	
N	**B**	**N**	**T**	**N**	**F**

Table 3.1: The truth tables of default negation and its decomposition.

As we can see from Table 3.1, $\sim l$ inverts both the truth ordering and the information ordering of l, making it equivalent to $-\neg l$. Finally, we arrive at the conclusion of how the conflation operator is related to default negation, the first major finding of this thesis.

Theorem 1. *For a negative d-literal $\sim l$, where l is a literal, $BLC(\sim l) = \sim l = -\neg l$.*

Corollary 1.1. *For a literal l, $BLC^{-1}(-l) = \sim\neg l$.*

This is the intuitive meaning of the conflation operator, mentioned near the end of Chapter 2. It applies classical negation and default negation simultaneously to a literal, and the two

types of negation do not cancel each other out. Conflation switches between two different types of "evidence" in favor of a formula, asserting truth versus asserting absence of falsity, converting "is true" (**T** or **B**) into "is not false" (**T** or **N**). Classical negation, on the other hand, inverts truth and falsity, converting "is true" into "is false" (**F** or **B**). Applying both classical negation and conflation converts "is true" into "is not true" (**F** or **N**), the default negation of "is true".

In addition to the steps we used above to arrive at $BLC(\sim l) = -\neg l$, there is another way of showing $BLC(\sim\neg l) = -l$.

Proof. For a literal l and interpretation I:

$\mathbf{t} \in V_I(-l)$ iff $\mathbf{f} \notin V_I(l)$, according to the decomposition of the conflation operator.
$\mathbf{f} \notin V_I(l)$ iff $\mathbf{t} \notin V_I(\neg l)$.
$\mathbf{t} \notin V_I(\neg l)$ iff I does not satisfy the d-literal $\neg l$.
I does not satisfy $\neg l$ iff I satisfies $\sim\neg l$.
Therefore, $\mathbf{t} \in V_I(-l)$ iff I satisfies $\sim\neg l$.

$\mathbf{f} \in V_I(-l)$ iff $\mathbf{t} \notin V_I(l)$, according to the decomposition of the conflation operator.
$\mathbf{t} \notin V_I(l)$ iff I does not satisfy the d-literal l.
I does not satisfy l iff I satisfies $\sim l$.
Therefore, $\mathbf{f} \in V_I(-l)$ iff I satisfies $\sim l$.

Because $\mathbf{t} \in V_I(-l)$ iff I satisfies $\sim\neg l$ and $\mathbf{f} \in V_I(-l)$ iff I satisfies $\sim l$, $BLC(\sim\neg l) = -l$. □

With Theorem 1, we now have a concept of default negation in Belnap logic, where $\sim l = -\neg l$ for a literal l. Therefore, d-literals are defined in Belnap logic as well, where a d-literal d can be treated as both a Belnap logical formula and an $\mathbf{ASP^4}$ construct.

Definition 12. In Belnap logic, a *d-literal* is either a positive d-literal or a negative d-literal. A *positive* d-literal is a literal l, while a *negative* d-literal is the default negation of a literal, $\sim l = -\neg l$.

By Proposition 12, Theorem 1, and Definition 12, the BLC of a d-literal in $\mathbf{ASP^4}$ is simply the same d-literal in Belnap logic.

Proposition 13. *For a d-literal d, $BLC(d) = d$.*

Having established the equivalence of d-literals in $\mathbf{ASP^4}$ and Belnap logic, we are now ready to examine rules that join d-literals together.

Chapter 4

Implications in Four-Valued Answer Set Programming

Having defined the Belnap logical conversions of individual d-literals in $\mathbf{ASP^4}$, we now turn our attention to rules.

4.1 The Basic Implication Operator

The bulk of this chapter concerns the implication operators that logic programming rules are meant to represent, but before that, the conjunction of d-literals that forms a rule's body needs to be considered. Conjunctions of d-literals have already been defined as a part of the $\mathbf{ASP^4}$ specification, but for clarity they will be defined again here, along with disjunctions of d-literals.

Definition 13. Let I be an interpretation, and $d_1, ..., d_n$ be d-literals. Then:

- $(d_1, ..., d_n)$ is a *conjunction* of d-literals, satisfied by I iff every one of $d_1, ..., d_n$ is satisfied by I. An empty conjunction is always satisfied.
- $(d_1; ...; d_n)$ is a *disjunction* of d-literals, satisfied by I iff at least one of $d_1, ..., d_n$ is satisfied by I. An empty disjunction is never satisfied.
- The classical negation of $(d_1, ..., d_n)$ is not yet defined, so we define it as $(\neg d_1; ...; \neg d_n)$.
- The classical negation of $(d_1; ...; d_n)$ is not yet defined, so we define it as $(\neg d_1, ..., \neg d_n)$.

Since a conjunction in $\mathbf{ASP^4}$ is meant to represent a conjunction in Belnap logic, we use the intuition of De Morgan's laws to define that the classical negation of an $\mathbf{ASP^4}$ conjunction is an $\mathbf{ASP^4}$ disjunction, and vice versa. This will be relevant later in this chapter when we touch upon disjunctive logic programming, and is needed to show that the BLC of an $\mathbf{ASP^4}$ conjunction is indeed a Belnap logical conjunction.

Proposition 14. *For a conjunction of d-literals* $(d_1, ..., d_n)$*,* $BLC((d_1, ..., d_n)) = d_1 \wedge ... \wedge d_n$*.*

Proof. Let $d_1, ..., d_n$ be d-literals, and I an interpretation.

$(d_1, ..., d_n)$ is satisfied by I iff I satisfies all of $d_1, ..., d_n$.
For all $1 \leq i \leq n$, I satisfies d_i iff $\mathbf{t} \in V_I(d_i)$.
Therefore, I satisfies $(d_1, ..., d_n)$ iff $\mathbf{t} \in V_I(d_i)$ for all $1 \leq i \leq n$.
$\mathbf{t} \in V_I(d_i)$ for all $1 \leq i \leq n$ iff $\mathbf{t} \in V_I(d_1 \wedge ... \wedge d_n)$.
Therefore, I satisfies $(d_1, ..., d_n)$ iff $\mathbf{t} \in V_I(d_1 \wedge ... \wedge d_n)$.

$\neg(d_1, ..., d_n)$ is satisfied by I iff I satisfies any of $\neg d_1, ..., \neg d_n$.
For any $1 \leq i \leq n$, I satisfies $\neg d_i$ iff $\mathbf{f} \in V_I(d_i)$.
Therefore, I satisfies $\neg(d_1, ..., d_n)$ iff $\mathbf{f} \in V_I(d_i)$ for any $1 \leq i \leq n$.
$\mathbf{f} \in V_I(d_i)$ for any $1 \leq i \leq n$ iff $\mathbf{f} \in V_I(d_1 \wedge ... \wedge d_n)$.
Therefore, I satisfies $\neg(d_1, ..., d_n)$ iff $\mathbf{f} \in V_I(d_1 \wedge ... \wedge d_n)$.

Since I satisfies $(d_1, ..., d_n)$ iff $\mathbf{t} \in V_I(d_1 \wedge ... \wedge d_n)$ and I satisfies $\neg(d_1, ..., d_n)$ iff $\mathbf{f} \in V_I(d_1 \wedge ... \wedge d_n)$, $BLC((d_1, ..., d_n)) = d_1 \wedge ... \wedge d_n$. □

Now we move on to implications, the core part of every program rule. In $\mathbf{ASP^2}$, the behavior of the arrow in a rule appropriately matches the behavior of implications in classical logic, as in $a \to b = \neg a \vee b$, for models (but not stable models). The presence of classical negation in $\mathbf{ASP^2}_{CN}$ causes program rules to start exhibiting behaviors different from classical implications, the exact details of which are not relevant to this thesis. Generalizing to $\mathbf{ASP^4}$ has even more issues, as there are multiple possible implication operators with different properties.

Belnap's original formulation of his logic [2] contains no implication operator (not using $\neg A \vee B$ due to it not following modus ponens for some truth values), instead only containing a meta-theoretic entailment relation where formula A entails formula B iff $V_I(A) \leq_t V_I(B)$ for all interpretations I. This means entailment is a binary condition that a pair of formulas may or may not fulfill, instead of being a four-valued operator, and entailment also cannot be embedded into a formula like $A \to B$. The definition of $V_I(A) \leq_t V_I(B)$ as an implication operator is a usable one, but it is not the only intuitively sensible type of implication that we can think of in four-valued logic. Hazen et al. [11] examine this in detail, listing a number of desirable properties that one can expect a useful implication operator to have. As logic is meant to be a simplified model of reality, how the model behaves ultimately boils down to the requirements of the specific problem being modelled, again without any objectively correct one-size-fits-all answer. The list by Hazen et al. was produced via intuition, but Hazen et al. stated that they themselves do not fully agree on which of these properties are necessary, and I personally do not agree to some of them as I consider them to be unnecessarily restrictive

(e.g. what they call designated antecedent in a diamond). Therefore, I will simply list the clauses that are considered in the writing of this thesis.

Two important properties of the classical two-valued implications are modus ponens (if the antecedent is true then the consequent is true; not satisfying this condition causes the implication to be false) and vacuous truth (if the antecedent is false or the consequent is true then the implication is true). However, in Belnap logic, there are two types of "evidence" of a formula's truth: information on its truth (**T** or **B**), and the lack of information on its falsity (**T** or **N**); the conflation operator swaps between these two types of evidence. Similarly, there are two types of evidence of falsity, information on falsity (**F** or **B**) and the lack of information on truth (**F** or **N**). This means we can have several similar versions of modus ponens and vacuous truth in a Belnap implication. Additionally, since the Belnap implication should be a generalization of the classical implication, it must take on the same truth values as the classical implication when its arguments are classical. Overall, these are the properties that we may intuitively define a Belnap implication operator to have:

- *Modus ponens*: If A is (true or not false) then B is (true or not false); not satisfying this condition causes $A \to B$ to be (not true or false).
- *Vacuous truth*: If A is (not true or false) or B is (true or not false), then $A \to B$ is (true or not false).
- *Classicality*: $A \to B$ must agree with the two-valued implication when its arguments are **T** or **F**.

For now, these are merely rough guidelines rather than hard requirements. They will be made more rigorous at a later section in this chapter.

Moreover, another property that will receive significant attention in this chapter is contraposition, where $A \to B = \neg B \to \neg A$. Not every implication operator discussed below will have this property; its presence and absence will be directly relevant.

Before examining viable implication operators, we will quickly remark that defining $A \to B \equiv \neg A \lor B$ in Belnap logic does not fulfill the classical "if A is true then B is true" version of modus ponens, but it does fulfill several other versions of modus ponens and vacuous truth listed above. We will leave $\neg A \lor B$ for now and come back to it after we have defined modus ponens and vacuous truth rigorously.

First, we examine how the basic "arrow operator" in $\mathbf{ASP^4}$ rules defined thus far translates to Belnap logic. Let $\to_t$ be an operator in Belnap logic such that $BLC((d \leftarrow D)) = D \to_t d$, where d is a d-literal and D is a conjunction of d-literals; the reason for calling this operator $\to_t$ will be elaborated on shortly. Conjunctions in $\mathbf{ASP^4}$ can be converted into conjunctions

in Belnap logic and vice versa via Proposition 14, so D can be treated as a single unit here when discussing whether it is satisfied.

Remembering that $(d \leftarrow D)$ is not satisfied iff D is satisfied but d is not satisfied, by the definition of BLCs, we know that $\mathbf{t} \notin V_I(D \rightarrow_t d)$ iff $\mathbf{t} \in V_I(D)$ and $\mathbf{t} \notin V_I(d)$ for any interpretation I; this gives us the truth condition of $\rightarrow_t$. Rule satisfaction in $\mathbf{ASP^4}$ is a binary condition rather than four-valued, so we can define $\rightarrow_t$ as false iff it is not true, which causes its falsity condition to be $\mathbf{f} \in V_I(D \rightarrow_t d)$ iff $\mathbf{t} \in V_I(D)$ and $\mathbf{t} \notin V_I(d)$. With both halves of its decomposition, we can construct its full truth table.

$\rightarrow_t$	**T**	**F**	**B**	**N**	$(\rightarrow_t)_\mathbf{t}$	**T**	**F**	**B**	**N**	$(\rightarrow_t)_\mathbf{f}$	**T**	**F**	**B**	**N**
T	**T**	**F**	**T**	**F**	**T**	**T**		**T**		**T**		**F**		**F**
F	**T**	**T**	**T**	**T**	**F**	**T**	**T**	**T**	**T**	**F**				
B	**T**	**F**	**T**	**F**	**B**	**T**		**T**		**B**		**F**		**F**
N	**T**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**T**	**N**				

Table 4.1: The truth tables of $\rightarrow_t$ and its decomposition.

Theorem 2. *For an arbitrary* $\mathbf{ASP^4}$ *rule containing d-literals* $d_0, ..., d_n$*,* $BLC((d_0 \leftarrow d_1, ..., d_n)) = d_1 \wedge ... \wedge d_n \rightarrow_t d_0$*.*

$\rightarrow_t$ represents the "preservation of truth" from antecedent to consequent, in exactly the same way as how rules in $\mathbf{ASP^4}$ behave, hence the t subscript. Only the preservation of truth matters to $\rightarrow_t$, while the falsities of its consequent and antecedent have no effect on its value, and it does not have the contraposition property; these behaviors appropriately reflect how $\mathbf{ASP^4}$ treats positive and negative literals of the same atom as effectively different atoms that do not affect each other. It agrees with the two-valued implication where appropriate, and obeys the following versions of modus ponens and vacuous truth:

- If A is true then B is true; not satisfying this condition causes $A \rightarrow_t B$ to be not true.
- If A is true then B is true; not satisfying this condition causes $A \rightarrow_t B$ to be false.
- If A is not true or B is true then $A \rightarrow_t B$ is true.
- If A is not true or B is true then $A \rightarrow_t B$ is not false.

With this, we now have a complete mapping of $\mathbf{ASP^4}$ to Belnap logic. Any Belnap logical formula of the appropriate form can be converted as-is into an $\mathbf{ASP^4}$ rule as per Theorem 2.

4.2 Other Implication Operators in Belnap Logic

Our work does not end here, however, as there are a few other implication operators in Belnap logic that are worth examining. So far, $\mathbf{ASP^4}$ still only treats positive and negative

literals of the same atom as practically different atoms, but as we will soon see, some of the other implication operators do not behave this way.

One of the implication operators considered in Hazen et al. [11], and also mentioned in Omori and Sano [15], is the "classical material implication" $\rightarrow_{cmi}$, originally proposed by Arieli and Avron [1]. Its definition is that the implication takes on the same value as the consequent if the antecedent is true; otherwise the implication is (vacuously) equal to $\mathbf{T}$. Arieli and Avron defined a consequence relation $A \models^{\mathbf{4}} B$ iff every model of A is a model of B, and showed that $X \wedge A \models B$ iff $X \models^{\mathbf{4}} A \rightarrow_{cmi} B$ for all Belnap formulas X, A, and B; so $\rightarrow_{cmi}$ can be considered one of the more intuitive implication operators in Belnap logic.

To see how $\rightarrow_{cmi}$ can be implemented in $\mathbf{ASP^4}$, we start with its decomposition.

$\rightarrow_{cmi}$	**T**	**F**	**B**	**N**	$(\rightarrow_{cmi})_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$(\rightarrow_{cmi})_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**F**	**B**	**N**	**T**	**T**		**T**		**T**		**F**	**F**	
F	**T**	**T**	**T**	**T**	**F**	**T**	**T**	**T**	**T**	**F**				
B	**T**	**F**	**B**	**N**	**B**	**T**		**T**		**B**		**F**	**F**	
N	**T**	**T**	**T**	**T**	**N**	**T**	**T**	**T**	**T**	**N**				

Table 4.2: The truth tables of $\rightarrow_{cmi}$ and its decomposition.

Proposition 15. *For formulas A and B in interpretation I,*

- $\mathbf{t} \notin V_I(A \rightarrow_{cmi} B)$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \notin V_I(B)$*;*
- $\mathbf{f} \in V_I(A \rightarrow_{cmi} B)$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{f} \in V_I(B)$*.*

$\rightarrow_{cmi}$ obeys the following versions of modus ponens and vacuous truth:

- If A is true then B is true; not satisfying this condition causes $A \rightarrow_{cmi} B$ to be not true.
- If A is true then B is not false; not satisfying this condition causes $A \rightarrow_{cmi} B$ to be false.
- If A is not true or B is true then $A \rightarrow_{cmi} B$ is true.
- If A is not true or B is not false then $A \rightarrow_{cmi} B$ is not false.

We can immediately see that $(\rightarrow_{cmi})_{\mathbf{t}}$ is identical to $(\rightarrow_{t})_{\mathbf{t}}$, the preservation of truth from antecedent to consequent. But whereas the falsity condition of $\rightarrow_{t}$ is simply the opposite of its truth condition, making its result binary, the falsity condition of $\rightarrow_{cmi}$ is different. $A \rightarrow_{cmi} B$ is true iff truth is preserved from A to B, while it is false iff A is true and B is false. This sounds exactly like the truth and falsity "conditions" of the implication operator

in classical logic, except in classical logic these two conditions are not independent from each other.

Now is the time to take a step back and remember what these implication operators in Belnap logic, and $\mathbf{ASP^4}$ rules, are meant to model in the real world. An implication $A \rightarrow B$ represents the idea of "if A then B" in some sense. A model that uses $\rightarrow_t$ would see truth preservation from A to B as information stating $A \rightarrow B$ is true, and the violation of such truth preservation as information stating $A \rightarrow B$ is false. The truth and falsity conditions of $\rightarrow_t$ cannot overlap, so $\rightarrow_t$ is a binary condition that is either satisfied or violated. However, if the model uses $\rightarrow_{cmi}$, it still sees truth preservation from A to B as information stating that $A \rightarrow B$ is true, but it would also see A being true and B being false as information stating that $A \rightarrow B$ is false. For a rule that uses $\rightarrow_{cmi}$, we can think of an interpretation that satisfies its truth condition as an "example" that "satisfies" this rule, and an interpretation that satisfies its falsity condition as an explicit "counter-example" that "violates" this rule. Should an interpretation be considered a model of $A \rightarrow B$ if it is both an example and a counter-example of $A \rightarrow_{cmi} B$? If it is neither an example nor a counter-example of $A \rightarrow_{cmi} B$? There is no objectively right answer to these questions; the treatment of gluts and gaps in a program's rules would likely vary depending on the specific requirements of the problem being modelled.

The most intuitive approach is that a model should satisfy and also not violate a rule. Since $\rightarrow_t$ and $\rightarrow_{cmi}$ have the same truth condition, and $D \rightarrow_t d$ is implemented in $\mathbf{ASP^4}$ as the rule $(d \leftarrow D)$, we know that $\mathbf{t} \in V_I(D \rightarrow_{cmi} d)$ iff I satisfies $(d \leftarrow D)$; this means the truth condition of $D \rightarrow_{cmi} d$ can be implemented as the same rule. On the other hand, the falsity condition of $D \rightarrow_{cmi} d$ is identical to the non-satisfaction condition of $(\leftarrow \neg d, D)$, so this integrity constraint can be used to eliminate the interpretations that are counter-examples to $D \rightarrow_{cmi} d$. This proves the following theorem:

Theorem 3. *Let I be an interpretation, and $d_0, ..., d_n$ be d-literals.*

- $\mathbf{t} \in V_I(d_1, ..., d_n \rightarrow_{cmi} d_0)$ *iff the rule* $(d_0 \leftarrow d_1, ..., d_n)$ *is satisfied by* I.
- $\mathbf{f} \in V_I(d_1, ..., d_n \rightarrow_{cmi} d_0)$ *iff the rule* $(\leftarrow \neg d_0, d_1, ..., d_n)$ *is not satisfied by* I.

The presence of $\neg d$ in the implementation of the falsity condition of $D \rightarrow_{cmi} d$ indicates that d and $\neg d$ are no longer treated as entirely independent from each other; they both affect the truth value of $D \rightarrow_{cmi} d$.

As previously stated, it depends on the specific requirements of the problem being modelled whether satisfaction and violation should be taken into account when determining whether an interpretation should be a model of a rule that uses $\rightarrow_{cmi}$. If satisfying the rule is relevant, its truth condition can be implemented as a regular $\mathbf{ASP^4}$ rule. If not violating the rule is

relevant, its falsity condition can be implemented as an integrity constraint. One or both halves of the rule can be implemented as the problem requires, and this treatment can even be done on a per-rule basis. For example, one rule may have both its truth and falsity conditions implemented, requiring its truth value to be **T** only in models, while another rule may have only its truth condition implemented, allowing its truth value to be either **T** or **B**.

The two implication operators we have examined so far both do not follow the contraposition property, while the next two implication operators we will examine do have this property. Contraposition is not supported by $\mathbf{ASP^4}$ rules alone, so some additional considerations will be needed to implement such a feature.

The first contrapositive-supporting implication operator will be called $\rightarrow_{le}$, where the implication is **T** iff the antecedent is less than or equal to the consequent in the truth ordering; otherwise the implication is **F**. This is the implication operator originally presented by Belnap [2], except converted into an operator in the logic itself instead of simply being a meta-theoretical relation between two formulas, meaning it can be embedded in formulas as one can do in classical logic. As usual, we start with the decomposition of this operator.

$\rightarrow_{le}$	**T**	**F**	**B**	**N**	$(\rightarrow_{le})_\mathbf{t}$	**T**	**F**	**B**	**N**	$(\rightarrow_{le})_\mathbf{f}$	**T**	**F**	**B**	**N**
T	**T**	**F**	**F**	**F**	**T**	**T**				**T**		**F**	**F**	**F**
F	**T**	**T**	**T**	**T**	**F**	**T**	**T**	**T**	**T**	**F**				
B	**T**	**F**	**T**	**F**	**B**	**T**		**T**		**B**		**F**		**F**
N	**T**	**F**	**F**	**T**	**N**	**T**			**T**	**N**		**F**	**F**	

Table 4.3: The truth tables of $\rightarrow_{le}$ and its decomposition.

Proposition 16. *For formulas A and B in interpretation I:*

- $\mathbf{t} \notin V_I(A \rightarrow_{le} B)$ *iff* $(\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \notin V_I(B))$ *or* $(\mathbf{f} \notin V_I(A)$ *and* $\mathbf{f} \in V_I(B))$*;*
- $\mathbf{f} \in V_I(A \rightarrow_{le} B)$ *iff* $(\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \notin V_I(B))$ *or* $(\mathbf{f} \notin V_I(A)$ *and* $\mathbf{f} \in V_I(B))$*.*

Proposition 17. *For formulas A and B, $A \rightarrow_{le} B = X(A,B) \wedge Y(A,B)$, where $X(A,B) = A \rightarrow_t B = -\neg B \rightarrow_t -\neg A$ and $Y(A,B) = \neg B \rightarrow_t \neg A = -A \rightarrow_t -B$.*

Belnap in [2] derived that the $\leq_t$ condition in four-valued logic requires forward preservation of truth from antecedent to consequent, and backward preservation of falsity from consequent to antecedent. $\rightarrow_t$ represents forward truth preservation only, and does not obey contraposition, so $A \rightarrow_{le} B$ is constructed from conjoining $A \rightarrow_t B$ with its "contrapositive" $\neg B \rightarrow_t \neg A$. Backward preservation of falsity is also equivalent to forward preservation of non-falsity (if antecedent is not false then consequent is not false), represented by $-A \rightarrow_t -B$, a fact that can easily be checked via truth tables. Similarly, forward

preservation of truth is equivalent to backward preservation of non-truth (if consequent is not true then antecedent is not true), represented by $-\neg B \to_t -\neg A$. As a result, there are four total ways in which $\to_{le}$ can be constructed from $\to_t$, hence writing $X(A, B)$ and $Y(A, B)$ in two possible forms each in Proposition 17. Later in this chapter we will examine contrapositives in Belnap logic more carefully, referring back to these two different "types" of contrapositives.

The final implication we will look at is the "strong implication" $\to_{si}$ in Arieli and Avron [1], defined as $A \to_{si} B = (A \to_{cmi} B) \wedge (\neg B \to_{cmi} \neg A)$. This is similar to the $A \to_{le} B = (A \to_t B) \wedge (\neg B \to_t \neg A)$ above, conjoining a non-contrapositive implication operator with its "contrapositive".

$\to_{si}$	**T**	**F**	**B**	**N**	$(\to_{si})_{\mathbf{t}}$	**T**	**F**	**B**	**N**	$(\to_{si})_{\mathbf{f}}$	**T**	**F**	**B**	**N**
T	**T**	**F**	**F**	**N**	**T**	**T**				**T**		**F**	**F**	
F	**T**	**T**	**T**	**T**	**F**	**T**	**T**	**T**	**T**	**F**				
B	**T**	**F**	**B**	**N**	**B**	**T**		**T**		**B**		**F**	**F**	
N	**T**	**N**	**N**	**T**	**N**	**T**			**T**	**N**				

Table 4.4: The truth tables of $\to_{si}$ and its decomposition.

Proposition 18. *For formulas A and B in interpretation I:*

- $\mathbf{t} \notin V_I(A \to_{si} B)$ *iff* $(\mathbf{t} \in V_I(A)$ *and* $\mathbf{t} \notin V_I(B))$ *or* $(\mathbf{f} \notin V_I(A)$ *and* $\mathbf{f} \in V_I(B))$*;*
- $\mathbf{f} \in V_I(A \to_{si} B)$ *iff* $\mathbf{t} \in V_I(A)$ *and* $\mathbf{f} \in V_I(B)$*.*

We can see that the truth condition of $\to_{si}$ is identical to the truth condition of $\to_{le}$, requiring both forward truth preservation and backward falsity preservation, while its falsity condition is identical to the falsity condition of $\to_{cmi}$, being violated when its antecedent is true and consequent is false. The truth condition of $\to_{si}$ can be constructed in four possible ways, similarly to $\to_{le}$ stated earlier in Proposition 17.

Proposition 19. *For formulas A and B, $A(\to_{le})_{\mathbf{t}}B = X_{\mathbf{t}}(A, B) \wedge Y_{\mathbf{t}}(A, B)$, where $X_{\mathbf{t}}(A, B) = A(\to_{cmi})_{\mathbf{t}}B = -\neg B(\to_{cmi})_{\mathbf{t}} -\neg A$ and $Y(A, B) = \neg B(\to_{cmi})_{\mathbf{t}}\neg A = -A(\to_{cmi})_{\mathbf{t}} -B$. However, there is no similar equivalence for $(\to_{si})_{\mathbf{f}}$.*

Since $\to_{le}$ can be constructed from two $\to_t$ rules, and similarly for $\to_{si}$ with $\to_{cmi}$, these two implications can be implemented in $\mathbf{ASP^4}$ simply by implementing their constituents. However, rather than writing them out as separate theorems, we can devise a general classification system for implication operators in Belnap logic, taking into account the list of properties that we may intuitively expect implication operators to have. This system would encompass all four implication operators we have examined so far, and have a simple and intuitive way of implementation in $\mathbf{ASP^4}$.

4.3 General Unidirectional Implication Operators in Belnap Logic

We are now going to rigorously define modus ponens and vacuous truth in Belnap logic. The four implication operators we examined previously all fulfill some of these modus ponens and vacuous truth properties, which we will show later in the relevant section.

Definition 14. Let Op be a binary Belnap operator, and $P, Q, R \in \{+, -\}$, where $+$ is the unary identity operator in Belnap logic and $-$ is the conflation operator. [1] Then $MP(Op)$ is the set of *modus ponens properties* of Op, and $VT(Op)$ is the set of *vacuous truth properties* of Op, each of which being a set of triples of operators in $\{+, -\}$.

- $(P, Q, R) \in MP(Op)$ iff the following condition holds: $V_I(Op(A, B)) \in \{R(\mathbf{F}), R(\mathbf{N})\}$ iff $V_I(A) \in \{P(\mathbf{T}), P(\mathbf{B})\}$ and $V_I(B) \in \{Q(\mathbf{F}), Q(\mathbf{N})\}$ for all formulas A, B and all interpretations I.
- $(P, Q, R) \in VT(Op)$ iff the following condition holds: $V_I(Op(A, B)) \in \{R(\mathbf{T}), R(\mathbf{B})\}$ iff $V_I(A) \in \{P(\mathbf{F}), P(\mathbf{N})\}$ or $V_I(B) \in \{Q(\mathbf{T}), Q(\mathbf{B})\}$ for all formulas A, B and all interpretations I.

As stated earlier in this chapter, in Belnap logic there are two types of "evidence" toward the truth of a formula X: X being true, and X being not false. Similarly, X being false and X being not true are two types of "evidence" toward the falsity of a formula. Modus ponens and vacuous truth properties make use of these types of "evidence", with the required type of "evidence" being controlled by the P, Q, R functions. For example, if $P = +$, then $V_I(X) \in \{P(\mathbf{T}), P(\mathbf{B})\} = \{\mathbf{T}, \mathbf{B}\}$ means X is true in I, whereas if $P = -$, then $V_I(X) \in \{P(\mathbf{T}), P(\mathbf{B})\} = \{\mathbf{T}, \mathbf{N}\}$ means X is not false in I. The modus ponens property represents the property of "if A is (true or not false, controlled by P) and B is (not true or false, controlled by Q), then $Op(A, B)$ is (not true or false, controlled by R)". The vacuous truth property represents the property of "if A is (not true or false, controlled by P) or B is (true or not false, controlled by Q), then $Op(A, B)$ is (true or not false, controlled by R)". Of course, not all two-argument Belnap operators satisfy these properties; the operators that satisfy these properties are what we may consider to be implication operators.

Theorem 4. *For a two-argument Belnap operator Op, and* $P, Q, R \in \{+, -\}$*,* $(P, Q, R) \in MP(Op)$ *iff* $(P, Q, R) \in VT(Op)$*.*

[1] The $+$ symbol is used here to denote the identity operator, because it and the conflation operator denoted by $-$ play roles akin to multiplication by 1 and -1, where multiplying by 1 does nothing, multiplying by -1 turns it from "positive" to "negative" and vice versa, and multiplying by -1 twice also does nothing.

Proof. Let Op be a two-argument Belnap operator, $P, Q, R \in \{+, -\}$, A, B be Belnap formulas, and I an interpretation.

Assume $(P, Q, R) \in VT(Op)$, so that $V_I(Op(A, B)) \in \{R(\mathbf{T}), R(\mathbf{B})\}$ iff $V_I(A) \in \{P(\mathbf{F}), P(\mathbf{N})\}$ or $V_I(B) \in \{Q(\mathbf{T}), Q(\mathbf{B})\}$. Then we negate both sides of this biconditional.

The left side becomes $V_I(Op(A, B)) \notin \{R(\mathbf{T}), R(\mathbf{B})\}$.
If $R = +$, then $\{R(\mathbf{T}), R(\mathbf{B})\} = \{\mathbf{T}, \mathbf{B}\}$, so the left side becomes $V_I(Op(A, B)) \notin \{\mathbf{T}, \mathbf{B}\}$, which means $V_I(Op(A, B)) \in \{\mathbf{F}, \mathbf{N}\}$, because $V_I(Op(A, B))$ has to take on a value in $\mathbf{4}$. And because $R = +$, $\{\mathbf{F}, \mathbf{N}\} = \{R(\mathbf{F}), R(\mathbf{N})\}$, so the left side becomes $V_I(Op(A, B)) \in \{R(\mathbf{F}), R(\mathbf{N})\}$.
Similarly, if $R = -$, then $V_I(Op(A, B)) \notin \{R(\mathbf{T}), R(\mathbf{B})\} = \{\mathbf{T}, \mathbf{N}\}$ means $V_I(Op(A, B)) \in \{\mathbf{F}, \mathbf{B}\} = \{R(\mathbf{F}), R(\mathbf{N})\}$.
In both cases, the left-side condition becomes $V_I(Op(A, B)) \in \{R(\mathbf{F}), R(\mathbf{N})\}$.

The right side becomes $V_I(A) \notin \{P(\mathbf{F}), P(\mathbf{N})\}$ and $V_I(B) \notin \{Q(\mathbf{T}), Q(\mathbf{B})\}$. By similar procedures as above, this becomes $V_I(A) \in \{P(\mathbf{T}), P(\mathbf{B})\}$ and $V_I(B) \in \{Q(\mathbf{F}), Q(\mathbf{N})\}$.

Overall, the biconditional becomes $V_I(Op(A, B)) \in \{R(\mathbf{F}), R(\mathbf{N})\}$ iff $V_I(A) \in \{P(\mathbf{T}), P(\mathbf{B})\}$ and $V_I(B) \in \{Q(\mathbf{F}), Q(\mathbf{N})\}$, which is the definition of $(P, Q, R) \in MP(Op)$ according to Definition 14.

For the reverse direction, assume $(P, Q, R) \in MP(Op)$, so that $V_I(Op(A, B)) \in \{R(\mathbf{F}), R(\mathbf{N})\}$ iff $V_I(A) \in \{P(\mathbf{T}), P(\mathbf{B})\}$ and $V_I(B) \in \{Q(\mathbf{F}), Q(\mathbf{N})\}$. Then we can negate both sides of the biconditional, and use the same procedures as above to show that the biconditional is equivalent to $V_I(Op(A, B)) \in \{R(\mathbf{T}), R(\mathbf{B})\}$ iff $V_I(A) \in \{P(\mathbf{F}), P(\mathbf{N})\}$ or $V_I(B) \in \{Q(\mathbf{T}), Q(\mathbf{B})\}$ for all possible values of P, Q, R. As per Definition 14, this is the definition of $(P, Q, R) \in VT(Op)$.

Therefore, $(P, Q, R) \in MP(Op)$ iff $(P, Q, R) \in VT(Op)$. $\square$

Theorem 4 shows that the modus ponens and vacuous truth properties are actually different ways of describing the same property. This makes sense if we examine the truth table of an implication operator. The modus ponens property specifies four entries in the truth table that are different from the rest of the entries (e.g. not true instead of true). The vacuous truth property specifies two rows and two columns, leaving out four entries that are the same four entries specified by the modus ponens property. Therefore we only need to specify modus ponens properties from now on, since every modus ponens property has an equivalent vacuous truth property.

Definition 15. Let a *general unidirectional implication* $\xrightarrow[f,g]{F,G}$ be a two-argument Belnap logical operator such that for Belnap formulas A and B, $A \xrightarrow[f,g]{F,G} B = (-\neg F(A) \vee_{\mathbf{t}} G(B)) \oplus$

$(\neg f(A) \vee_{\mathbf{f}} -g(B))$, where $\vee_{\mathbf{t}}$ and $\vee_{\mathbf{f}}$ are the truth and falsity conditions of the disjunction operator shown in Table 2.3 (where blank entries indicate $\mathbf{N}$ as previously stated), and $F, G, f, g \in \{+, -\}$. [2]

Theorem 5. *Let $F, G, f, g \in \{+, -\}$. Then* $\{(F, G, +), (f, g, -)\} \subseteq MP\left(\xrightarrow[f,g]{F,G}\right)$.

Proof. Let $F, G, f, g \in \{+, -\}$, A, B be Belnap formulas, and I an interpretation. Let $Op(A, B) = A \xrightarrow[f,g]{F,G} B$, so that $Op_{\mathbf{t}}(A, B) = -\neg F(A) \vee_{\mathbf{t}} G(B)$ and $Op_{\mathbf{f}}(A, B) = \neg f(A) \vee_{\mathbf{f}} -g(B)$.

For the truth condition of Op, we know that $\mathbf{t} \notin V_I(Op(A, B))$ iff $\mathbf{t} \notin V_I(-\neg F(A))$ and $\mathbf{t} \notin V_I(G(B))$.
For the right side of this biconditional, we know that $\mathbf{t} \notin V_I(-\neg F(A))$ iff $\mathbf{f} \in V_I(\neg F(A))$, and $\mathbf{f} \in V_I(\neg F(A))$ iff $\mathbf{t} \in V_I(F(A))$. Therefore, the right side becomes $\mathbf{t} \notin V_I(Op(A, B))$ iff $\mathbf{t} \in V_I(F(A))$ and $\mathbf{t} \notin V_I(G(B))$.
We also know that $\mathbf{t} \in V_I(F(A))$ iff $V_I(F(A)) \in \{\mathbf{T}, \mathbf{B}\}$. Since $F(F(A)) = A$ for any $F \in \{+, -\}$, we then know that $V_I(F(A)) \in \{\mathbf{T}, \mathbf{B}\}$ iff $V_I(F(F(A))) = V_I(A) \in \{F(\mathbf{T}), F(\mathbf{B})\}$.
Similar to above, $\mathbf{t} \notin V_I(G(B))$ iff $V_I(B) \in \{G(\mathbf{F}), G(\mathbf{N})\}$.
Therefore, the biconditional becomes $\mathbf{t} \notin V_I(Op(A, B))$ iff $V_I(A) \in \{F(\mathbf{T}), F(\mathbf{B})\}$ and $V_I(B) \in \{G(\mathbf{F}), G(\mathbf{N})\}$.
For the left side of the biconditional, we know that $\mathbf{t} \notin V_I(Op(A, B))$ iff $V_I(Op(A, B)) \in \{\mathbf{F}, \mathbf{N}\} = \{+\mathbf{F}, +\mathbf{N}\}$.
Therefore, the biconditional becomes $V_I(Op(A, B)) \in \{+\mathbf{F}, +\mathbf{N}\}$ iff $V_I(A) \in \{F(\mathbf{T}), F(\mathbf{B})\}$ and $V_I(B) \in \{G(\mathbf{F}), G(\mathbf{N})\}$. This is the same condition as the definition of $(F, G, +) \in MP(Op)$ as per Definition 14.

For the falsity condition of Op, we know that $\mathbf{f} \in V_I(Op(A, B))$ iff $\mathbf{f} \in V_I(\neg f(A))$ and $\mathbf{f} \in V_I(-g(B))$.
For the right side of this biconditional, we know that $\mathbf{f} \in V_I(\neg f(A))$ iff $\mathbf{t} \in V_I(f(A))$, and $\mathbf{f} \in V_I(-g(B))$ iff $\mathbf{t} \notin V_I(g(B))$. Therefore, the right side becomes $\mathbf{f} \in V_I(Op(A, B))$ iff $\mathbf{t} \in V_I(f(A))$ and $\mathbf{t} \notin V_I(g(B))$.
By procedures similar to the previous half of this proof, we can show that $\mathbf{t} \in V_I(f(A))$ iff $V_I(A) \in \{f(\mathbf{T}), f(\mathbf{B})\}$ and $\mathbf{t} \notin V_I(g(B))$ iff $V_I(B) \in \{g(\mathbf{F}), g(\mathbf{N})\}$. Therefore, the biconditional becomes $\mathbf{f} \in V_I(Op(A, B))$ iff $V_I(A) \in \{f(\mathbf{T}), f(\mathbf{B})\}$ and $V_I(B) \in \{g(\mathbf{F}), g(\mathbf{N})\}$.
For the left side of the biconditional, we know that $\mathbf{f} \in V_I(Op(A, B))$ iff $V_I(Op(A, B)) \in \{\mathbf{F}, \mathbf{B}\} = \{-\mathbf{F}, -\mathbf{N}\}$.
Therefore, the biconditional becomes $V_I(Op(A, B)) \in \{-\mathbf{F}, -\mathbf{N}\}$ iff $V_I(A) \in \{f(\mathbf{T}), f(\mathbf{B})\}$

[2] Here $-\neg F(A) \vee_{\mathbf{t}} G(B)$ is the truth condition, since the $\vee_{\mathbf{t}}$ can only take on values in $\{\mathbf{T}, \mathbf{N}\}$. Similarly, $\neg f(A) \vee_{\mathbf{f}} -g(B)$ is the falsity condition, and the two are combined via gullibility as per Definition 5.

and $V_I(B) \in \{g(\mathbf{F}), g(\mathbf{N})\}$. This is the same condition as the definition of $(f, g, -) \in MP(Op)$ as per Definition 14. □

Theorem 5 shows that $\xrightarrow[f,g]{F,G}$ has two separate modus ponens properties, one with its truth condition and one with its falsity condition. Its truth condition is "if the antecedent is (true or not false, controlled by F) and the consequent is (not true or false, controlled by G), then the implication is not true", while its falsity condition is "if the antecedent is (true or not false, controlled by f) and the consequent is (not true or false, controlled by g), then the implication is false". The functions F, G, f, g control the specifics of these modus ponens properties. F, G are placed above the arrow, representing how they control the implication's truth condition, while f, g are below the arrow, controlling its falsity condition. F, f are at the left side of the arrow, controlling how it handles the antecedent, while G, g are at the right side, controlling how it handles the consequent.

Theorem 6. *Let A, B be Belnap formulas, and I an interpretation where $V_I(A), V_I(B) \in \{\mathbf{T}, \mathbf{F}\}$. Then for any $F, G, f, g \in \{+, -\}$, if $V_I(A) = \mathbf{T}$ and $V_I(B) = \mathbf{F}$, then $V_I\left(A \xrightarrow[f,g]{F,G} B\right) = \mathbf{F}$; otherwise $V_I\left(A \xrightarrow[f,g]{F,G} B\right) = \mathbf{T}$.*

Proof. Let A, B be Belnap formulas, and I an interpretation where $V_I(A), V_I(B) \in \{\mathbf{T}, \mathbf{F}\}$. Then because $V_I(A), V_I(B) \in \{\mathbf{T}, \mathbf{F}\}$, for any $F, G, f, g \in \{+, -\}$, $V_I(F(A)) = V_I(f(A)) = V_I(A)$, and $V_I(G(B)) = V_I(g(B)) = V_I(B)$. So in I, $A \xrightarrow[f,g]{F,G} B$ simplifies down to $(-\neg A \vee_{\mathbf{t}} B) \oplus (\neg A \vee_{\mathbf{f}} -B)$. For the same reason, $V_I(-A) = V_I(A)$ and $V_I(-B) = V_I(B)$, so the expression further simplifies down to $(\neg A \vee_{\mathbf{t}} B) \oplus (\neg A \vee_{\mathbf{f}} B) = \neg A \vee B$, which we know to match the behavior of classical logic when $V_I(A), V_I(B) \in \{\mathbf{T}, \mathbf{F}\}$. □

With Theorems 5 and 6, we can see that the general unidirectional implication operator is a proper four-valued generalization of the two-valued implication operator. However, it is unidirectional because it does not have the contraposition property. If we want such a property, we must define the contrapositive ourselves and manually conjoin it with the original implication, much like what was done with $\rightarrow_{le}$ and $\rightarrow_{si}$.

Definition 16. Let A, B be Belnap formulas, and Op be a two-argument Belnap operator. Then $C_\neg$ and C_- are *contraposition transformations*, where $C_\neg(Op(A, B)) = Op(\neg B, \neg A)$, $C_-(Op(A, B)) = Op(-A, -B)$, and $C_{-\neg}(Op(A, B)) = C_\neg(C_-(Op(A, B))) = C_-(C_\neg(Op(A, B))) = Op(-\neg A, -\neg B)$.

Theorem 7. *Let A, B be Belnap formulas, and $F, G, f, g \in \{+, -\}$.*

- $C_\neg\left(A \xrightarrow[f,g]{F,G} B\right) = A \xrightarrow[-g,-f]{-G,-F} B$.

- $C_{-}\left(A \xrightarrow[f,g]{F,G} B\right) = A \xrightarrow[-f,-g]{-F,-G} B$.
- $C_{-\neg}\left(A \xrightarrow[f,g]{F,G} B\right) = A \xrightarrow[g,f]{G,F} B$.

Proof. Let A, B be Belnap formulas, $F, G, f, g \in \{+, -\}$, and $Op(A, B) = A \xrightarrow[f,g]{F,G} B$. First, we note that for any $h \in \{+, -\}$ and formula X, $h(\neg X) = \neg h(X)$ and $h(-X) = -h(X)$, because negation and conflation are commutative with each other.

Let $C_{\neg}(Op(A, B)) = (-\neg F'(A) \vee_{\mathbf{t}} G'(B)) \oplus (\neg f'(A) \vee_{\mathbf{f}} -g'(B))$.
$C_{\neg}(Op_{\mathbf{t}}(A, B)) = -\neg F(\neg B) \vee_{\mathbf{t}} G(\neg A) = -F(B) \vee_{\mathbf{t}} \neg G(A) = -\neg(-G(A)) \vee_{\mathbf{t}} (-F(B))$. Therefore, $F' = -G$ and $G' = -F$.
$C_{\neg}(Op_{\mathbf{f}}(A, B)) = \neg f(\neg B) \vee_{\mathbf{f}} -g(\neg A) = f(B) \vee_{\mathbf{f}} -\neg g(A) = \neg(-g(A)) \vee_{\mathbf{f}} -(-f(B))$. Therefore, $f' = -g$ and $g' = -f$.

C_{-} is much simpler. For any $h'(X) = h(-X)$, we know $h' = -h$, which applies to all of F, G, f, g in $C_{-}(Op(A, B))$.

The $C_{-\neg}$ case can be obtained by simply composing the procedures of the above two cases. □

With Theorem 7, we can now construct $\rightarrow_t$, $\rightarrow_{cmi}$, $\rightarrow_{le}$, and $\rightarrow_{si}$ from general unidirectional implications. The following theorem can be verified by simply checking the appropriate truth tables.

Theorem 8. *Let A, B be Belnap formulas.*

- $A \rightarrow_t B = A \xrightarrow[+,+]{+,+} B$.
- $A \rightarrow_{cmi} B = A \xrightarrow[+,-]{+,+} B$.
- $A \rightarrow_{le} B = \left(A \xrightarrow[+,+]{+,+} B\right) \wedge C_{\neg}\left(A \xrightarrow[+,+]{+,+} B\right) = \left(A \xrightarrow[+,+]{+,+} B\right) \wedge C_{-}\left(A \xrightarrow[+,+]{+,+} B\right) = \left(A \xrightarrow[+,+]{+,+} B\right) \wedge \left(A \xrightarrow[-,-]{-,-} B\right)$.
- $A \rightarrow_{si} B = \left(A \xrightarrow[+,-]{+,+} B\right) \wedge C_{\neg}\left(A \xrightarrow[+,-]{+,+} B\right) = \left(A \xrightarrow[+,-]{+,+} B\right) \wedge \left(A \xrightarrow[+,-]{-,-} B\right) \neq \left(A \xrightarrow[+,-]{+,+} B\right) \wedge C_{-}\left(A \xrightarrow[+,-]{+,+} B\right) = \left(A \xrightarrow[+,-]{+,+} B\right) \wedge \left(A \xrightarrow[-,+]{-,-} B\right)$.
- $\neg A \vee B = A \xrightarrow[+,-]{-,+} B$.

As we can see, $\rightarrow_{le}$ can be constructed from $\rightarrow_t$ and either its $C_\neg$ contrapositive or its C_- contrapositive, but $\rightarrow_{si}$ must be constructed from $\rightarrow_{cmi}$ and its $C_\neg$ contrapositive; its C_- contrapositive will not work.

Theorem 8 tells us the modus ponens and vacuous truth properties that the four other implication operators we previously examined have, through Theorem 5. The same can be said about $\neg A \vee B$, as stated earlier in this thesis when it was last mentioned. It does not fulfill $(+,+,+)$ or $(+,+,-)$, the purely forward truth preservation form of modus ponens displayed by $\rightarrow_t$, but instead fulfills $(-,+,+)$ and $(+,-,-)$. Therefore, whether $\neg A \vee B$ is usable or useful as an implication operator depends on the specific problem being modelled.

Overall, a large number of possible implication operators can be constructed from general unidirectional implication operators and their contrapositives. However, conjoining multiple general unidirectional implications together means the resulting implication may no longer have modus ponens or vacuous truth properties exactly as defined in Definition 14. The intuitive meaning of such a composite implication can be determined by examining the modus ponens properties of the truth and falsity conditions of each of its unidirectional constituents, and examining how these truth and falsity conditions are combined by the truth and falsity conditions of conjunction. It ultimately depends on the specific problem being modelled whether these operators make sense.

The implementation of general unidirectional implications and contraposition in $\mathbf{ASP^4}$ has some nuances, which will be discussed in the next and final section of this chapter.

4.4 Implementation of General Unidirectional Implications and Contraposition in ASP

A Belnap formula can be implemented in $\mathbf{ASP^4}$ if it is a "rule", an implication with a conjunction of d-literals as its antecedent and a d-literal as its consequent. However, at first glance, it seems that the $C_\neg$ contrapositive of a rule is an implication with a d-literal as its antecedent and a disjunction of d-literals as its consequent. For example, $C_\neg(d_1 \wedge ... \wedge d_n \rightarrow_t d_0) = \neg d_0 \rightarrow_t \neg d_1 \vee ... \vee \neg d_n$. This suggests that implementing contraposition may require disjunctive logic programming, a form of logic programming where the head of a rule is a disjunction of d-literals.

In a disjunctive version of $\mathbf{ASP^4}$, a rule is of the form $(d_1; ...; d_m \leftarrow d_{m+1}, ..., d_n)$, where the head is a disjunction of d-literals and the body a conjunction of d-literals; a disjunction of d-literals is satisfied iff any one of its members is satisfied. The reduct of program P with respect to interpretation I is the set of rules $\{(head^+(r) \leftarrow body^+(r) \mid r \in P, body^-(r) \cap I = \emptyset, head^-(r) \subseteq I\}$. However, allowing disjunction in rule heads causes ASP to become much

more computationally expensive [3], in addition to no longer guaranteeing the minimality of stable models. For example, the program $\{(a; \sim a \leftarrow)\}$ has two stable models, $\{a\}$ and $\emptyset$, each of which is the minimal model of its own reduct, but only one of them is the minimal model of the original program. There are many more problems and nuances with disjunctive logic programming, which are beyond the scope of this thesis, so we will avoid resorting to disjunctive logic programming for the scope of this thesis.

Fortunately, Theorem 7 allows us to have $C_\neg$ contrapositions of rules that still have a conjunction of d-literals as the antecedent and a single d-literal as the consequent: $C_\neg\left(d_1 \wedge ... \wedge d_n \xrightarrow[f,g]{F,G} d_0\right) = d_1 \wedge ... \wedge d_n \xrightarrow[-g,-f]{-G,-F} d_0$. Since there will always be no more than one d-literal in the consequent, we do not need to involve disjunctive logic programming.

Another issue with implementing general unidirectional implications, and contraposition, is that it sometimes requires the head of a rule to be a default-negated literal. First, we need to redefine a few things in $\mathbf{ASP^4}$ to accommodate allowing negative d-literals in rule heads.

Definition 17. Let $\mathbf{ASP}^{\mathbf{4}}_{NH}$ be an extension of $\mathbf{ASP^4}$ with the following modifications:

- The head of a rule can now also be a negative d-literal.
- If the head of rule r is the positive d-literal l, then $head^+(r) = \{l\}$ and $head^-(r) = \emptyset$. If the head of r is the negative d-literal $\sim l$, then $head^+(r) = \emptyset$ and $head^-(r) = \{l\}$. If the head of r is empty, then $head^+(r) = head^-(r) = \emptyset$.
- The definition of the reduct is changed such that for a program P and interpretation I, $P^I = \{(head^+(r) \leftarrow body^+(r) \mid r \in P, body^-(r) \cap I = \emptyset, head^-(r) \subseteq I\}$.

All other definitions remain unchanged.

When default negation is allowed in rule heads, the definition of the reduct is the same as the definition of the reduct in the disjunctive version of ASP briefly mentioned above. For a rule $(\sim l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$, if $l_0 \in I$ and $\{l_{m+1}, ..., l_n\} \cap I = \emptyset$, then the rule is transformed into $(\leftarrow l_1, ..., l_m)$, an integrity constraint. This fact, plus the fact that allowing default negation in rule heads does not make ASP any more expressive (as shown by Janhunen [13]), suggests that rules with default-negated heads may be replaceable by integrity constraints. We will now show that this is true.

Theorem 9. *For a conjunction of literals $L = (l_1, ..., l_n)$, let $\sim L$ denote the conjunction of d-literals $(\sim l_1, ..., \sim l_n)$. Let h be a literal, and B^+, B^- be conjunctions of literals. Let P be an*

[3] As stated in the introduction of this thesis, computational complexity is beyond the scope of this thesis, and interested readers can refer to works like Faber and Leone [5]. In short, ASP with default negation is co-NP, while allowing both default negation and disjunction results in Π_2^P.

arbitrary $\mathbf{ASP}^{\mathbf{4}}_{NH}$ *program,* $P_1 = P \cup \{(\sim h \leftarrow B^+, \sim B^-)\}$*, and* $P_2 = P \cup \{(\leftarrow h, B^+, \sim B^-)\}$*. Then* P_1 *and* P_2 *have the same stable models.*

Proof. For interpretations X, Y where $X \subseteq Y$, let (X, Y) be an SE-model of $\mathbf{ASP}^{\mathbf{4}}_{NH}$ program P iff Y is a model of P and X is a model of P^Y. Turner [20] showed that two programs have the same stable models iff they have the same SE-models. [4] Let P be an arbitrary program, $r_1 = (\sim h \leftarrow B^+, \sim B^-)$, $r_2 = (\leftarrow h, B^+, \sim B^-)$, $P_1 = P \cup \{r_1\}$, and $P_2 = P \cup \{r_2\}$; then P_1 and P_2 have the same stable models iff they have the same SE-models.

For any rule r and interpretation I, let the rule $r^I = (\mathit{head}^+(r) \leftarrow \mathit{body}^+(r))$ iff $\mathit{body}^-(r) \cap I = \emptyset$ and $\mathit{head}^-(r) \subseteq I$; otherwise r^I does not exist. For any program P such that $P' = P \cup \{r\}$, then by the definition of reducts, r^I does not exist iff $(P')^I = P^I$, and if r^I exists, then $(P')^I = P^I \cup \{r^I\}$.

Proof that every SE-model of P_1 is an SE-model of P_2:
Let (X, Y) be an SE-model of P_1, so that Y is a model of P_1 and X is a model of P_1^Y.
If r_1^Y exists, then $P_1^Y = P^Y \cup \{r_1^Y\}$; otherwise $P_1^Y = P^Y$. Either way, $P^Y \subseteq P_1^Y$. Also, X satisfies every rule in P_1^Y. Therefore, X is a model of P^Y.
r_1 and r_2 have the same models. Y is a model of every rule in P_1, so Y is a model of r_1; so Y is a model of r_2. Since $P \subseteq P_1$, Y is a model of P. Therefore, Y is a model of $P_2 = P \cup \{r_2\}$.
Case 1: Assume $B^- \cap Y \neq \emptyset$. Then r_2^Y does not exist, so $P_2^Y = P^Y$. Since X is a model of P_Y, X is a model of P_2^Y. In this case, (X, Y) is an SE-model of P_2.
Case 2: Assume $B^- \cap Y = \emptyset$. Then $r_2^Y = (\leftarrow h, B^+)$.
Case 2.1: Assume $h \notin Y$. Since $X \subseteq Y$, $h \notin X$. Therefore X satisfies r_2^Y.
Case 2.2: Assume $h \in Y$. Then $r_1^Y = (\leftarrow B^+)$, which X satisfies since (X, Y) is an SE-model of P_1, so $B^+ \not\subseteq X$. Therefore, X satisfies r_2^Y.
In both cases 2.1 and 2.2, X satisfies r_2^Y. Since X is a model of P^Y and satisfies r_2^Y, it is a model of $P^Y \cup \{r_2^Y\} = P_2^Y$. Therefore (X, Y) is an SE-model of P_2 in case 2.
In both cases 1 and 2, (X, Y) is an SE-model of P_2. Therefore, every SE-model of P_1 is an SE-model of P_2.

Proof that every SE-model of P_2 is an SE-model of P_1: Let (X, Y) instead of an SE-model of P_2, so that Y is a model of P_2 and X is a model of P_2^Y.
By similar procedures as before, X is a model of P^Y, and Y is a model of P_1.
Case 1: Assume $B^- \cap Y \neq \emptyset$. Then r_1^Y does not exist. By similar procedures as before, (X, Y) is an SE-model of P_1 in this case.

[4]Technically Turner [20] only applies to consistent programs, but our definition of $\mathbf{ASP}^{\mathbf{4}}_{NH}$ treats complementary literals as if they are different atoms (can substitute every negative literal $\neg a$ with a new atom a'), so his result is still applicable to us.

Case 2: Assume $B^- \cap Y = \emptyset$.
Case 2.1: Assume $h \notin Y$. Then r_1^Y does not exist. By similar procedures as before, (X, Y) is an SE-model of P_1 in this case.
Case 2.2: Assume $h \in Y$. Since Y is a model of $r_2 = (\leftarrow h, B^+, {\sim}B^-)$, and $h \in Y$ and $B^- \cap Y = \emptyset$, the only way for Y to satisfy r_2 is to have $B^+ \not\subseteq Y$. Since $X \subseteq Y$, that means $B^+ \not\subseteq X$. Therefore X satisfies $r_1^Y = (\leftarrow B^+)$. By similar procedures as before, (X, Y) is an SE-model of P_1 in this case.
In all cases, (X, Y) is an SE-model of P_1. Therefore, every SE-model of P_2 is an SE-model of P_1.

P_1 and P_2 have the same SE-models, so they have the same stable models. $\square$

Theorem 9 shows that every rule with a default-negated head can be transformed into an integrity constraint by moving its head to the body and removing the default negation. This way, $\mathbf{ASP}^\mathbf{4}_{NH}$ is reduced to $\mathbf{ASP}^\mathbf{4}$ without changing the language's expressiveness, and general unidirectional implications can be implemented in $\mathbf{ASP}^\mathbf{4}$ without problem.

Theorem 10. *Let $l_0, ..., l_n$ be literals, I an interpretation, and $F, G, f, g \in \{+, -\}$. Let $R = l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n \xrightarrow[f,g]{F,G} l_0$. Let r_F be a conjunction of d-literals where $r_F = (l_1, ..., l_m, {\sim}l_{m+1}, ..., {\sim}l_n)$ iff $F = +$, and $r_F = ({\sim}\neg l_1, ..., {\sim}\neg l_m, \neg l_{m+1}, ..., \neg l_n)$ iff $F = -$; and let r_f be similarly defined regarding $l_1, ..., l_n$ and f. Then,*

- *if $G = +$, $\mathbf{t} \in V_I(R)$ iff the rule $(l_0 \leftarrow r_F)$ is satisfied;*
- *if $G = -$, $\mathbf{t} \in V_I(R)$ iff the rule $(\leftarrow \neg l_0, r_F)$ is satisfied;*
- *if $g = +$, $\mathbf{f} \notin V_I(R)$ iff the rule $(l_0 \leftarrow r_f)$ is satisfied;*
- *if $g = -$, $\mathbf{f} \notin V_I(R)$ iff the rule $(\leftarrow \neg l_0, r_f)$ is satisfied.*

Overall, a general unidirectional implication is implemented as $(G(l_0) \leftarrow F(l_1), ..., F(l_m), F({\sim}l_{m+1}), ..., F({\sim}l_n))$ for its truth condition, and similarly with f, g for its falsity condition. $+$ leaves things unchanged, while $-l = {\sim}\neg l$ and $-{\sim}l = \neg l$.

Proof. Assume we have all specifications stated in Theorem 10.

Let $Op(A, B) = A \xrightarrow[f,g]{F,G} B$, where A, B are Belnap formulas. By Theorem 8, we have established that $A \xrightarrow[+,+]{+,+} B = A \rightarrow_t B$. Thus, $A(\rightarrow_t)_\mathbf{t}B = -\neg A \vee_\mathbf{t} B$ and $A(\rightarrow_t)_\mathbf{f}B = \neg A \vee_\mathbf{f} -B$. Therefore, the truth condition of $Op(A, B)$ is $F(A)(\rightarrow_t)_\mathbf{t}G(B)$, and its falsity condition is $f(A)(\rightarrow_t)_\mathbf{f}g(B)$. In other words, $\mathbf{t} \in V_I(Op(A, B))$ iff $\mathbf{t} \in V_I(F(A) \rightarrow_t G(B))$, and $\mathbf{f} \in V_I(Op(A, B))$ iff $\mathbf{f} \in V_I(f(A) \rightarrow_t g(B))$.

By Theorem 2, $\mathbf{t} \in V_I(F(A) \rightarrow_t G(B))$ iff the rule $(BLC^{-1}(G(B)) \leftarrow BLC^{-1}(F(A)))$ is satisfied. Let $A = l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n$ and $B = l_0$.
If $F = +$, then $BLC^{-1}(F(A)) = BLC^{-1}(l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n) = (l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$.
If $F = -$, then $BLC^{-1}(F(A)) = BLC^{-1}(-l_1 \wedge ... \wedge -l_m \wedge \neg l_{m+1} \wedge ... \wedge \neg l_n) = (\sim\neg l_1, ..., \sim\neg l_m, \neg l_{m+1}, ..., \neg l_n)$.
If $G = +$, then $BLC^{-1}(G(B)) = BLC^{-1}(l_0) = l_0$.
If $G = -$, then $BLC^{-1}(G(B)) = BLC^{-1}(-l_0) = \sim\neg l_0$, which can be moved from the rule head to the rule body as $\neg l_0$, via Theorem 9.
In all cases, the rule's head and body match those presented in Theorem 10.

By the definition of $\rightarrow_t$, $\mathbf{f} \in V_I(F(A) \rightarrow_t G(B))$ iff $\mathbf{t} \notin V_I(f(A) \rightarrow_t g(B))$. Therefore, $\mathbf{f} \in V_I(Op(A, B))$ iff $\mathbf{t} \notin V_I(f(A) \rightarrow_t g(B))$. In other words, $\mathbf{f} \notin V_I(Op(A, B))$ iff $\mathbf{t} \in V_I(f(A) \rightarrow_t g(B))$.
Then by Theorem 2, $\mathbf{t} \in V_I(f(A) \rightarrow_t g(B))$ iff the rule $(BLC^{-1}(g(B)) \leftarrow BLC^{-1}(f(A)))$ is satisfied. The remainder of this part of the proof proceeds identically to the previous part of the proof. □

Now we are capable of implementing all implication operators we have discussed. The general unidirectional implication operators are implemented using Theorem 10, while contraposition transformations are reduced to more general unidirectional implications via Theorem 7. Implementation summaries are given below.

$l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n \rightarrow_t l_0$ can be implemented as:

- Truth condition: $(l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$
- Falsity condition: $(l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$

Note that the truth and falsity conditions are identical and redundant with each other, due to $\rightarrow_t$ being effectively two-valued.

$l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n \rightarrow_{cmi} l_0$ can be implemented as:

- Truth condition: $(l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$
- Falsity condition: $(\leftarrow \neg l_0, l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$

$l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n \rightarrow_{le} l_0$ can be implemented as:

- Truth condition: $(l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$
- Falsity condition: $(l_0 \leftarrow l_1, ..., l_m, \sim l_{m+1}, ..., \sim l_n)$
- Truth condition of contrapositive: $(\leftarrow \neg l_0, \sim\neg l_1, ..., \sim\neg l_m, \neg l_{m+1}, ..., \neg l_n)$

- Falsity condition of contrapositive: $(\leftarrow \neg l_0, {\sim}\neg l_1, ..., {\sim}\neg l_m, \neg l_{m+1}, ..., \neg l_n)$

Like with $\rightarrow_t$, the truth and falsity conditions of $\rightarrow_{le}$ are redundant due to it being two-valued.

$l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n \rightarrow_{si} l_0$ can be implemented as:

- Truth condition: $(l_0 \leftarrow l_1, ..., l_m, {\sim}l_{m+1}, ..., {\sim}l_n)$
- Falsity condition: $(\leftarrow \neg l_0, l_1, ..., l_m, {\sim}l_{m+1}, ..., {\sim}l_n)$
- Truth condition of contrapositive: $(\leftarrow \neg l_0, {\sim}\neg l_1, ..., {\sim}\neg l_m, \neg l_{m+1}, ..., \neg l_n)$
- Falsity condition of contrapositive: $(\leftarrow \neg l_0, l_1, ..., l_m, {\sim}l_{m+1}, ..., {\sim}l_n)$

As mentioned when first introducing $\rightarrow_{si}$, its falsity condition is the same as that of $\rightarrow_{cmi}$; the falsity conditions of its non-contrapositive and contrapositive halves are identical and redundant.

$\neg(l_1 \wedge ... \wedge l_m \wedge -\neg l_{m+1} \wedge ... \wedge -\neg l_n) \vee l_0$ can be implemented as:

- Truth condition: $(l_0 \leftarrow {\sim}\neg l_1, ..., {\sim}\neg l_m, \neg l_{m+1}, ..., \neg l_n)$
- Falsity condition: $(\leftarrow \neg l_0, l_1, ..., l_m, {\sim}l_{m+1}, ..., {\sim}l_n)$

Finally, we present an example of using some of the implication operators we have examined, and implementing it.

Example 10. Suppose we are in front of a train track, and trying to decide whether to cross the track. Let t represent knowing that a train is coming, and c represent deciding to cross. The following considerations need to be made when modelling this problem:

- We may lack access to any information on the train schedule, so t may take on the value of **N**. We may also have access to multiple conflicting versions of the train schedule, some of which may be wrong or outdated, so t may also take on the value of **B**.
- We cannot decide to simultaneously cross and not cross, so c cannot take on the value of **B**, necessitating the constraint $c \wedge \neg c \rightarrow_t$. Using another implication operator is unnecessary here because an integrity constraint using $\rightarrow_t$ is already unsatisfiable if its body is true.
- If we do not have a compelling reason that we definitely should cross, then it is better to be safe and not cross; ${\sim}c \rightarrow_t \neg c$. We can use a different implication operator here, but doing so ultimately only gives us trivialities like ${\sim}c \wedge c \rightarrow_t$ or redundancies with $c \wedge \neg c \rightarrow_t$.

- If we have a version of the train schedule saying that no train is coming, and also do not have any conflicting versions of the train schedules saying a train is coming, then we conclude that it is safe to cross. However, the "counter-example" of being absolutely sure that no train is coming but still deciding to not cross is not very unreasonable, as we may have some other reason to not cross, so we do not need to rule out this counter-example. Similarly, if we look at the contrapositive of this rule, it says that if we did not cross, we can conclude that there must have been a train coming or that we must have been unable to verify that no train was coming, which is not necessarily true due to the aforementioned other potential reason to not cross; so we do not need to enforce the contrapositive here. Therefore, we should write this rule as $\neg t \wedge \sim t \rightarrow_t c$.

- If we know the train is coming, then we must not cross. The counter-example of knowing the train is coming but still crossing is lethal, so it must be eliminated. Therefore, this rule should at least be $t \rightarrow_{cmi} \neg c$. When we look at its contrapositive, we reason that if we crossed, that means we knew no train was coming, which must be true in our model of the problem, so we can incorporate the contrapositive and have this rule be $t \rightarrow_{si} \neg c$.

- If we cannot verify that the train is not coming, then we must not cross. Similarly to the previous point, the counter-example of not knowing the train is not coming but still crossing is lethal and must be eliminated. Also similarly to the previous point, the contrapositive of if we crossed then we did not have any information telling us the train was coming is true. Therefore, this rule should also be $\sim\neg t \rightarrow_{si} \neg c$.

When converting the above Belnap formulas into $\mathbf{ASP^4}$ rules, some of the resulting rules are redundant with each other. Overall, we have the program $P = \{(\leftarrow c, \neg c), (\neg c \leftarrow \sim c), (c \leftarrow \neg t, \sim t), (\neg c \leftarrow t), (\leftarrow c, t), (\neg c \leftarrow \sim\neg t), (\leftarrow c, \sim\neg t)\}$. Now we add in some facts about our knowledge of the train schedule to see what conclusions we can make.

- Let $P_1 = P \cup \{(t \leftarrow)\}$, meaning that we know the train is coming and also has no information saying the train is not coming. Then our only stable model is $\{t, \neg c\}$, meaning we should not cross.

- Let $P_2 = P \cup \{(\neg t \leftarrow)\}$, meaning that we know the train is not coming and also has no information saying the train is coming. Then our only stable model is $\{\neg t, c\}$, meaning we should cross.

- Let $P_3 = P \cup \{(t \leftarrow), (\neg t \leftarrow)\}$, meaning that we have conflicting information on whether the train is coming. Then our only stable model is $\{t, \neg t, \neg c\}$, meaning we should not cross.

- Let $P_4 = P \cup \emptyset$, meaning that we have no information on whether the train is coming. Then our only stable model is $\{\neg c\}$, meaning we should not cross.

We can see that the only time we know we can safely cross is when we are absolutely sure that the train is not coming, and that we know not to cross when facing conflicting information on whether the train is coming and the lack of information.

Chapter 5

Conclusion and Further Research

Now that we are finished with presenting our major findings, we will summarize what we have done and present opportunities for further research.

This thesis, and paraconsistent logic as a whole, is motivated by the desire to avoid the principle of explosion $A \wedge \neg A \rightarrow B$ in classical logic, since otherwise the presence of a contradiction would cause every possible formula to become true. In answer set programming with classical negation, this is handled by rules like $(b \leftarrow a, \neg a)$ or $(\leftarrow a, \neg a)$, which artificially enforce the principle of explosion, resulting in some awkwardness such as the language being a three-valued logic that lacks clearly-defined intuitive meaning when a model contains neither an atom nor its negation. Thus we instead define a four-valued version of ASP, showing that Belnap logic is a more intuitive basis for logic programming that uses both classical negation and default negation. We have shown that the conflation operator in Belnap logic has an intuitive meaning corresponding to default negation, as in $\sim a = -\neg a$. We then examined a number of implication operators in Belnap logic, culminating in the creation of a general unidirectional implication operator that generalizes the notion of modus ponens in a four valued context, which are implemented in ASP as a combination of normal rules and integrity constraints. The primary contributions of this thesis are the definition of ASP-like default negation in Belnap logic, and the general unidirectional implication in Belnap logic that encompasses many previous Belnap implications and can be easily implemented in ASP.

This thesis only considers basic ASP and ground atoms, but predicates with variables can simply be replaced by their groundings, so extending from propositional to first-order logic should make no difference as long as the domain is finite. Similarly, certain extensions of ASP are reducible to basic ASP, such as cardinality rules and weighted rules, so the results of this thesis should also be applicable to ASP with these extensions. Disjunctive logic

programming, on the other hand, is a true extension of ASP that we briefly examined in the previous chapter, so extending four-valued ASP with disjunction is an obvious direction of further research. An area of particular interest is the effect of moving d-literals between the head and body of a rule on a program's stable models. For example, the program $\{(a, \sim a \leftarrow)\}$ has two stable models, but moving the second a into the rule body to produce $\{(a \leftarrow a)\}$ eliminates one of the stable models.

Another concession that must be made is the fact that Belnap logic is not immune to the principle of explosion either. Belnap's original formulation of it [2], which does not contain the conflation, gullibility, or consensus operators, is immune to the principle of explosion due to not being able to form contradictions or tautologies, but the language is also not functionally complete. Adding the three missing operators makes the language functionally complete, and while the classical principle of explosion $A \wedge \neg A \rightarrow B$ is not valid, a form of the principle of explosion like $A \wedge -\neg A \rightarrow B$ still takes on the value $\mathbf{T}$ in all interpretations for all general unidirectional implications. That is because $A \wedge -\neg A$ is a "hypercontradiction", like saying "A is simultaneously true and not true", which is impossible in Belnap logic, as opposed to saying "A is simultaneously true and false", which is possible. This is not a problem in $\mathbf{ASP^4}$ because an interpretation is a set of literals, not a set of d-literals, so we cannot have something like $\sim a$ as an explicit assertion in our knowledge base; a conjunction like $(a, \sim a)$ is unsatisfiable by any interpretation. However, another potential way of resolving "hypercontradictions" is to extend Belnap logic in the same way that Belnap logic extends classical logic, by constructing a set of truth values that are subsets of the power set of $\mathbf{4}$. This is a topic explored by Shramko and Wansing [19], who presented a sixteen-valued logic among others; it may be possible to generalize $\mathbf{ASP^4}$ to arbitrarily higher-ordered multilattices.

Finally, other approaches to non-classical ASP can be considered, such as fuzzy logic and probabilistic logic, in works like Janssen et al. [14] (fuzzy ASP) and Finger and de Morais [7] (probabilistic ASP). While these approaches involve drastically different underlying machinery to function, it may be possible to convert two-valued fuzzy ASP into four-valued fuzzy ASP in the same way that two-valued normal ASP is converted into four-valued ASP, and similarly for probabilistic ASP. However, doing so may also require the development of four-valued fuzzy or probabilistic logic beforehand.

Bibliography

[1] Ofer Arieli and Arnon Avron. The value of the four values. *Artificial Intelligence*, 102:97–142, 1998.

[2] Nuel D. Belnap, Jr. A useful four-valued logic: How a computer should think. In *Entailment: The Logic of Relevance and Necessity*, volume 2, pages 506–541. Princeton University Press, 1992.

[3] Glenn Bruns and Michael Huth. Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. *ACM Transactions on Information and System Security*, 14(1), 2011.

[4] Chen Chen and Zuoquan Lin. Restricted four-valued semantics for answer set programming. *PRICAI 2016: Trends in Artificial Intelligence*, pages 68–79, 2016.

[5] Wolfgang Faber and Nicola Leone. On the complexity of answer set programming with aggregates. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, page 97–109. Springer-Verlag, 2007.

[6] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich C. Teppan. Industrial applications of answer set programming. *Künstliche Intelligenz*, 32:165–176, 2018.

[7] Marcelo Finger and Eduardo Menezes de Morais. Probabilistic answer set programming. In *Proceedings of the 2013 Brazilian Conference on Intelligent Systems*, pages 150–156. IEEE Computer Society, 2013.

[8] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[9] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 8 1991.

[10] Matthew L. Ginsberg. Multivalued logics: A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4:265–316, 1988.

[11] Allen P. Hazen, Francis Jeffry Pelletier, and Geoff Sutcliffe. Making Belnap's "useful four-valued logic" useful. In *The Thirty-First International Florida Artificial Intelligence Research Society Conference*. Association for the Advancement of Artificial Intelligence, 2018.

[12] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[13] Tomi Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. *Logic Programming and Nonmotonic Reasoning*, 1:93–106, 2001.

[14] Jeroen Janssen, Steven Schockaert, Dirk Vermeir, and Martine De Cock. *Answer Set Programming For Continuous Domains: A Fuzzy Logic Approach*, volume 5 of *Atlantis Computational Intelligence Systems*. Atlantis Press, 2012.

[15] Hitoshi Omori and Katsuhiko Sano. Generalizing functional completeness in Belnap-Dunn logic. *Studia Logica*, 103:883–917, 2015.

[16] Graham Priest. *Paraconsistent Logic*, pages 287–393. Springer Netherlands, 2002.

[17] R. Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Springer Science+Business Media, 1978.

[18] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[19] Yaroslav Shramko and Heinrich Wansing. *Truth and Falsehood: An Inquiry into Generalized Logical Values*. Springer Science+Business Media, 2011.

[20] Hudson Turner. Strong equivalence made easy: Nested expressions and weighted constraints. *Theory and Practice of Logic Programming*, 3:609–622, 2003.

[21] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for logic programs. *Journal of the ACM*, 38(3):620–650, 1991.