Foundations of Fuzzy Answer Set Programming

Dissertation submitted in fulfillment of the requirements for the degree of Doctor of Science: Computer Science

Jeroen Janssen

June 2011

Promotors: Prof. Dr. Dirk Vermeir (VUB)
           Prof. Dr. Martine De Cock (UGent)
           Dr. Steven Schockaert (UGent)
Abstract

Answer set programming (ASP) is a declarative language tailored towards solving combinatorial optimization problems. It has been successfully applied to e.g. planning problems, configuration and verification of software, diagnosis and database repairs. However, ASP is not directly suitable for modeling problems with continuous domains. Such problems occur naturally in diverse fields such as the design of gas and electricity networks, computer vision and investment portfolios. To overcome this problem we study FASP, a combination of ASP with fuzzy logic – a class of many-valued logics that can handle continuity. In this thesis we focus on the following issues:

1. An important question when modeling continuous optimization problems is how we should handle overconstrained problems, i.e. problems that have no solutions. In many cases we can opt to accept an imperfect solution, i.e. a solution that does not satisfy all the stated rules (constraints). However, this leads to the question: what imperfect solutions should we choose? We investigate this question and improve upon the state-of-the-art by proposing an approach based on aggregation functions.

2. Users of a programming language often want a rich language that is easy to model in. However, implementers and theoreticians prefer a small language that is easy to implement and reason about. We create a bridge between these two desires by proposing a small core language for FASP and by showing that this language is capable of expressing many of its common extensions such as constraints, monotonically decreasing functions, aggregators, S-implicators and classical negation.

3. A well-known technique for solving ASP consists of translating a program $P$ to a propositional theory whose models exactly correspond to the answer sets of $P$. We show how this technique can be generalized to FASP, paving the
ABSTRACT

way to implement efficient fuzzy answer set solvers that can take advantage of existing fuzzy reasoners.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>9</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>11</td>
</tr>
<tr>
<td>1.1  Answer Set Programming</td>
<td>14</td>
</tr>
<tr>
<td>1.2  Fuzzy Logic</td>
<td>17</td>
</tr>
<tr>
<td>1.3  Overview</td>
<td>20</td>
</tr>
<tr>
<td><strong>2 Preliminaries</strong></td>
<td>23</td>
</tr>
<tr>
<td>2.1  Order Theory</td>
<td>23</td>
</tr>
<tr>
<td>2.2  Answer Set Programming</td>
<td>26</td>
</tr>
<tr>
<td>2.2.1 Definitions</td>
<td>26</td>
</tr>
<tr>
<td>2.2.2 Classical negation vs negation-as-failure</td>
<td>34</td>
</tr>
<tr>
<td>2.2.3 Links to SAT</td>
<td>36</td>
</tr>
<tr>
<td>2.3  Fuzzy Logic</td>
<td>40</td>
</tr>
<tr>
<td>2.3.1 Fuzzy Sets</td>
<td>40</td>
</tr>
<tr>
<td>2.3.2 Logical Operators on Bounded Lattices</td>
<td>41</td>
</tr>
<tr>
<td>2.3.3 Formal Fuzzy Logics</td>
<td>48</td>
</tr>
<tr>
<td><strong>3 Fuzzy Answer Set Programming</strong></td>
<td>51</td>
</tr>
<tr>
<td>3.1  Introduction</td>
<td>51</td>
</tr>
<tr>
<td>3.2  Definitions</td>
<td>52</td>
</tr>
<tr>
<td>3.2.1 Language</td>
<td>52</td>
</tr>
<tr>
<td>3.2.2 Semantics</td>
<td>54</td>
</tr>
<tr>
<td>3.3  Example: Fuzzy Graph Coloring</td>
<td>64</td>
</tr>
</tbody>
</table>
## CONTENTS

4 Aggregated Fuzzy Answer Set Programming 67
  4.1 Introduction ............................................. 67
  4.2 Aggregated Fuzzy Answer Set Programming .......... 69
    4.2.1 Models ............................................. 69
    4.2.2 Answer Sets ....................................... 77
  4.3 Illustrative Example ................................... 95
  4.4 Relationship to Existing Approaches ............... 104
    4.4.1 Fuzzy and Many-Valued Logic Programming Without Partial Rule Satisfaction ............................................. 105
    4.4.2 Weighted Rule Satisfaction Approaches .............. 106
    4.4.3 Van Nieuwenborgh et al. .......................... 110
    4.4.4 Valued Constraint Satisfaction Problems ........... 114
    4.4.5 Answer Set Optimization ............................ 114
  4.5 Summary ................................................ 115

5 Core Fuzzy Answer Set Programming 117
  5.1 Introduction ............................................. 117
  5.2 The FASP Core Language .................................. 119
  5.3 Constraints .............................................. 120
    5.3.1 Implementing Constraints .......................... 121
    5.3.2 Locking the Truth Value ............................ 127
  5.4 Monotonically Decreasing Functions .................. 128
  5.5 Aggregators ............................................. 132
  5.6 S-implicators ........................................... 140
  5.7 Strong Negation ......................................... 145
  5.8 Summary ................................................ 148
  5.A Proofs .................................................. 149
  5.B Diagram ................................................. 156

6 Reducing FASP to Fuzzy SAT 157
  6.1 Introduction ............................................. 157
  6.2 Completion of FASP Programs ............................ 159
  6.3 Loop Formulas .......................................... 165
  6.4 Example: the ATM location selection problem ........ 172
  6.5 Discussion .............................................. 180
  6.6 Summary ................................................. 182
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusions</td>
<td>183</td>
</tr>
<tr>
<td>Samenvatting</td>
<td>187</td>
</tr>
<tr>
<td>List of Publications</td>
<td>191</td>
</tr>
<tr>
<td>Bibliography</td>
<td>195</td>
</tr>
<tr>
<td>List of Symbols</td>
<td>213</td>
</tr>
<tr>
<td>Index</td>
<td>217</td>
</tr>
</tbody>
</table>
Acknowledgment

Writing a thesis and doing research is not a one-man-effort and hence I am indebted to many people. First of all I want to thank my three promotors Dirk, Martine and Steven for their help and guidance and for giving me the opportunity to spend these four years on interesting research. Without all of you this thesis simply would not exist.

I would also like to thank my PhD jury for providing me with helpful remarks and comments that considerably improved the final version of this text.

Next to my promotors and jury I should also thank FWO-Vlaanderen for funding the research project I have worked on in these four years.

Furthermore I would like to thank Umberto Straccia for hosting me during my four-week visit at ISTI-CNR. I had a terrific time there and it was a very interesting experience.

I also want to thank my parents, grandparents and sister Helena for their everlasting support and understanding throughout the years. I may not always say it in so many words, but you all mean a great deal to me.

A big thank you should furthermore go to my friends and (ex-)bandmates from According to Garp (ex-Consensus) and Lightbulb Inferno for understanding that the time I could invest in them was limited when deadlines were tight and for the ever-needed stress relief I always found while playing with them. I would especially like to thank Katrijn for designing the cover of this book, but all of you guys rock!

I am also very grateful to all the people in the department of computer science in Brussels and the department of applied mathematics and computer science in Ghent. There are too many to name them all, but big thanks especially go to my former and present office mates Tom, Mike, Koen, Klaas, Luis, Viktor & Etienne for the nice atmosphere in the office; to Patricia for organizing all the paintball events, game nights, summer outings, etc. and for the help and company during my stay in Tacoma; to Gilles for always being enthusiastic and open to silly ideas; to Kim for the interesting talks on all things computer science and non-computer science; to
ACKNOWLEDGMENT

Timur for the fun we had during the road trip in Tacoma and finally to all others for creating such a great environment to work in.

Moreover I would like to thank my computer science co-graduates Bert, Bart, Davy, Femke O, Femke DB, Stéphanie, Pascal and Jeroen for the many nights out and evenings of playing poker while whining about the life of PhD students. I might be the first to finish my thesis, but I look forward to toasting at your PhD parties in the coming years!

Last, but – as the cliché goes – certainly not least, I’d like to thank Virginie for understanding that I did not have much time for her while writing this thesis and for cheering me up and supporting me when things got rough. It was much appreciated!
Language is one of the most important tools that exist. It allows humans to communicate efficiently and to transfer knowledge between generations. According to Benjamin Whorf, language even shapes views and influences thoughts. Unfortunately, while human language is useful for communication between humans, it is not as efficient for communicating with our modern day devices. Therefore, ever since the rise of computers, the need has grown for languages that enable us to tell these machines what we expect them to do. Such languages are called *programming languages*. Their foundations can be dated back to the 1800s, where Joseph Marie Jacquard used punched cards to encode cloth patterns for his textile machine, called the “Jacquard loom”. Charles Babbage improved on this idea when designing his “analytical engine” by allowing the machine to be reprogrammed using punched cards. Hence, instead of merely using the punched cards as data, the analytical engine could perform arbitrary computations that were encoded in the


CHAPTER 1. INTRODUCTION

punched cards. As such, we can consider this the first real programmable machine.

The 1940s witnessed the birth of the first machines that resemble our modern day electrical computers. Initially these machines were programmed using patched cables that encoded specific machine-instructions. Input and output was done using punched cards. Since the (re)programming of these computers was a laborious task requiring many people, the idea arose to unify programs with data and store them in memory. This led to the creation of stored-program computers, such as EDVAC (Electronic Discrete Variable Automatic Computer, successor of ENIAC\(^4\)) and SSEM (Small-Scale Experimental Machine\(^5\)). Contrary to the earlier designs, these systems could read programs from punched cards and store them in memory, thereby making (re)programming them as easy as inserting a new stack of punched cards.

While the creation of stored-program computers eliminated the physical burden of programming, the mental activity required was still high due to the use of machine-specific codes. These low-level languages allowed the programmer to greatly optimize their programs for specific machines, but also made it hard to express complex problems due to their poor readability and the fact that they are far removed from natural language. To solve these problems, so called “higher-level” programming languages were developed. One of the first such languages was “Plankalkül” (“planning calculus”). It was described by Konrad Zuse in 1943 \([8,183,184]\), but was only implemented in 1998\(^6\) and independently in 2000 \([146]\). In the 1950s the first high-level programming languages with working implementations were created. The most important among them are Fortran (Formula Translator), COBOL (Common Business Oriented Language) and LISP (List Processor). Fortran was mostly oriented towards scientific computing, COBOL towards business and finance administration systems and LISP towards artificial intelligence systems. Though they focused on different domains, each of them could be used to write general purpose programs. In 1960 computer scientists from Europe and the United States developed a new language, called ALGOL 60 (algorithmic language). Though the language, and especially its formulation, contained many innovations, it did not gain widespread use. Its ideas influenced many of the languages created later, however.

In the 1960s through the 1970s many of the major programming language paradigms that are still in use today were developed. For example, Simula (end of the


(1960s) was the first language supporting object-oriented programming, Smalltalk (mid 1970s) the first fully object-oriented programming language, Prolog (1972) the first logic programming language and ML (1973) the first statically typed functional programming language. Most of our modern languages have clear influences from these languages and can thus be categorized in one of the associated paradigms. Other important programming languages created in this period were Logo (1968, a LISP offspring developed for teaching), PASCAL (1970, an ALGOL offspring) and C (1972, a systems programming language).

The 1980s mostly saw the creation of languages that recombined and improved upon the ideas from the paradigms and languages invented in the 1960s and 1970s. For example, C++ (1980) combined C with object-oriented programming, Objective-C (1983) combined C with Smalltalk-style messaging and object-oriented programming and Erlang (1986) combined functional programming with provisions for programming distributed systems. Next to these languages, a subparadigm of functional programming, called purely functional programming, was also created. Notable examples of the latter are Miranda (1985) and Haskell (1990).

In the 1990s general interest arose in programming languages that improve programmer productivity, so called rapid development languages. Most of these languages incorporated object-oriented features or were fully object-oriented and had garbage-collection utilities to relieve the programmer of manual memory management. Examples are Python (1991), Visual Basic (1991), Ruby (1993), Java (1995) and Delphi (1995). The rise of the internet also spurred the development of scripting languages such as JavaScript (1995) and PHP (1995), which enabled the fast creation of interactive and dynamic websites. Due to the occurrence of computers with multiple cores, in the 2000s languages tailored for these machines were created, such as Clojure (2007) and Go (2009).

All programming languages mentioned above are general-purpose programming languages. This means they can be used to write software for many different application domains. While such languages have the advantage of only needing to learn one language for writing a variety of software, most of these languages do not support special constructs for specific application domains. This makes the translation of the requirements of a new software package into code much harder. Domain-specific languages are languages that are tailored towards one specific problem domain. Notable examples are regular expressions for handling text, SQL for describing database.
interactions and Yacc for creating compiler front-ends. Since the 1990s interest in domain-specific languages has increased. In fact, a new programming methodology, called language-oriented programming, has arisen that proposes to create a new language describing the domain first, and then use this language to write the final program [178].

Answer set programming is a declarative domain-specific language tailored towards solving combinatorial optimization problems. It has roots in logic programming and non-monotonic reasoning. In this thesis, we study a new domain-specific language, called fuzzy answer set programming, that is aimed towards solving continuous optimization problems. It combines answer set programming with fuzzy logic – a mathematical logic which can describe continuous concepts in an intuitive manner. In the next two sections we describe the history and general idea of these two cornerstones in more detail.

1.1 Answer Set Programming

To create systems that are capable of human-like reasoning, we need languages that are tailored towards representing knowledge and a method for reasoning over this knowledge. An idea that immediately comes to mind is to use logic to describe our knowledge and use model-finding algorithms (e.g. SAT solving) for the reasoning part. One of the limitations of classical logic when mimicking human reasoning is that it works monotonically: when new knowledge is added, the set of conclusions that can be inferred using classical logic grows. In contrast humans constantly revise their knowledge when new information becomes available. For example if we know that Pingu is a bird, we assume that he can fly. If we afterwards are told that he is a penguin, however, we need to revise our belief, as we know that penguins can’t fly.

During the last decades, researchers have studied non-monotonic logics as a way to overcome this limitation of classical logic. Several such logics have been proposed, such as circumscription [124], default logic [18, 116, 143, 145], auto-epistemic logic [127], non-monotonic modal logics in general [125] and logic programming with negation-as-failure [27, 66, 172]. In this thesis, we will focus on the latter.

Non-monotonicity in logic programming is obtained using a special construct called negation-as-failure, which is denoted as “not $\alpha$” and intuitively means that the negation of $\alpha$ is true when we fail to derive $\alpha$. Defining the semantics of this construct proved to be a challenge, however. The most important proposed
definitions are the Clark completion [27], the stable model semantics [66] and the well-founded semantics [172]. The stable model semantics refine the conclusions of the Clark completion in the presence of positive mutual dependencies between predicates [54]. The well-founded semantics on the other hand are more cautious in their conclusions than both the stable models and the Clark completion when there are mutual dependencies between predicates with the negation-as-failure construct. It has been shown that the well-founded semantics are an approximation of the stable model semantics [7]. A lot of research has also been devoted to the relationships between stable model semantics and other non-monotonic logic formalisms. For a good overview of these links we refer the reader to [5]. Attention has also been given to studying extensions of these semantics. In [100] the stable model semantics is extended to programs with disjunctions, which has been shown to make the language capable of modeling a larger class of problems [51]. Another important extension is the addition of a second form of negation, called classical negation [67]. Whereas negation-as-failure denotes that the negation follows from a failure to derive a proof term, classical negation denotes that the negation of the proof term can explicitly be derived.

At the end of the 1990s researchers began to notice that the stable model semantics gives rise to a certain logic programming paradigm that is different from the proof-derivation based approach of languages such as Prolog [121,135]. Vladimir Lifschitz named this new paradigm “answer set programming” (ASP) in [98,99]. The basic idea of answer set programming is that a programmer translates a certain problem into an answer set program (a logic program under the stable model semantics) such that the answer sets (stable models) of the program correspond to the problem solutions. This program is then given as input to an answer set solver which computes the answer sets of the program. This solver has three possible outputs:

1. No answer set exists. In this case, the modeled problem does not have a solution.

2. One answer set exists. In this case, the answer set corresponds to the solution of the modeled problem.

3. Multiple answer sets exist. In this case, the modeled problem has multiple solutions. The user can ask the answer set solver to compute all answer sets, or only a single one if this suffices.

For example, consider the problem of finding a large clique, i.e. a subset $V$ of an
undirected graph such that: (i) there is an edge between every pair of vertices in $V$; (ii) the cardinality of $V$ is greater than or equal to a given $l$. If we take $l = 3$, for example, we can solve this using the following answer set program $P_{\text{clique}}$ (from [99])$^9$:

\begin{align*}
   \text{in}(X) & \leftarrow \text{not out}(X) \quad (1.1) \\
   \text{out}(X) & \leftarrow \text{not in}(X) \quad (1.2) \\
   \text{sizeOk} & \leftarrow \text{in}(X), \text{in}(Y), \text{in}(Z), X \neq Y, X \neq Z, Y \neq Z \quad (1.3) \\
   \text{joined}(X, Y) & \leftarrow \text{edge}(X, Y) \quad (1.4) \\
   \text{joined}(X, Y) & \leftarrow \text{edge}(Y, X) \quad (1.5) \\
   & \leftarrow \text{in}(X), \text{in}(Y), X \neq Y, \text{not joined}(X, Y) \quad (1.6) \\
   & \leftarrow \text{not sizeOk} \quad (1.7)
\end{align*}

In this program, rules (1.1) and (1.2) state that a vertex of the graph should either be in the clique or out of the clique. Rule (1.3) ensures that $\text{sizeOk}$ is only true when there are at least three vertices in the clique. The (1.4) and (1.5) rules declare that two vertices are joined if there is an edge between them. Last, rules (1.6) and (1.7) are constraint rules. Intuitively such rules remove solutions that make the right-hand side true. In the case of $P_{\text{clique}}$, rule (1.6) prohibits solutions where two vertices in the clique are not joined, whereas rule (1.7) stops solutions that do not have the right clique size.

Given the above program, the ASP programmer does not compile or run it using an interpreter, but solves it by means of an answer set solver. For this solver to work, the above program also needs to be supplemented with fact rules describing a graph. For example, consider the graph $G$ with vertex set $V = \{v_1, v_2, v_3, v_4, v_5\}$ and edge set $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_4, v_5)\}$ that is depicted in Figure 1.1. This graph is encoded using the following set $F_{\text{clique}}$ of ASP rules:

\begin{align*}
   \text{edge}(v_1, v_2) & \leftarrow \\
   \text{edge}(v_1, v_3) & \leftarrow \\
   \text{edge}(v_2, v_3) & \leftarrow \\
   \text{edge}(v_4, v_5) & \leftarrow
\end{align*}

$^9$Note that existing answer set solvers support an extension that allows to write the combination of rules (1.1)–(1.3) and (1.7) as the single rule “$3 \{\text{in}(X)\}$”. Since we do not consider these extensions in this thesis, we opted to remove this syntactic sugar.
1.2. FUZZY LOGIC

If we hand program $P_{\text{clique}} \cup F_{\text{clique}}$ as input to an ASP solver, it computes an answer set of the program. For our above program the resulting answer set is

$$A = \{\text{edge}(v_1, v_2), \text{edge}(v_1, v_3), \text{edge}(v_2, v_3), \text{edge}(v_4, v_5), \text{in}(v_1), \text{in}(v_2), \text{in}(v_3), \text{out}(v_4), \text{out}(v_5), \text{sizeOk}, \text{joined}(v_1, v_2), \text{joined}(v_2, v_1), \text{joined}(v_1, v_3), \text{joined}(v_3, v_1), \text{joined}(v_2, v_3), \text{joined}(v_3, v_2), \text{joined}(v_4, v_5), \text{joined}(v_5, v_4)\}$$

As expected, this corresponds to the clique $\{v_1, v_2, v_3\}$ of size 3 that exists for $G$. Due to the creation of (relatively) fast answer set solvers such as Smodels [156] and DLV [53], ASP was successfully applied to problems occurring in planning [49,50,99], configuration and verification [157,158], diagnosis [48], database repairs [2], game theory [39] and bio-informatics [6]. Furthermore it has been used to provide decision support for the space shuttle [136].

1.2 Fuzzy Logic

**History** Besides monotonicity, there are other aspects of classical logic that have been questioned throughout history. One of the most important ones is the principle of bivalence, i.e. the fact that propositions are either true or false. This principle was already a source of controversy among the old Greeks. While the school of Chrisippus defended it strongly, it was questioned by the Epicureans [44]. The main motivation for the rejection of bivalence was due to the perceived incapability of classical logic to handle propositions that refer to future contingencies. For example, consider the following proposition $p$: "Belgium will have a new government tomorrow". When stating this proposition, we cannot say that $p$ is false, since this would mean that this is impossible. Likewise we cannot say that $p$ is true, since this
CHAPTER 1. INTRODUCTION

would mean that this is necessary. Classical logic dictates that \( p \) should be either true or false, however. In the 4th century BC this already led Aristotle to believe that a third logical status of propositions exists [62]. While there has been some interest in examining this third truth value during the Middle Ages, it was only at the turn of the 19th century that more serious attempts were constructed. Notable efforts are the ones by Hugh MacColl, Charles S. Peirce and Nicolai A. Vasil’ev, who grouped propositions as either affirmative, negative and indifferent using considerations dealing with temporal or modal concepts [62]. The era of many-valued logic only started with the works of Jan Łukasiewicz [109] and Emil L. Post [142] in 1920, however. They built successful formulations of many-valued logic by adapting the truth-table method that was applied to classical logic by Frege, Peirce and others [62]. Łukasiewicz initially considered a logic with three truth values, where next to true and false propositions could be possible. In 1922 he generalized this work to a logic with an infinite number of truth values [17]. Post applied his many-valued logic to problems of the representability of functions\(^{10}\).

In the beginning of the 1930s Gödel used multiple truth values to understand intuitionistic logic [69]. This led to the family of Gödel systems, which were extended to infinite truth values by Jaskowski in 1936 [85]. In the late 1930s many-valued logics were applied to paradoxes by Bochvar [16] and to partial functions and relations by Kleene [89].

In the 1950s and 1960s important work on the completeness of infinite-valued logics was done. Chang showed the relations between algebraic structures and many-valued systems [23] and proved a completeness result for infinite-valued Łukasiewicz logic [24]. Dummett proved a completeness result for infinite-valued Gödel logic [47] and McNaughton created an analytical characterization of the class of truth degree functions that are definable in infinite-valued Łukasiewicz logic [126]. This time period furthermore saw proofs of the completeness of first-order infinite-valued Łukasiewicz logic [24, 74] and Gödel logic [75]. It was also shown that the former system is not (recursively) axiomatizable [149].

Of significant importance for this thesis is the introduction of fuzzy sets by Lotfi Zadeh in 1965 [182]. Fuzzy set theory is a generalization of classical set theory with many-valued characteristic functions. Its introduction was motivated by the incapability of sets to model concepts that are not properly delineated. For example, if we say that the sky is blue, we do not mean that the sky has exactly the RGB value 0000FF, but rather that the color we perceive of the sky is similar to this

1.2. FUZZY LOGIC

RGB value. The question then arises how we can define the set of colors that are similar to 0000FF. It is easy to find examples that clearly are contained in this set (e.g. a light shade of blue) and examples that clearly are not contained in this set (e.g. red), but there are also colors that live on the boundary of these two sets, such as "greenish blue". Fuzzy sets take a graded approach: an element is not necessarily either fully contained in a set or not contained at all, but rather it can be contained to a certain degree. For example, for colors such as "greenish blue" we could say that it belongs to the set of colors similar to blue to a degree of 0.8, whereas red is contained in this set to a degree of 0 and a light shade of blue to a degree of 1. Note that fuzzy sets can thus be seen as mappings from a universe of discourse X to values in [0, 1].

From the 1970s and onwards the study of the foundations of fuzzy set theory was one of the driving motivations behind further research into many-valued logics. Notable works include the extension of many-valued logics with a graded notion of inference and entailment by Pavelka [138], the first complexity results of infinite-valued Łukasiewicz logic in [144] and a detailed study of infinite-valued logics based on triangular norms [72]. The latter systems have the same importance to fuzzy set theory as classical logic has to set theory. Therefore they are often called fuzzy logics.

The intuition of truth values Łukasiewicz proposed his many-valued logic as a method for modeling sentences referring to future contingencies, such as proposition p introduced above. His main idea was to add a third truth value possible and assign this value to such propositions, thereby rejecting the bivalence property. However, a couple of years after the publication of the works by Łukasiewicz, researchers in the foundations of probability theory became aware that probabilities differ from truth-values [44]. De Finetti pointed out that uncertainty is a meta-concept with respect to truth degrees and that it is still based on the idea of bivalence:

"Even if, in itself, a proposition cannot be but true or false, it may occur that a given person does not know the answer, at least at a given moment. Hence for this person, there is a third attitude in front of a proposition. This third attitude does not correspond to a third truth-value distinct from yes or no, but to the doubt between the yes and the no (as people, who, due to incomplete or undecipherable information, appear as of unknown sex in a given statistics, do not constitute a third sex. They only form the group of people whose sex is unknown.)" [38]
CHAPTER 1. INTRODUCTION

(translation from [44]).

De Finetti furthermore pointed out that many-valued logics do not reject the principle of bivalence, but are only compact representations of several ordinary propositions [38]. Gonseth moreover remarked that Łukasiewicz’ original interpretation of the third truth value neglects the mutual dependence of possible propositions [62]. For example consider proposition $p$ from above. If $p$ is possible, so is $\neg p$. In the original Łukasiewicz system this leads us to conclude that $p \land \neg p$ is possible. However, this runs counter to our intuition as $p \land \neg p$ should always be false, independent from the truth value $p$.

The above shows that many-valued logics have nothing to do with uncertainty or probability, but instead deal with capturing the idea of partial truth, i.e. the fact that the compatibility of an object w.r.t. a certain concept is a matter of degree. Hence, the fuzzy answer set programming framework we create in this thesis does not capture uncertainty or probability, but deals with this graded truth.

1.3 Overview

While ASP allows to model combinatorial optimization problems in a concise and declarative manner, it is not suitable for expressing continuous optimization problems. These problems arise in many different domains, such as scheduling and designing gas and electricity networks [137], computer vision [76] and investment portfolios [73]. Fuzzy answer set programming (FASP) combines ASP and fuzzy logic. The resulting language is capable of expressing continuous optimization problems in the same declarative manner as ASP allows the modeling of combinatorial optimization problems.

In this thesis we study the semantics and properties of extensions for FASP, as well as some implementation methods. We begin by recalling some preliminary notions regarding answer set programming and fuzzy logic in Chapter 2, followed by an introduction to fuzzy answer set programming in Chapter 3.

In Chapter 4 we investigate the use of allowing FASP rules to be partially fulfilled. This is often useful when modeling problems for which no perfect solution exists, or for which we are only interested in the best solution that can be found in a given time frame. Previous proposals for FASP supported partial rule satisfaction with weights. This is not ideal, however, as it puts the burden of finding the right weights on the programmer, a task that might not always be so straightforward. We improve upon these approaches by making the weights of a rule dynamic and by
1.3. OVERVIEW

aggregating the value a prospective solution attaches to these weights into a global suitability score.

While users of a programming language often want the language to contain many features, implementers and theoreticians prefer a small, core language that is easy to implement and reason about. In Chapter 5 we create a bridge between these two wishes by proposing a simple core language for FASP that we show is capable of expressing many of its common extensions.

In Chapter 6 we research whether FASP can be translated to a fuzzy SAT problem. This provides the theoretical foundations for the creation of FASP solvers that rely on fuzzy SAT solving techniques such as mixed integer programming and other forms of mathematical programming. However, to build a real solver additional research on the efficient grounding of programs and the translation of functions other than t-norms is also needed. Our results are furthermore interesting on a theoretical level, as they show us where the use of FASP over fuzzy SAT is advantageous.

The results in this thesis have been published, or submitted for publication, in international journals and the proceedings of international conferences with peer review. Specifically, the framework defined in Chapter 4 was introduced in [83] and studied in more detail in [79]; the core FASP language defined in Chapter 5 was introduced in [80]; a preliminary version of the reduction to fuzzy SAT appeared in [78] and was extended in [81]. Next to the work that is presented here, I also worked on fuzzy argumentation frameworks [84] and literal preferences in fuzzy answer set programming [82]. Furthermore, I co-authored papers on communication in answer set programming [9,10], possibilistic answer set programming [12], the application of answer set programming to biological networks [56–58], fuzzy equilibrium logic [151,153] and satisfiability in Łukasiewicz logic [152,154].
In this chapter we introduce some preliminary notions from order theory, answer set programming and fuzzy logic. Note that all notations that are frequently used in this thesis are included in a list of symbols at the back of the book.

2.1 Order Theory

Definition 2.1 (from [14]). A binary relation $\leq$ on a set $P$ is a preorder iff for each $x, y, z$ in $P$ the relation obeys the following conditions:

1. Reflexivity: $x \leq x$
2. Transitivity: $(x \leq y) \land (y \leq z) \Rightarrow (x \leq z)$

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

$$(x \leq y) \land (y \leq x) \Rightarrow (x = y)$$

then $\leq$ is called an order relation.
A set $P$ together with a preorder $\leq$ on $P$ is called a **preordered set** and is denoted as $(P, \leq)$. For a given preorder $P = (P, \leq_P)$ we denote $\leq_P$ by $\leq$ if no confusion is possible. Furthermore for a preordered set $(P, \leq)$ the notation $y \geq x$ is equivalent to $x \leq y$. Last, a preordered set $(P, \leq)$ is called finite if $P$ consists of a finite number of elements.

A set $P$ together with an order relation $\leq$ is called a **partially ordered set** (short: **poset**). For an order relation $\leq$ we abbreviate $(x \leq y) \land (x \neq y)$ as $x < y$ (or sometimes $y > x$).

**Definition 2.2** (from [14]). Let $(P, \leq)$ be a poset, let $A$ be a subset of $P$ and let $x$ be an element of $P$. Then we define:

- $x$ is a **lower bound** for $A$ iff $\forall y \in A \cdot x \leq y$
- $x$ is an **upper bound** for $A$ iff $\forall y \in A \cdot y \leq x$

**Definition 2.3** (from [14]). Let $(P, \leq)$ be a poset, let $A$ be a subset of $P$ and let $x$ be an element of $P$. Then we define:

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

The least element and greatest element of $A$ are commonly referred to as the **minimum**, resp. **maximum** of $A$. Note that if they exist, they must be unique [14]. A poset $P$ is called **bounded** if it contains a minimum and a maximum. We denote these elements with $0_P$, resp. $1_P$. If the poset used is clear from the context we denote them with 0, respectively 1. For a given set $A$ we denote the minimum and maximum as $\min A$, resp. $\max A$, if they exist.

**Definition 2.4** (from [14]). Let $(P, \leq)$ be a poset, let $A$ be a subset of $P$ and let $x$ be an element of $P$. Then we define:

- $x$ is the **infimum** of $A$ iff $x$ is the greatest lower bound for $A$
- $x$ is the **supremum** of $A$ iff $x$ is the least upper bound for $A$

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

**Definition 2.5** (from [14]). A poset \((P, \leq)\) is called a **lattice** iff each pair \(\{x, y\} \subseteq P\) has an infimum and supremum. If every subset of \(P\) has an infimum and supremum we call \(P\) a **complete lattice**.

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

**Definition 2.6** (from [169]). Let \((P_1, \leq_1)\) and \((P_2, \leq_2)\) be two posets and let \(f : P_1 \to P_2\) mapping. Then we define

- \(f\) is increasing iff \(\forall x, y \in P_1 \cdot x \leq_1 y \Rightarrow f(x) \leq_2 f(y)\)
- \(f\) is decreasing iff \(\forall x, y \in P_2 \cdot x \leq_2 y \Rightarrow f(x) \geq_2 f(y)\)

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

**Proposition 2.7** (from [169]). Let \(\mathcal{L}\) be a complete lattice and let \(f : \mathcal{L} \to \mathcal{L}\) be an increasing function. Then \(f\) has a least fixpoint, i.e. there is an \(x \in \mathcal{L}\) such that \(f(x) = x\) and for all \(y \in \mathcal{L}\) such that \(f(y) = y\) we have that \(x \leq y\). We denote the least fixpoint of \(f\) as \(f^*\).

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

**Definition 2.8** (from [5]). Let \(\mathcal{L}\) be a complete lattice and let \(f : \mathcal{L} \to \mathcal{L}\) be an increasing function. Then

- \(f^0 = 0_{\mathcal{L}}\)
- \(f^n = f(f^{n-1})\) if \(n\) is a successor ordinal
- \(f^\alpha = \sup\{f^\beta \mid \beta < \alpha\}\) if \(\alpha\) is a limit ordinal
CHAPTER 2. PRELIMINARIES

Proposition 2.9 (from [5]). Let $L$ be a complete lattice and let $f : L \rightarrow L$ be an increasing function. Then $f^\alpha = f^\alpha$, where $\alpha$ is a limit ordinal.

2.2 Answer Set Programming

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

2.2.1 Definitions

In this section we define the syntax and semantics of ASP. The terminology is based on material from [173].

Language

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

A literal is either an atom $a$ or a negated atom $\neg a$ (called a classically or strongly negated literal). An extended literal is either a literal or an expression of the form not $l$ (called a negation-as-failure literal or naf-literal), where $l$ is a literal. An (extended) literal is called grounded if its underlying atom is grounded.

For a set of literals $L$ we use not $L$ to denote the set \{not $l \mid l \in L$\} and $\neg L$ to denote the set \{$\neg l \mid l \in L$\}, where $\neg(\neg l) = l$. With $L^+$ we denote the positive part of $L$, i.e. $L^+ = \{a \in L \mid a$ is an atom$\}$. Furthermore, for a set of extended literals $L$ we denote with $L^-$ the set of literals underlying the naf-literals in $L$, i.e. $L^- = \{l \mid$ not $l \in L$\}. For a set of grounded literals $L$, we say that $L$ is consistent iff $L \cap \neg L = \emptyset$. Lastly, for a set of grounded atoms $A$, we denote the set of all literals over $A$ as $\text{Lit}_A$, i.e. $\text{Lit}_A = A \cup \neg A$. 
2.2. ANSWER SET PROGRAMMING

Definition 2.10. A normal rule \( r \) is an expression of the form

\[
a ← β
\]

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

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

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

Definition 2.11. An answer set program (ASP program) is a countable set of rules.

Example 2.12. Consider the following program \( P_{ex2.12} \).

\[
\text{rightOf}(\text{john}, \text{chris}) ← \quad (2.1) \\
\text{rightOf}(\text{chris}, \text{cathy}) ← \quad (2.2) \\
\text{rightOf}(X, Y) ← \text{rightOf}(X, Z), \text{rightOf}(Z, Y) \quad (2.3)
\]

In this program rules (2.1)–(2.2) are facts stating who is sitting immediately on the right of whom. For example rule (2.1) states that 
john is sitting immediately on the right of 
chris. Rule (2.3) then describes that the relation “right of” is transitive. Note that all rules in this program are positive (and thus also simple).
CHAPTER 2. PRELIMINARIES

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

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

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

**Grounding**

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

**Definition 2.13** (from [173]). Let $P$ be a program. The set of all constants appearing in $P$ is called the **Herbrand universe**, denoted $\mathcal{U}_P$. The **Herbrand base** $\mathcal{B}_P$ of $P$ is the set containing all grounded atoms that can be constructed from the predicates in $P$ and the terms in $\mathcal{U}_P$.

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

$$\text{gnd}(P) = \bigcup_{r \in P} \text{gnd}_{\mathcal{U}_P}(r)$$

**Example 2.14.** Consider the following program $P_{ex2.14}$:

$$p(X,Y) \leftarrow q(X), r(Y)$$
$$q(a) \leftarrow$$
$$r(b) \leftarrow$$
Its grounding is the following program \( \text{gnd}(P_{\text{ex2.14}}) \):

\[
\begin{align*}
p(a,a) & \leftarrow q(a), r(a) \\
p(a,b) & \leftarrow q(a), r(b) \\
p(b,a) & \leftarrow q(b), r(a) \\
p(b,b) & \leftarrow q(b), r(b) \\
q(a) & \leftarrow \\
r(b) & \leftarrow
\end{align*}
\]

Note that the grounding process can be exponential in the size of the program. Therefore researchers have recently started to devote their attention to the study of more efficient grounding methods (see e.g. [65, 97, 167, 168]). In the remainder of this thesis we assume that all programs are grounded, unless stated otherwise.

**Semantics**

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

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

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

**Definition 2.15.** Let \( P \) be a program. An interpretation \( I \) of \( P \) is called a model of \( P \) iff \( \forall r \in P : I \models r \). Furthermore, \( I \) is a minimal model of \( P \) iff \( I \) is a model of \( P \) and no model \( J \) of \( P \) exists such that \( J \subset I \).

For constraint-free positive programs the minimal model is guaranteed to exist and can be computed using the following monotonic operator.
Definition 2.16 (from [171]). Let \( P \) be a constraint-free positive program and let \( I \) be an interpretation of \( P \). The immediate consequence operator \( \Pi_P \) of \( P \) is a \( \mathcal{P}(B_P) \rightarrow \mathcal{P}(B_P) \) function defined as

\[
\Pi_P(I) = \{ a \mid a \leftarrow \beta \in P \land \beta \subseteq I \}
\]

It is easy to see that this operator is monotonic, and due to Proposition 2.7 and Proposition 2.9, that it has a least fixpoint that can be computed using an iterated fixpoint computation. Define \( \Pi_P^0 = \emptyset \) and \( \Pi_P^i = \Pi_P(\Pi_P^{i-1}) \) for \( i \geq 1 \), then \( \Pi_P^j \) is the least fixpoint of \( \Pi_P \), denoted as \( \Pi_P^\ast \), iff \( \Pi_P(\Pi_P^j) = \Pi_P^j \). In other words, the least fixpoint of \( \Pi_P \) can be computed by iteratively applying \( \Pi_P \), starting from the empty interpretation, until a fixpoint is found. Note that this computation always ends if \( B_P \) is finite, which we will assume in the remainder of this thesis. The following proposition shows that this least fixpoint coincides with the minimal model of \( P \), hence the procedure explained above gives us a procedural method for computing the minimal model of a program.

Proposition 2.17 (from [173]). Let \( P \) be a constraint-free positive program. Then \( I \) is a model of \( P \) iff \( \Pi_P(I) \subseteq I \). Furthermore, the unique minimal model of \( P \) equals the least fixpoint of \( \Pi_P \).

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

Proposition 2.18 (from [173]). Let \( P \) be a simple program. An interpretation \( I \) is the unique minimal model of \( P \) iff \( I \) is the unique minimal model of \( P' \).

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

**Definition 2.19.** Let \( P \) be a simple program. An interpretation \( A \) is called an **answer set** of \( P \) iff \( A \) is the minimal model of \( P \).

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

**Example 2.20.** Consider the following program \( P_{\text{ex2.20}} \):

\[
\begin{align*}
\text{person}(\text{john}) & \leftarrow \\
\text{suitable}_\text{for}_\text{job}(X) & \leftarrow \text{person}(X), \text{not}_\text{criminal}_\text{record}(X)
\end{align*}
\]

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

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

**Definition 2.21** (from [66]). Let \( P \) be a normal program and let \( I \) be an interpretation of \( P \). The **reduct** of \( P \) w.r.t. \( I \), denoted as \( P^I \), is the program

\[
\{a \leftarrow (\beta \setminus \text{not}_\beta) \mid a \leftarrow \beta \in P \land (I \models \text{not}_\beta)\}
\]

In other words, \( P^I \) is obtained by removing all naf-literals not \( l \) for which \( l \not\in I \) from the bodies of the rules in \( P \) and removing all rules containing not \( l \) for which \( l \in I \).
CHAPTER 2. PRELIMINARIES

Intuitively this means we remove the naf-literals of rules that could be satisfied by \( I \), judged only by looking at the negative information in \( I \), and discard the rules that can never be satisfied by \( I \).

**Definition 2.22** (from [173]). Let \( P \) be a normal program. An interpretation \( A \) is called an **answer set** of \( P \) iff \( A \) is the minimal model of \( P^A \).

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

**Example 2.23.** Consider again program \( P_{ex2.20} \) from Example 2.20 together with its two minimal models \( M_1 \) and \( M_2 \). Computing the reducts gives us the program \( P_{ex2.20}^{M_1} \):

\[
person(john) \leftarrow
\]

and \( P_{ex2.20}^{M_2} \):

\[
person(john) \leftarrow
suitable\_for\_job(john) \leftarrow person(john)
\]

It is easy to see that the minimal model of \( P_{ex2.20}^{M_1} \) is \( \{person(john)\} \), which is not equal to \( M_1 \), hence \( M_1 \) is not an answer set of \( P_{ex2.20} \). The minimal model \( M_2 \) is an answer set of \( P_{ex2.20} \), however, as \( M_2 \) is the minimal model of \( P_{ex2.20}^{M_2} \).

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

**Proposition 2.24** (from [5]). Let \( P \) be a normal program. Any answer set of \( P \) is a minimal model of \( P \).

The reverse does not hold, as Example 2.23 shows. Note that a program can have multiple answer sets or even no answer sets, as shown in the following examples.

**Example 2.25.** Consider program \( P_{nondet} \):

\[
a \leftarrow \text{not } b
\]

\[
b \leftarrow \text{not } a
\]
This program has two answer sets: \( A_1 = \{a\} \) and \( A_2 = \{b\} \).

**Example 2.26.** Consider program \( P_{\text{empty}} \):

\[
p \leftarrow \neg p
\]

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

The fact that programs can have multiple answer sets or no answer sets forms the basis for the **answer set programming paradigm** [121, 135]. In this paradigm, we solve a certain combinatorial problem by writing an ASP program such that the answer sets of the program correspond to the solutions of the problem. Most often this is done by writing a program in a generate-define-test style, as shown in the following example:

**Example 2.27.** Consider the problem of coloring the vertices of a graph in either black or white, such that adjacent nodes are colored differently. We can model this problem using the ASP program \( P_{\text{gc}} \):

\[
\begin{align*}
\text{black}(X) & \leftarrow \neg \text{white}(X) & \text{(2.4)} \\
\text{white}(X) & \leftarrow \neg \text{black}(X) & \text{(2.5)} \\
\text{sim}(X,Y) & \leftarrow \text{white}(X), \text{white}(Y) & \text{(2.6)} \\
\text{sim}(X,Y) & \leftarrow \text{black}(X), \text{black}(Y) & \text{(2.7)} \\
& \leftarrow \text{edge}(X,Y), \text{sim}(X,Y) & \text{(2.8)}
\end{align*}
\]

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

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

\[
\begin{align*}
&\text{node}(a) \leftarrow \\
&\text{node}(b) \leftarrow \\
&\text{node}(c) \leftarrow \\
&\text{edge}(a,b) \leftarrow \\
&\text{edge}(b,a) \leftarrow \\
&\text{edge}(a,c) \leftarrow \\
&\text{edge}(c,a) \leftarrow
\end{align*}
\]

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

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

and

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

where

\[
I = \{\text{node}(a), \text{node}(b), \text{node}(c), \text{edge}(a,b), \text{edge}(b,a), \text{edge}(a,c), \text{edge}(c,a), \\
\text{edge}(b,c), \text{edge}(c,b), \text{sim}(a,a), \text{sim}(b,b), \text{sim}(c,c)\}
\]

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

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

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

### 2.2.2 Classical negation vs negation-as-failure

As mentioned before, in ASP we have two types of negation: classical negation and negation-as-failure. The former states that the negation of an atom $a$ can be explicitly derived, whereas not $a$ is true if we cannot derive $a$. The important difference between these two constructs is perhaps best illustrated by an example:
Example 2.28. Suppose we are building an ASP program for controlling a car. To ensure safety this car must abide by the traffic rules and so we need to express the usual rule of giving way to the right, i.e. that we have to yield to cars coming from the right. Doing this with negation-as-failure we obtain the rule

\[
driveon \leftarrow \text{not carOnRight}
\]

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

\[
driveon \leftarrow \neg \text{carOnRight}
\]
CHAPTER 2. PRELIMINARIES

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

Classical negation can be eliminated from the program by introducing for each literal \( \neg a \) a new atom \( a' \) and adding the constraint \( \leftarrow a, a' \) to the program. The constraint ensures that any model will be consistent, and thus ensures that the semantics are preserved. For more details see [5]. Unless stated otherwise, for all programs in the remainder of this chapter we assume that classical negation has been eliminated in this way.

2.2.3 Links to SAT

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

Example 2.29. Consider program \( P_{gc} \cup F_{2,1a} \) from Example 2.27. Its corresponding SAT problem, denoted as \( \text{comp}(P_{gc} \cup F_{2,1a}) \), is obtained by replacing \( \leftarrow \) by \( \equiv \), replacing \( \neg l \) by \( \neg l \) for any literal \( l \), replacing empty bodies by \( \text{True} \), replacing empty heads by \( \text{False} \) and grouping rule bodies of rules with the same heads by disjunction:

\[
\begin{align*}
\text{node}(a) & \equiv \text{True} \\
\text{node}(b) & \equiv \text{True} \\
\text{node}(c) & \equiv \text{True} \\
\text{edge}(a, b) & \equiv \text{True} \\
\text{edge}(b, a) & \equiv \text{True} \\
\text{edge}(a, c) & \equiv \text{True} \\
\text{edge}(c, a) & \equiv \text{True} \\
\text{black}(a) & \equiv \neg \text{white}(a) \\
\text{white}(a) & \equiv \neg \text{black}(a) \\
\text{black}(b) & \equiv \neg \text{white}(b)
\end{align*}
\]
2.2. ANSWER SET PROGRAMMING

white(b) ≡ ¬black(b)
black(c) ≡ ¬white(c)
white(c) ≡ ¬black(c)
sim(a, a) ≡ (white(a) ∧ white(a)) ∨ (black(a) ∧ black(a))
sim(a, b) ≡ (white(a) ∧ white(b)) ∨ (black(a) ∧ black(b))
sim(a, c) ≡ (white(a) ∧ white(c)) ∨ (black(a) ∧ black(c))
sim(b, a) ≡ (white(b) ∧ white(a)) ∨ (black(b) ∧ black(a))
sim(b, b) ≡ (white(b) ∧ white(b)) ∨ (black(b) ∧ black(b))
sim(b, c) ≡ (white(b) ∧ white(c)) ∨ (black(b) ∧ black(c))
sim(c, a) ≡ (white(c) ∧ white(a)) ∨ (black(c) ∧ black(a))
sim(c, b) ≡ (white(c) ∧ white(b)) ∨ (black(c) ∧ black(b))
sim(c, c) ≡ (white(c) ∧ white(c)) ∨ (black(c) ∧ black(c))
False ≡ edge(a, a) ∧ sim(a, a)
False ≡ edge(a, b) ∧ sim(a, b)
False ≡ edge(a, c) ∧ sim(a, c)
False ≡ edge(b, a) ∧ sim(b, a)
False ≡ edge(b, b) ∧ sim(b, b)
False ≡ edge(b, c) ∧ sim(b, c)
False ≡ edge(c, a) ∧ sim(c, a)
False ≡ edge(c, b) ∧ sim(c, b)
False ≡ edge(c, c) ∧ sim(c, c)

One can easily verify that answer sets $A_1$ and $A_2$ from Example 2.27 are models of the above translation to SAT.

Formally this translation is called the completion of an ASP program. It is also commonly called Clark’s completion after Keith Clark, who originally proposed this correspondence as a method for describing the semantics of negation-as-failure [27].
CHAPTER 2. PRELIMINARIES

**Definition 2.30** (from [27]). Let $P$ be a normal program. The completion of $P$, denoted $\text{comp}(P)$, is defined as the following set of propositions:

\[
\text{comp}(P) = \{ a \equiv \bigvee \{ \text{comp}(\beta) \mid a \leftarrow \beta \in P \} \mid a \in B_P, \beta \neq \emptyset \} \\
\cup \{ a \equiv \text{True} \mid (a \leftarrow) \in B_P \} \\
\cup \{ \text{False} \equiv \text{comp}(\beta) \mid (\leftarrow \beta) \in B_P \}
\]

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

In [54], Fages showed that under certain conditions the answer sets of an ASP program and the models of its completion coincide. The question then arises why we need ASP at all and why we cannot just write our problems directly as their SAT encodings? After all, the completion of the graph coloring program introduced above is almost as concise as the grounded program. There are ASP programs for which the models of the completion do not coincide with the answer sets, however. This occurs when there are atoms that positively depend on other atoms, as shown in the following example.

**Example 2.31.** Consider the following program $P_{ex\,2.31}$:

\[a \leftarrow a\]

The answer set of $P_{ex\,2.31}$ is $\emptyset$. However, its completion $\text{comp}(P_{ex\,2.31})$ has two models: $\emptyset$ and $\{a\}$.

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

**Example 2.32.** Consider the following program $P_{ex\,2.32}$:

\[a \leftarrow a\]

\[p \leftarrow \text{not } p, \text{not } a\]
One can easily verify that \( P_{ex2.32} \) has no answer sets. However, its completion has the single (and therefore trivially minimal) model \( \{a\} \).

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

**Proposition 2.33** (from [66]). Let \( P \) be a normal answer set program. Then any answer set \( A \) of \( P \) is a minimal model of the completion \( \text{comp}(P) \) of \( P \).

We can now answer the question why we need ASP. While the graph coloring example could be concisely encoded in SAT, this does not hold in general. For example, programs incorporating recursion require a more involved translation [101]. However, in many application domains it is quite convenient to define predicates recursively, such as the transitive closure defined by the \( \text{rightOf} \) predicate in Example 2.12. The following program illustrates the use of recursion on the problem of finding Hamilton cycles in a graph.

**Example 2.34.** Consider the problem of determining Hamilton cycles of a graph, i.e. finding a path in the graph that visits every vertex exactly once. In [121] the authors propose to encode this problem with the following program \( P_{hc} \):

\[
\begin{align*}
\text{in}(U, V) & \leftarrow \text{edge}(U, V), \neg \text{out}(U, V) \quad (2.9) \\
\text{out}(U, V) & \leftarrow \text{edge}(U, V), \neg \text{in}(U, V) \quad (2.10) \\
\text{reachable}(V) & \leftarrow \text{in}(v_0, V) \quad (2.11) \\
\text{reachable}(V) & \leftarrow \text{reachable}(U), \text{in}(U, V) \quad (2.12) \\
& \quad \leftarrow \text{in}(U, V), \text{in}(U, W), V \neq W \quad (2.13) \\
& \quad \leftarrow \text{in}(U, W), \text{in}(V, W), U \neq V \quad (2.14) \\
& \quad \leftarrow \text{vertex}(U), \neg \text{reachable}(U), \text{in}(U, V) \quad (2.15)
\end{align*}
\]

where in rules (2.13) and (2.14) \( V \neq W \) and \( U \neq V \) are extensions of the ungrounded ASP language which denote that the grounded instances of these rules where \( V = W \), resp. \( U = V \) holds should not be included in the grounding of the program. Rules (2.9) and (2.10) are the generate rules, where \( \text{in}(U, V) \) means edge \( (U, V) \) is included in the cycle, and \( \text{out}(U, V) \) means the edge
(U, V) is not included in the cycle. Rules (2.11) and (2.12) are the defining part, encoding when a vertex is reachable. Note that we have to explicitly state the starting vertex \( v_0 \) for this program to work. Furthermore note that we do not state that \( \text{reachable}(v_0) \) is necessarily true. This is to ensure that the program adds an edge to \( v_0 \), and thus creates a cycle rather than a path. Last, rules (2.13)–(2.15) are the constraints eliminating answer sets in which a vertex is visited twice or a certain node is never visited.

Note that the reachable predicate is defined recursively. This can lead to counterintuitive results for the models of the completion. Consider the following input rules I from [4]:

\[
\text{vertex}(v_0) \leftarrow \\
\text{vertex}(v_1) \leftarrow \\
\text{edge}(v_0, v_0) \leftarrow \\
\text{edge}(v_1, v_1) \leftarrow
\]

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

### 2.3 Fuzzy Logic

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

#### 2.3.1 Fuzzy Sets

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

**Definition 2.35.** Consider a complete lattice \( \mathcal{L} \). An \( \mathcal{L} \)-fuzzy set in a universe \( X \) is a mapping from \( X \) to \( \mathcal{L} \).
2.3. FUZZY LOGIC

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

**Definition 2.36.** Given two \(\mathcal{L}\)-fuzzy sets \(A\) and \(B\) in a universe \(X\) we define the Zadeh inclusion \(A \subseteq B\) of \(A\) and \(B\) as follows:

\[
A \subseteq B \equiv \forall x \in X \cdot A(x) \leq B(x)
\]

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

**Definition 2.37.** Consider an \(\mathcal{L}\)-fuzzy set \(A\) in a universe \(X\). The support of \(A\) is the set \(\text{supp}(A)\) that is defined by \(\text{supp}(A) = \{ x \mid x \in X, A(x) > 0 \}\).

2.3.2 Logical Operators on Bounded Lattices

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

**Negators, Triangular Norms and Triangular Conorms**

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

**Definition 2.38.** A negator \(\mathcal{N}\) on a bounded lattice \(\mathcal{L}\) is a decreasing \(\mathcal{L} \rightarrow \mathcal{L}\) mapping that satisfies \(\mathcal{N}(0) = 1\) and \(\mathcal{N}(1) = 0\). The negator \(\mathcal{N}\) is called involutive iff for each \(x \in \mathcal{L}\) we have \(\mathcal{N}({\mathcal{N}(x)}) = x\).

Note that if we take \(True = 1\) and \(False = 0\), the boundary conditions \(\mathcal{N}(0) = 1\) and \(\mathcal{N}(1) = 0\) ensure that any negator behaves as the negation \(\neg\) of classical logic on the lattice \(\{0, 1\}, \leq\). The generalizations of other logical operators will similarly need certain boundary conditions to ensure that the classical behavior is recovered for the lattice \(\{0, 1\}, \leq\).
Example 2.39. We introduce two common negators over \([0, 1], \leq\), the lattice most commonly associated with fuzzy logic.

1. The Gödel negator \(N_M\) (also known as drastic negator) on a bounded lattice \(L\) is the \(L \rightarrow L\) mapping defined as

\[
N_M(x) = \begin{cases} 
1 & \text{if } x = 0 \\
0 & \text{otherwise}
\end{cases}
\]

2. The standard negator \(N_W\) (also known as Łukasiewicz negator) is the \([0, 1] \rightarrow [0, 1]\) mapping defined as \(N_W(x) = 1 - x\). Note that this negator is involutive.

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

Definition 2.40 (from [90]). A triangular norm \(T\) (short: t-norm) on a bounded lattice \(L\) is an increasing, associative and commutative \(L^2 \rightarrow L\) mapping that satisfies the boundary condition \(T(1, x) = x\) for any \(x \in L\).

Definition 2.41 (from [90]). A triangular conorm \(S\) (short: t-conorm) on a bounded lattice \(L\) is an increasing, associative and commutative \(L^2 \rightarrow L\) mapping that satisfies the boundary condition \(S(0, x) = x\) for any \(x \in L\).

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

Example 2.42 (from [37]). The following two t-norms and t-conorms are well-known.

1. Consider a bounded lattice \(L = (L, \leq)\). One can immediately see that \(\sqcap\) is a triangular norm on \(L\), which we will denote as \(T_M\). Likewise \(\sqcup\) is a triangular t-conorm on \(L\) which we will denote as \(S_M\).
2. Consider a bounded lattice $\mathcal{L} = (L, \leq)$. The drastic t-norm $\mathcal{T}_Z$ is a $\mathcal{L}^2 \to \mathcal{L}$ mapping defined by

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

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

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

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

**Proposition 2.43** (from [37]). For any t-norm $\mathcal{T}$ and t-conorm $\mathcal{S}$ on a bounded lattice $\mathcal{L} = (L, \leq)$ we have that for any $x, y \in \mathcal{L}$.

$$
\mathcal{T}_Z(x, y) \leq \mathcal{T}(x, y) \leq \mathcal{T}_M(x, y) \\
\mathcal{S}_M(x, y) \leq \mathcal{S}(x, y) \leq \mathcal{S}_Z(x, y)
$$

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

**Definition 2.44.** Let $\mathcal{N}$ be a negator on a bounded lattice $\mathcal{L}$. The dual image of a $\mathcal{L}^2 \to \mathcal{L}$ mapping $f$ w.r.t. $\mathcal{N}$ is the $\mathcal{L}^2 \to \mathcal{L}$ mapping $f^{\leftrightarrow \mathcal{N}}$ defined as

$$
f^{\leftrightarrow \mathcal{N}}(x, y) = \mathcal{N}(f(\mathcal{N}(x), \mathcal{N}(y)))
$$

It turns out that if $\mathcal{N}$ is involutive the dual of a t-norm (resp. t-conorm) is a t-conorm (resp. t-norm) and vice versa [37].
CHAPTER 2. PRELIMINARIES

<table>
<thead>
<tr>
<th>t-norm</th>
<th>t-conorm</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_M(x, y) = \min(x, y)$</td>
<td>$S_M(x, y) = \max(x, y)$</td>
</tr>
<tr>
<td>$T_W(x, y) = \max(0, x + y - 1)$</td>
<td>$S_W(x, y) = \min(1, x + y)$</td>
</tr>
<tr>
<td>$T_P(x, y) = x \cdot y$</td>
<td>$S_P(x, y) = x + y - x \cdot y$</td>
</tr>
</tbody>
</table>

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

Example 2.45. In Table 2.1 we have defined the well known t-norms $T_M$, $T_W$ and $T_P$ on the complete lattice $([0, 1], \leq)$. They are respectively called the minimum t-norm, Łukasiewicz t-norm and product t-norm. The t-conorms $S_M$, $S_W$ and $S_P$ defined in this same table are respectively called the maximum t-conorm, Łukasiewicz t-conorm (also known as the bounded sum) and product t-conorm (also known as the probabilistic sum).

Each t-norm is dual with the t-conorm on the same line, w.r.t. the standard negator $N_W$. Specifically we have that $T_M^{\leftrightarrow N_W} = S_M$, $T_W^{\leftrightarrow N_W} = S_W$ and $T_P^{\leftrightarrow N_W} = S_P$ and $S_M^{\leftrightarrow N_W} = T_M$, $S_W^{\leftrightarrow N_W} = T_W$ and $S_P^{\leftrightarrow N_W} = T_P$.

Note that different t-norms have different properties. For example, consider the t-norms introduced in Example 2.45 on the current page. The minimum t-norm $T_M$ is the only t-norm that satisfies idempotency (i.e. $T_M(x, x) = x$ for every $x$ [90]), whereas the Łukasiewicz t-norm $T_W$ is the only t-norm in Table 2.1 that satisfies the law of contradiction\(^1\) w.r.t. the standard negator $N_W$. Which t-norm to use is thus greatly dependent on the application and especially on the properties of classical conjunction that are important in the problem at hand. A thorough discussion of the logical properties that remain valid for the three t-norms $T_M$, $T_W$ and $T_P$ can be found in [86].

Implicators

Definition 2.46. An implicator $I$ on a bounded lattice $\mathcal{L}$ is a $\mathcal{L}^2 \rightarrow \mathcal{L}$ mapping that is increasing in its first partial mapping and decreasing in its second partial mapping and furthermore satisfies the boundary conditions $I(0, 0) = 0$ and $I(1, x) = x$ for each $x \in \mathcal{L}$.

\(^1\)In classical logic the law of contradiction states that $p \land \neg p = \text{False}$.  

44
2.3. FUZZY LOGIC

<table>
<thead>
<tr>
<th>S-implicator</th>
<th>Residual implicator</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{TM,NW}(x,y) = \max(1 - x, y)$</td>
<td>$I_M(x,y) = \begin{cases} 1 &amp; \text{if } x \leq y \ y &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$I_{TW,NW}(x,y) = \min(1 - x + y, 1)$</td>
<td>$I_W(x,y) = \min(1 - x + y, 1)$</td>
</tr>
<tr>
<td>$I_{TP,NW}(x,y) = 1 - x + x \cdot y$</td>
<td>$I_P(x,y) = \begin{cases} 1 &amp; \text{if } x \leq y \ y \cdot x &amp; \text{otherwise} \end{cases}$</td>
</tr>
</tbody>
</table>

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

It turns out that any implicator $I$ on a bounded lattice $L$ also satisfies the boundary conditions $I(0,x) = 0$ and $I(x,1) = 1$, for any $x \in L$ [36].

Now the question arises how implicators can be constructed. Since we already constructed t-conorms and negators, a natural idea is to start from a generalization of the classical logic tautology $p \Rightarrow q \equiv \neg p \lor q$.

**Definition 2.47.** Let $L$ be a bounded lattice, let $S$ be a t-conorm on $L$ and let $N$ be a negator on $L$. The $L^2 \rightarrow L$ mapping $I_{S,N}$ defined as $I_{S,N}(x,y) = S(N(x),y)$ is called the S-implicator induced by $S$ and $N$.

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

**Definition 2.48.** Let $L$ be a bounded lattice, let $T$ be a t-norm on $L$ and let $N$ be an involutive negator on $L$. The $L^2 \rightarrow L$ mapping $I_{T,N}$ defined as $I_{T,N}(x,y) = N(T(x,N(y)))$ is called the S-implicator induced by $T$ and $N$.

**Example 2.49.** In the first column of Table 2.2 one can find the S-implicators on $[0,1]$ that are induced by the t-norms from Table 2.1 and the involutive negator $N_W$. The implicator $I_{TM,NW}$ is called the Kleene-Dienes implicator, $I_{TW,NW}$ the Łukasiewicz implicator and $I_{TP,NW}$ the Reichenbach implicator.
CHAPTER 2. PRELIMINARIES

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

$$T_M(0.5, I_{T_{M,NW}}(0.5, 0.3)) = 0.5 > 0.3$$

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

**Definition 2.50.** Let $L$ be a complete lattice and let $T$ be a t-norm on $L$. The **residual implicator** (short: R-implicator) of $T$ is the $L^2 \to L$ mapping

$$I_T(x, y) = \sup \{ \lambda \mid \lambda \in L \land T(x, \lambda) \leq y \}$$

The following proposition shows that R-implicators are indeed implicators as defined by Definition 2.46.

**Proposition 2.51** (from [36]). Let $L$ be a complete lattice and let $T$ be a t-norm on $L$. Then the residual implicator of $T$ is an implicator on $L$.

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

**Proposition 2.52** (from [36]). Let $L$ be a complete lattice and let $T$ be a t-norm on $L$. If for each $\lambda \in L$ and family $(x_i)_{i \in I}$ we have that $T(\sup_{i \in I} x_i, \lambda) = \sup_{i \in I} T(x_i, \lambda)$, i.e. all partial mappings of $T$ are supmorphisms, it holds that

$$T(x, y) \leq z \iff x \leq I(y, z)$$

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

The residuation principle is also commonly referred to as the **Galois connection** or **adjoint property**. In [35] it is shown that for a t-norm $T$ that satisfies the condition in Proposition 2.52, we have that $T(x, I_T(y, z)) \leq y$ for each $x, y, z \in L$, i.e. the generalization of the modus ponens, introduced above, holds. For this reason we will limit our attention to t-norms satisfying this condition in the remainder of this thesis.
Example 2.53. In the second column of Table 2.2 we listed some common residual implicators on \([0, 1], \leq\). The implicator \(\mathcal{I}_M\) is called the Gödel implicator, \(\mathcal{I}_W\) the Łukasiewicz implicator, \(\mathcal{I}_P\) the Goguen implicator. Note that the Łukasiewicz implicator is both a residual implicator and an S-implicator.

An important property of residual implicators is the following.

**Proposition 2.54** (from [35]). Let \(\mathcal{I}\) be a residual implicator on \(\mathcal{L}\). For any \(x, y \in \mathcal{L}\) it holds that \(\mathcal{I}(x, y) = 1\) iff \(x \leq y\).

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

**Proposition 2.55** (from [90]). Let \(T\) be a continuous t-norm on \(([0, 1], \leq)\). Then for any \(x, y \in [0, 1]\) it holds that \(T(x, \mathcal{I}_T(x, y)) = \min(x, y)\). This property is called *divisibility*.

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

**Definition 2.56.** Consider a residual implicator \(\mathcal{I}\) and t-norm \(T\) on \(\mathcal{L}\). The biimplicum of \(\mathcal{I}\) and \(T\) is the \(L^2 \to L\) mapping \(\approx\) defined for all \(x, y \in \mathcal{L}\) as:

\[
x \approx y = T(\mathcal{I}(x, y), \mathcal{I}(y, x))
\]

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

**Definition 2.57.** Let \(\mathcal{L}\) be a bounded lattice and let \(\mathcal{I}\) be an implicator on \(\mathcal{L}\). The induced negator of \(\mathcal{I}\) is then the \(L \to L\) mapping \(N_{\mathcal{I}}\) defined as \(N_{\mathcal{I}}(x) = \mathcal{I}(x, 0)\), for each \(x \in \mathcal{L}\).
Example 2.58. The Gödel negator \( N_M \) is the induced negator of the Gödel implicator; the Łukasiewicz negator \( N_W \) is the induced negator of the Łukasiewicz implicator.

2.3.3 Formal Fuzzy Logics

In [72] it is shown that for the minimum t-norm, Łukasiewicz t-norm and product t-norm a propositional calculus capable of describing their properties can be constructed. The author of [72] furthermore identifies a propositional calculus that captures the properties shared by all continuous t-norms on \((0, 1]\). We briefly discuss this in this section.

Definition 2.59 (from [72]). The propositional calculus \( \text{PC}(T) \) given by the continuous t-norm \( T \) on \((0, 1]\) has propositional variables \( p_1, \ldots, p_n \), the connectives \( \land \) and \( \rightarrow \) and the truth constant \( \overline{0} \) for 0. Formulas are defined as usual: each propositional variable is a formula; \( \overline{0} \) is a formula; if \( \varphi, \psi \), are formulas, then \( \varphi \land \psi \) and \( \varphi \rightarrow \psi \) are formulas. Further connectives are defined as follows:

1. \( \varphi \Delta \psi = \varphi \land (\varphi \rightarrow \psi) \)
2. \( \varphi \triangledown \psi = (((\varphi \rightarrow \psi) \rightarrow \psi) \land (((\psi \rightarrow \varphi) \rightarrow \varphi) \lor (\varphi \rightarrow \varphi)) \)
3. \( \neg \varphi = \varphi \rightarrow \overline{0} \)
4. \( \varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi) \)

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

1. \( e(\overline{0}) = 0 \)
2. \( e(\varphi \rightarrow \psi) = I_T(e(\varphi), e(\psi)) \)
3. \( e(\varphi \land \psi) = T(e(\varphi), e(\psi)) \)

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

One can see that 1-tautologies are formulas that are absolutely true under any evaluation. Furthermore note that from Proposition 2.55 we know that the definition of \( \triangle \) corresponds to the minimum, which is one of the reasons why the calculus is restricted to continuous t-norms. Now a logic can be constructed that captures the properties that are common to all continuous t-norms on \([0, 1], \leq\).

**Definition 2.60** (from [72]). The following formulas are axioms of the basic logic BL:

(A1) \((\varphi \to \psi) \to ((\psi \to \chi) \to (\varphi \to \chi))\)
(A2) \((\varphi \land \psi) \to \varphi\)
(A3) \((\varphi \land \psi) \to (\psi \land \varphi)\)
(A4) \((\varphi \land (\varphi \to \psi)) \to (\psi \land (\varphi \to \varphi))\)
(A5) \((\varphi \to (\psi \to \chi)) \to ((\varphi \land \psi) \to \chi)\)
(A6) \(((\varphi \land \psi) \to \chi) \to (\varphi \to (\psi \to \chi))\)
(A7) \(((\varphi \to \psi) \to \chi) \to (((\psi \to \varphi) \to \chi) \to \chi)\)
(A8) \(\mathbf{0} \to \varphi\)

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

The above axioms all have intuitive meanings and correspond to properties in classical logic. Axiom (A1) is transitivity of implication. Axiom (A2) is sometimes called weakening of conjunction in classical logic. Axioms (A3) and (A4) express the commutativity of \( \land \), respectively \( \triangle \). Axioms (A5) and (A6) are commonly called the shunting rules in classical logic. Axiom (A7) is a variant of proof by cases: if \( \chi \) follows from \( \varphi \to \psi \) then if \( \chi \) also follows from \( \psi \to \varphi \) we have \( \chi \) is true. Finally Axiom (A8) states that anything can be proven from a false proposition.

The axioms of BL are all 1-tautologies in each PC\((T)\), where \( T \) is a continuous t-norm on \([0, 1], \leq\). Hence these properties are true for all continuous t-norms on \([0, 1], \leq\). It can be shown that BL is sound, i.e. that each provable formula in BL is a 1-tautology in each PC\((T)\). Two types of completeness are distinguished: standard completeness and general completeness. The former states that every 1-tautology in each PC\((T)\) can be proven in BL and has been shown to hold in [26]. The latter defines completeness with respect to a more general semantics for BL, called BL-algebras. The general completeness theorem states that a formula
Table 2.3: Propositional logics for the common continuous t-norms on \((0,1],[\leq]\) (from [72])

<table>
<thead>
<tr>
<th>t-norm</th>
<th>logic name</th>
<th>extra axiom(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_M)</td>
<td>Gödel logic</td>
<td>(\varphi \rightarrow (\varphi \land \varphi))</td>
</tr>
<tr>
<td>(T_W)</td>
<td>Łukasiewicz logic</td>
<td>(\lnot\lnot \varphi \rightarrow \varphi)</td>
</tr>
<tr>
<td>(T_P)</td>
<td>product logic</td>
<td>(\lnot \lnot \chi \rightarrow ((\varphi \land \chi \rightarrow \psi \land \chi) \rightarrow (\varphi \rightarrow \psi))) (\varphi \triangle \lnot \varphi \rightarrow 0)</td>
</tr>
</tbody>
</table>

\(\varphi\) is provable in \(\textbf{BL}\) if it is a general BL-tautology, i.e. a tautology for each BL-algebra [72]. We can thus conclude that the logic \(\textbf{BL}\) captures all properties that are common to the continuous t-norms on \((0,1],[\leq]\).

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

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

Finally, reasoning in the above logics can be done using existing methods. For Gödel logic we can use boolean SAT solvers, for Łukasiewicz and rational Pavelka logic we can use mixed integer programming (MIP) [71] or constraint satisfaction [154] and for product logic we can use bounded mixed integer quadratically constrained programming (bMICQP), as is used for fuzzy description logics [15]. Similar to the boolean case, checking whether a set of formulas \(\Theta\) is satisfiable, i.e. whether some model exists for \(\Theta\), is NP-complete [72].

---

50
3.1 Introduction

In the previous chapter we introduced ASP, a language that allows to model combinatorial problems in a declarative manner. Unfortunately, ASP is limited to expressing problems in boolean logic. Many interesting applications require different logics, however. For example, suppose we want to write an ASP program that finds the disease from which a patient is suffering, given a set of his symptoms as the input. Obviously the output of this program will be uncertain, as certain symptoms may occur in 80% of the patients, while others may only occur in about 30%. Hence, the answer sets and the computation of the answer sets should reflect this in some way. This can be done by extending the ASP semantics with a theory of uncertainty, such as possibility or probability theory.

Another interesting application domain for which ASP has no direct support is modeling continuous optimization problems. For example, suppose we wish to optimally place ATMs somewhere along the roads connecting different cities, such that each city has an ATM nearby. Modeling this problem using ASP requires the semantics to cope with defining the “nearness” degree of an ATM and a town. This
can be done by extending the ASP semantics with fuzzy logic.

In recent years, researchers have extended ASP and, more general, logic programming to handle the aforementioned problem domains. Most notable are the probabilistic [30, 61, 110, 111, 129, 130, 163] and possibilistic [1, 11, 132, 133] extensions to handle uncertainty; the fuzzy extensions [22, 77, 112–114, 117, 118, 147, 163, 175–177] which allow to encode the intensity to which the predicates are satisfied, and, more generally, many-valued extensions [28, 29, 31–33, 52, 59, 87, 88, 91, 92, 94–96, 103, 104, 128, 155, 159, 160, 164, 165].

In this thesis we focus on fuzzy answer set programming (FASP). This ASP extension bases its semantics on fuzzy logic (in the narrow sense). It is capable of encoding continuous optimization problems in a concise manner, similar to how ASP is able to encode discrete optimization problems. Currently, there exist a multitude of different fuzzy answer set programming languages which extend the basic idea with various enhancements. In Chapter 4 we provide a detailed comparison of these frameworks. In this chapter we distill from previous proposals the FASP language that will be used for presenting our contributions in the succeeding chapters. We begin by introducing the necessary definitions in Section 3.2. The syntax is introduced in Section 3.2.1, followed by the semantics in Section 3.2.2. Afterwards, in Section 3.3, we illustrate the main ideas of the language with a fuzzy variant of the graph coloring problem. Interestingly the example has the same structure as the ASP program for graph coloring introduced in Example 2.27 on page 33. This shows that FASP preserves the declarative advantage of ASP, while adding the power to model continuous problems.

3.2 Definitions

In this section we define the syntax and semantics of fuzzy answer set programming.

3.2.1 Language

Fuzzy answer set programming (FASP) is built from a language containing terms, atoms, truth values and function symbols as basic building blocks. A term is either a constant or a variable. Similar as for ASP we adopt the convention that constants (variables) are denoted by a symbol starting with a lower-case (upper-case) character. An atom is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate symbol of arity $n$ and $t_i$, for $1 \leq i \leq n$, are terms. A term or an atom is
3.2. DEFINITIONS

called **grounded** if it does not contain any variables.

**Definition 3.1.** A **FASP rule** on a complete lattice \( L \) is an expression of the form

\[
    r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m)
\]

(3.1)

where \( r \) is the **label** of the rule, \( a \) is either an atom or a value from \( L \), \( b_i \), for \( 1 \leq i \leq n \), and \( c_j \), for \( 1 \leq j \leq m \), are either atoms or values from \( L \), and \( f \) is a (total) \( L^{n+m} \rightarrow L \) mapping that is increasing in its \( n \) first and decreasing in its \( m \) last arguments. Furthermore we require that \( f \) is computable in finite time. When there is no cause for confusion, we will use the label of the rule to denote the rule itself. For a rule as (3.1), the left-hand side \( a \) is called the **head** of the rule, denoted \( r_h \), the right-hand side \( f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) is called the **body** of the rule, denoted \( r_b \).

In the remainder of this thesis we implicitly assume all lattices to be complete. For convenience we will often shorten the notation of a general rule such as (3.1) using \( r : a \leftarrow a \). The **Herbrand base** of a rule \( r \), denoted \( B_r \), is defined as the set of atoms occurring in \( r \). Similar to ASP, FASP rules can be divided in certain classes, depending on the conditions satisfied by their head and body.

1. A rule \( r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) on a lattice \( L \) is called a **constraint** if \( a \in L \).

2. A rule \( r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) on a lattice \( L \) is called a **fact** if all \( b_i \), for \( 1 \leq i \leq n \), and \( c_j \), for \( 1 \leq j \leq m \), are elements from \( L \).

3. A rule \( r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) on a lattice \( L \) is called **positive** if \( m = 0 \) or \( c_j \in L \) for \( 1 \leq j \leq m \).

4. A rule \( r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) on a lattice \( L \) is called **simple** if it is positive and not a constraint.

For convenience we will write any FASP rule \( r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) with \( f \) the identity function, \( n = 1 \) and \( m = 0 \) as \( r : a \leftarrow b_1 \).

**Definition 3.2.** A **fuzzy answer set program** (**FASP program**) on a complete lattice \( L \) is a finite set of FASP rules on \( L \).
CHAPTER 3. FUZZY ANSWER SET PROGRAMMING

Given a FASP program \( P \), the Herbrand base \( B_P \) is defined as \( B_P = \bigcup \{ B_r \mid r \in P \} \). The grounding \( gnd(P) \) of a FASP program \( P \) is defined as in Section 2.2.1. Unless stated otherwise, we assume all programs to be grounded. Furthermore, for an atom \( a \in B_P \) we denote with \( P_a \) the set of rules in \( P \) with atom \( a \) in the head. The lattice on which \( P \) is defined is denoted as \( \mathcal{L}_P \). We also adopt the convention that the lattice used in examples is \( ([0,1], \leq) \), unless stated otherwise.

Similar to ASP, FASP programs can be divided in certain classes, depending on the rules they contain.

1. A FASP program is called constraint-free if it does not contain constraints.
2. A FASP program is called positive if all rules occurring in it are positive rules.
3. A FASP program is called simple if all rules occurring in it are simple rules.

Note that all simple programs are positive.

Example 3.3. Consider the following FASP program \( P_{ex3.3} \):

\[
\begin{align*}
\text{f:} & \quad \text{white}(b) \leftarrow 0.4 \\
\text{r:} & \quad \text{black}(a) \leftarrow \mathcal{N}_W(\text{white}(b)) \\
\text{c:} & \quad 0 \leftarrow \mathcal{T}_W(\text{white}(b), \text{white}(a))
\end{align*}
\]

Rule \( f \) is a fact, \( r \) is a regular rule and \( c \) is a constraint.

3.2.2 Semantics

An interpretation of a FASP program \( P \) is a \( B_P \) to \( \mathcal{L}_P \) mapping. For ease of presentation we write \( I = \{ a_1^{l_1}, \ldots, a_n^{l_n} \} \) for the interpretation \( I \) defined by \( I(a_i) = l_i \) if \( 1 \leq i \leq n \) and \( I(a) = 0 \) otherwise. We extend interpretations to lattice values and expressions of the form \( f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) as follows. Suppose \( I \) is an interpretation of a FASP program \( P \) then

1. \( I(l) = l \) if \( l \in \mathcal{L} \)
2. \( I(f(b_1, \ldots, b_n; c_1, \ldots, c_m)) = f(I(b_1), \ldots, I(b_n); I(c_1), \ldots, I(c_m)) \)

For a rule \( (r:a \leftarrow a) \in P \) we say that it is satisfied by an interpretation \( I \) of \( P \) iff \( I(a) \geq I(\alpha) \). Intuitively we can regard rules as residual implicators. Due
3.2. DEFINITIONS

to Proposition 2.54 \(r\) is then satisfied by \(I\) iff 
\[ I(I(\alpha), I(a)) \geq 1, \]
where \(I\) is the residual implicator corresponding to \(r\).

**Definition 3.4.** Let \(P\) be a FASP program. An interpretation \(I\) of \(P\) is a model of \(P\) iff every rule \(r \in P\) is satisfied by \(I\).

To define minimal models we need an ordering on interpretations. Given two interpretations \(I\) and \(J\) of a FASP program \(P\) we define \(I \subseteq J\) iff \(\forall a \in B_P \cdot I(a) \leq J(a)\) and \(I \subset J\) iff \(\forall a \in B_P \cdot I(a) \leq J(a)\) \(\land (I \neq J)\). Now an interpretation \(I\) of a FASP program \(P\) is called a minimal model of \(P\) iff \(I\) is a model of \(P\) and no model \(J\) of \(P\) exists such that \(J \subseteq I\).

Similar to ASP, minimal models of simple FASP programs can be characterized as the least fixpoints of a monotonic operator that captures forward chaining.

**Definition 3.5** (from [31]). Let \(P\) be a FASP program. The immediate consequence operator \(\Pi_P\) is the \((B_P \rightarrow L_P) \rightarrow (B_P \rightarrow L_P)\) mapping defined for an interpretation \(I\) of \(P\) and \(a \in B_P\) as
\[
\Pi_P(I)(a) = \sup\{I(r_b) \mid r \in P_a\}
\]

**Example 3.6.** Consider the following FASP program \(P_{\text{ex3.6}}\):

\[
\begin{align*}
  r_1 & : a \leftarrow 0.8 \\
  r_2 & : c \leftarrow 0.5 \\
  r_3 & : b \leftarrow T_M(a, c) \\
  r_4 & : b \leftarrow 0.2
\end{align*}
\]

Now consider the interpretation \(\emptyset\) that attaches to each atom \(a \in B_{P_{\text{ex3.6}}}\) the value 0. If we apply \(\Pi_P\) to this interpretation we obtain:

1. For \(a\): \(\Pi_P(\emptyset)(a) = 0.8\)
2. For \(b\): \(\Pi_P(\emptyset)(b) = \max(T_M(\emptyset(a), \emptyset(c)), 0.2) = 0.2\)
3. For \(c\): \(\Pi_P(\emptyset)(c) = 0.5\)
CHAPTER 3. FUZZY ANSWER SET PROGRAMMING

Proposition 3.7 (from [31]). Let \( P \) be a FASP program. If \( P \) is positive, the immediate consequence operator of \( P \) is monotonic, i.e. if \( I \subseteq I' \) then \( \Pi_P(I) \subseteq \Pi_P(I') \).

Due to the Tarski theorem introduced in Proposition 2.7 on page 25, the least fixpoint of the immediate consequence operator for simple FASP programs exists and is unique. For a given FASP program \( P \) the least fixpoint of \( \Pi_P \) is denoted as \( \Pi_P^\star \).

Proposition 3.8 (from [31]). Let \( P \) be a simple FASP program. The least fixpoint of \( \Pi_P \) exists and coincides with the unique minimal model of \( P \).

Due to Proposition 2.9 on page 26 this least fixpoint can be computed using an iterated fixpoint computation.

Definition 3.9. Let \( P \) be a positive FASP program. Formally, we define the sequence \( S(P) = \langle J_i \mid i \text{ an ordinal} \rangle \) by

\[
J_i = \begin{cases} 
\emptyset & \text{if } i = 0 \\
\Pi_P(J_{i-1}) & \text{if } i \text{ is a successor ordinal} \\
\bigcup_{j<i} J_j & \text{if } i \text{ is a limit ordinal}
\end{cases} \tag{3.2}
\]

where for all \( a \in B_P \) we have that \( \bigcup_{i \in I}(J_i)(a) = \sup_{i \in I} J_i(a) \).

The first element \( J_i \) in the sequence \( S(P) \) for which \( \Pi_P(J_i) = J_i \) then coincides with the least fixpoint of \( \Pi_P \).

Example 3.10. Consider program \( P_{\text{ex3.6}} \) from Example 3.6. In Example 3.6 we computed that \( \Pi_P(\emptyset) = \Pi_P(J_0) = J_1 = \{a^{0.8}, b^{0.2}, c^{0.5}\} \). For \( J_2 \) we obtain:

1. For \( a \): \( J_2(a) = \Pi_P(J_1)(a) = 0.8 \)
2. For \( b \): \( J_2(b) = \Pi_P(J_1)(b) = \max(\mathcal{T}_M(J_1(a), J_1(c)), 0.2) = \max(0.5, 0.2) = 0.5 \)
3. For \( c \): \( J_2(c) = \Pi_P(J_1)(c) = 0.5 \)

Hence, \( J_2 = \{a^{0.8}, b^{0.5}, c^{0.5}\} \). Since \( J_1 \neq J_2 \), \( J_1 \) is not the least fixpoint. Hence, we continue and compute \( J_3 \):

1. For \( a \): \( J_3(a) = \Pi_P(J_2)(a) = 0.8 \)
3.2. DEFINITIONS

2. For $b$: $J_3(b) = \Pi_P(J_2)(b) = \max(T_M(J_2(a), J_2(c)), 0.2) = \max(0.5, 0.2) = 0.5$

3. For $c$: $J_3(c) = \Pi_P(J_2)(c) = 0.5$

Hence $J_3 = J_2$, which means that $J_2$ is the least fixpoint of $\Pi_P$.

Note that, unlike ASP, the computation of this fixpoint may never end. This is illustrated in the next example.

Example 3.11 (from [160]). Consider the following FASP program $\text{Inf}$:

$$r: a \leftarrow \delta(a)$$

Where $\delta(x) = x + (1 - x)/2$. Obviously, $\delta$ is increasing and, moreover, $\forall x \in [0, 1] \cdot 0 < \delta(x) \leq 1$. Consider $\mathcal{S}(P)$. The first steps of the computation of the least fixpoint of $\Pi_{\text{Inf}}$ are shown below:

$$\begin{align*}
J_0 &= \{a^0\} \\
J_1 &= \Pi_P(J_0) = \{a^{0.5}\} \\
J_2 &= \Pi_P(J_1) = \{a^{0.75}\} \\
J_3 &= \Pi_P(J_2) = \{a^{0.875}\} \\
J_4 &= \Pi_P(J_3) = \{a^{0.9375}\} \\
J_5 &= \Pi_P(J_4) = \{a^{0.96875}\} \\
J_6 &= \Pi_P(J_5) = \{a^{0.984375}\} \\
... &= ...
\end{align*}$$

Clearly, $\forall i \in \mathbb{N} \cdot J_i \subset \{a^1\}$, but $J_\omega = \bigcup_{i \in \mathbb{N}} J_i = \{a^1\}$, which is the least fixpoint $\Pi_\omega$.

In [29] termination conditions for this operator are studied.

Using the above, we can introduce the semantics of FASP programs. Similar to ASP, the semantics of a FASP program $P$ are given by a certain subset of the models of $P$. Important to note is that we will view the truth values that are attached to atoms by these models as lower bounds. Rules in FASP then implement the intuition of (non-deterministic) forward chaining. In practice, we are therefore interested in
those models that are in accordance with this intuition. Effectively, there are two types of models we wish to exclude from our solution.

The first problem is the occurrence of atoms with a value above the one warranted by the rules. Consider the rule $r: a \leftarrow \alpha$. This rule will be satisfied by an interpretation $I$ whenever $I(a) \geq I(\alpha)$. Examples are the two models $M$ and $M'$, satisfying $M(a) = 1$, $M(\alpha) = 0.5 = M'(a)$ and $M'(\alpha) = 0.5$. However, the first model attaches a higher value to $a$ than what the rule actually supports (viz. 0.5) and is therefore unwanted. In other words, we do not want to conclude anything more than what is needed to satisfy the rule.

The second problem arises when atoms are “self-motivating”, i.e. their truth value is supported by some rule, but that support is ultimately based on the value of the atom itself. An illustration of this can be seen in the following two-rule program $P$:

$$
\begin{align*}
\text{r}_1: & \quad a \leftarrow b \\
\text{r}_2: & \quad b \leftarrow a
\end{align*}
$$

Both the models $M = \{a^1, b^1\}$ and $M' = \{a^0, b^0\}$ are free from the first problem we mentioned, but the support given to the value of $b$ is derived from the support for the value of $a$, which is itself derived from the value of $b$. Hence, only model $M'$ is free from knowledge not supported by the program.

The models that do not suffer from these defects will be called answer sets, and correspond to particular minimal models, as we will show later on. As the definition of answer sets for non-positive programs is an extension of the one for positive programs, we introduce them separately.

**Positive Programs**

**Definition 3.12.** Let $P$ be a positive FASP program. An interpretation $A$ is called the answer set of $P$ iff $A$ is the minimal model of $P$.

Intuitively the answer set of a positive FASP program corresponds to the maximal information we can derive by successively applying the immediate consequence operator, until no new knowledge can be discovered anymore, i.e. until a fixpoint is found. Note that not all positive programs have an answer set, as illustrated in the following example.
3.2. DEFINITIONS

**Example 3.13.** Consider the following positive FASP program $P_{ex3.13}$:

\[
\begin{align*}
  r_1 : & \ a \leftarrow 1 \\
  r_2 : & \ 0 \leftarrow a
\end{align*}
\]

One can easily see that $I = \{a^1\}$ is the least fixpoint of $\Pi_{P_{ex3.13}}$. However, $I$ is not an answer set of $P$ as $I(a) > 0$, meaning that it does not satisfy rule $r_2$. In fact, no models exist for this program as no interpretation $I'$ can satisfy both $I'(a) \geq 1$ and $0 \geq I'(a)$.

**General Programs**

In this section, we extend the definition of answer sets to cover arbitrary programs, similar to [113]. One might think that the semantics of non-positive programs could again be given by minimal models, but it turns out that minimal models are not at all suitable. For example, consider the following program:

\[
\begin{align*}
  r_1 : & \ a \leftarrow a \\
  r_2 : & \ 0 \leftarrow \lnot W(a)
\end{align*}
\]

The minimal model is $\{a^1\}$, but the motivation for $a$ depends on $a$ itself and thus $\{a^1\}$ is not acceptable. The underlying reason is that constraints should not be used as support of an atom, i.e. in (F)ASP, stating that $a$ is true is not at all the same as stating that solutions in which $a$ is false are not allowed.

To solve this problem, we reduce the semantics of such a program $P$ to that of a positive program $P^I$, which is called the reduct w.r.t. a candidate answer set $I$, similar to [31, 113]. Note that this generalizes the well-known Gelfond-Lifschitz transformation from [66].

**Definition 3.14.** Let $P$ be a FASP program. The **reduct** of a rule $(r : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m)) \in P$ w.r.t. an interpretation $I$ of $P$ is the positive rule $r^I$ defined as

\[ r^I = r : a \leftarrow f(b_1, \ldots, b_n; I(c_1), \ldots, I(c_m)) \]

The reduct of $P$ is the set of rules $P^I$ defined as

\[ P^I = \{r^I \mid r \in P\} \]
CHAPTER 3. FUZZY ANSWER SET PROGRAMMING

In other words, the reduct of a program w.r.t. an interpretation \( I \) is obtained by replacing all negatively occurring atoms by their value in \( I \).

Example 3.15. Consider program \( P_{ex3.3} \) from Example 3.3 on page 54. The reduct of \( P_{ex3.3} \) w.r.t. an interpretation \( I = \{ \text{white}(b)^{0.4}, \text{black}(a)^{1} \} \) is the following program \( P_{ex3.3}^I \):

\[
\begin{align*}
  f: & \text{white}(b) \leftarrow 0.4 \\
  r: & \text{black}(a) \leftarrow 0.6 \\
  c: & 0 \leftarrow \mathcal{T}_W(\text{white}(b), \text{white}(a))
\end{align*}
\]

To see that the above reduction generalizes the traditional Gelfond-Lifschitz (GL) transformation, it suffices to note that in traditional logic programming the only way to have a negative occurrence of an atom \( a \) in a rule body is via a negation-as-failure literal \( \neg a \). The GL transformation then essentially replaces such literals with their values in the intended stable model interpretation, yielding a positive program.

The semantics of a FASP program can now be defined in terms of the semantics of the positive reduct program.

Definition 3.16. Let \( P \) be a FASP program. An interpretation \( A \) of \( P \) is an answer set of \( P \) iff \( A \) is the answer set of \( P^A \).

Note that, in the boolean case, the idea of negation-as-failure is that \( \neg a \) is true for an atom \( a \) if during the application of forward chaining we fail to establish the truth of \( a \). Here, we generalize this taking as truth value for \( f(b_1, \ldots, b_n; c_1, \ldots, c_m) \) the highest possible value, i.e. by assuming for a candidate answer set \( A \) the lower bounds in \( A \) for \( \{c_1, \ldots, c_m\} \). Intuitively, \( A \) is then an answer set if under the above assumption, applying forward chaining again delivers \( A \).

Note that in general, FASP programs can have multiple answer sets or no answer sets, as shown in the following examples.

Example 3.17. Consider the following program \( P_{ex3.17} \):

\[
\begin{align*}
  r_1: & a \leftarrow \mathcal{N}_W(b) \\
  r_2: & b \leftarrow \mathcal{N}_W(a)
\end{align*}
\]
3.2. DEFINITIONS

It is easy to see that for any $l \in [0,1]$ we have that $\{a^l, b^{1-l}\}$ is an answer set of $P_{ex3.17}$. Hence, this program has an infinite number of answer sets (and therefore also models).

Example 3.18. Consider the following program $P_{ex3.18}$:

$$r : p \leftarrow N_M(p)$$

Now consider an interpretation $I = \{p^I\}$ with $l \in [0,1]$. By definition of $N_M$ we know that the reduct of $P_{ex3.18}$ is

$$r^I : p \leftarrow 0$$

Since the answer set of $P_{ex3.18}^I$ is $\emptyset$, we know that $I$ is not an answer set of $P_{ex3.18}$. Similarly we obtain that $\emptyset$ is not an answer set, meaning that this program has no answer sets.

Similar to ASP, the answer sets of a FASP program form a subset of the minimal models.

Proposition 3.19. Let $P$ be a FASP program. If $A$ is an answer set of $P$ it is a minimal model of $P$.

Proof. Suppose $A$ is an answer set of $P$, but not a minimal model of $P$. Then there must be some $M \subset A$ such that $M$ is a model of $P$.

Since $M \subset A$ we can easily see by definition of the reduct that for each $(r : a \leftarrow a) \in P$ we have $M(a^A) \leq M(a)$. Since $M$ is a model of $P$ we however know that $M(a) \geq M(a)$. From the foregoing it then follows that $M(a) \geq M(a^A)$. Hence $M$ is a model of $P^A$. This contradicts the assumption that $A$ is an answer set of $P$, from which the stated follows.

The reverse does not hold, as shown in the following example.

Example 3.20. Consider the following FASP program $P_{ex3.20}$:

$$r_1 : a \leftarrow a$$

$$r_2 : p \leftarrow T_W(N_W(p), N_W(a))$$
Now consider the interpretation $I = \{a^{0.2}, p^{0.4}\}$. We will show that it is a minimal model of $P_{\text{ex3.20}}$. First, it is clear that rule $r_1$ is trivially satisfied by $I$. Second, consider rule $r_2$. To satisfy this rule we need to have that $I(p) \geq T_W(N_W(I(p)), N_W(I(a)))$. By definition of $N_W$ and $T_W$ this means $I(p) \geq \max(1 - I(p) + 1 - I(a) - 1), 0)$, which does indeed hold. Hence, $I$ is a model of $P_{\text{ex3.20}}$. Now suppose there is some model $I'$ such that $I' \subset I$. Obviously $r_1$ is again trivially satisfied by $I'$. From the foregoing we know that for rule $r_2$ this interpretation then needs to satisfy the following inequation

$$I'(p) \geq \max(-I'(p) + 1 - I'(a), 0) \quad (3.3)$$

We consider three cases:

1. Suppose $I'(p) = I(p)$ and $I'(a) < I(a)$. Then (3.3) becomes $0.4 \geq \max(-0.4 + (1 - I'(a)), 0)$. Since $I'(a) \in [0, 0.2[\text{ we know that } (1 - I'(a)) \in [0.8, 1]$ and thus $(-0.4 + (1 - I'(a))) \in [0.4, 0.6[$, meaning (3.3) is not satisfied.

2. Suppose $I'(p) < I(p)$ and $I'(a) = I(a)$. Then (3.3) becomes $I'(p) \geq \max(-I'(p) + 0.8, 0)$. Since $I'(p) \in [0, 0.4[$ we know that $(-I'(p) + 0.8) \in [0.4, 0.8[$, meaning (3.3) is not satisfied.

3. Suppose $I'(p) < I(p)$ and $I'(a) < I(a)$. Then (3.3) becomes $I'(p) \geq \max(-I'(p) + (1 - I'(a)), 0)$. Since $I'(a) \in [0, 0.2[$ we know that $(1 - I'(a)) \in [0.8, 1]$. From this and the fact that $I'(p) \in [0, 0.4[\text{ it follows that } (-I'(p) + (1 - I'(a))) \in [0.4, 1]$, meaning (3.3) is not satisfied.

From the above it follows that no $I' \subset I$ exists that is a model of $P_{\text{ex3.20}}$, hence $I$ is a minimal model of $P_{\text{ex3.20}}$. Now let us check whether $I$ is an answer set of $P$. Consider the reduct $P_{\text{ex3.20}}^I$:

$$r_1^I: \quad a \leftarrow a$$

$$r_2^I: \quad p \leftarrow T_W(0.6, 0.8)$$

The minimal model of the reduct is $M = \{p^{0.4}\} \neq I$, hence $I$ is not an answer set of $P_{\text{ex3.20}}$.

It turns out that for constraint-free programs there is a correspondence with minimal fixpoints of the immediate consequence operator.
Proposition 3.21. Let $P$ be a constraint-free FASP program. If $M$ is an answer set of $P$, it is a minimal fixpoint of $\Pi_P$.

Proof. Suppose $M$ is an answer set of $P$. First we show that it must be a fixpoint of $\Pi_P$ and then that it must be a minimal fixpoint.

1. For any $a \in B_P$ we have:

$$
M(a) = \langle \text{Def. 3.16, Prop. 3.21} \rangle \Pi_P(M)(a) = \langle \text{Def. 3.5, Def. 3.14} \rangle \sup \{M(a^M) \mid (r : a \leftarrow a) \in P\} = \langle \text{Def. 3.5} \rangle \Pi_P(M)(a)
$$

where the fact that for any interpretation $M$ and rule $r : a \leftarrow a \in P$ we have that $M(a^M) = M(a)$ can easily be seen from the construction of the reduct in Definition 3.14. Hence, we can conclude that $M$ is a fixpoint of $\Pi_P$.

2. Suppose $M$ is not a minimal fixpoint of $\Pi_P$. Then there must be some $N \subset M$ such that $N = \Pi_P(N)$. We can show that $N$ is then a model of $P$:

$$
N = \Pi_P(N) \equiv \langle \text{Def. 3.5} \rangle \forall a \in B_P \cdot N(a) = \sup \{N(a) \mid (r : a \leftarrow a) \in P\} \\
\Rightarrow \langle (*) \rangle \forall a \in B_P \cdot \forall (r : a \leftarrow a) \in P \cdot N(a) \geq N(a) \\
\equiv \langle \text{Def. 3.4} \rangle N \text{ is a model of } P
$$

where the (*) justification is that the supremum is an upper bound and $\bigcup_{a \in B_P} P_a = P$ follows from the assumption that $P$ is constraint-free. Due to Proposition 3.19 this contradicts the fact that $M$ is an answer set of $P$.

As shown in the following example, the reverse does not hold, however.

Example 3.22. Consider program $P_{ex3.20}$ and its interpretation $I = \{a^{0.2}, p^{0.4}\}$ from Example 3.20 on page 61. It is not hard to see that $I$ is a fixpoint of $\Pi_{P_{ex3.20}}$. Now, suppose there is some interpretation $I'$ such that $I' \subset I$. Obviously $\Pi_{P_{ex3.20}}(I')(a) = I'(a)$. To obtain $\Pi_{P_{ex3.20}}(I')(p) = I'(p)$ the interpretation $I'$ needs to satisfy $I'(p) = T_W(N_W(p), N_W(a))$. As shown in Example 3.20,
there is no \( I' \) satisfying this equation, thus \( I \) is a minimal fixpoint of \( \Pi_p \). It is not an answer set, however, as was already shown in the aforementioned example.

### 3.3 Example: Fuzzy Graph Coloring

Consider a continuous graph coloring problem, where nodes can be colored using an infinite number of gray values, represented by a value between 0 (completely black) and 1 (completely white). The objective is to color a graph such that the colors of adjacent nodes are dissimilar to a degree that satisfies the weight of their connecting edge. We can model this problem using the following program \( P_{fgc} \):

\[
\begin{align*}
gen_1 : & \quad \text{white}(X) \leftarrow \mathcal{N}_W(\text{black}(X)) \\
\text{sim}_1 : & \quad \text{sim}(X,Y) \leftarrow T_M(I_M(\text{white}(X),\text{white}(Y)), I_M(\text{white}(Y),\text{white}(X))) \\
\text{sim}_2 : & \quad \text{sim}(X,Y) \leftarrow T_M(I_M(\text{black}(X),\text{black}(Y)), I_M(\text{black}(Y),\text{black}(X))) \\
\text{constr} : & \quad 0 \leftarrow T_W(\text{edge}(X,Y), \text{sim}(X,Y))
\end{align*}
\]

Similar to the ASP program modeling the black-and-white graph coloring in Example 2.27 on page 33, this program is written in a generate-define-test style. Rules \( gen_1 \) and \( gen_2 \) are the generate part, which create an arbitrary coloring of the graph. Rules \( \text{sim}_1 \) and \( \text{sim}_2 \) define the similarity degree of the colors of two nodes. Note that \( T_M(I_M(x,y), I_M(y,x)) \) is a generalization of the classical logic equivalence \( x \Leftrightarrow y \equiv (x \Rightarrow y) \land (y \Rightarrow x) \). Rule \( \text{constr} \) is a constraint that eliminates solutions where \( \text{sim}(X,Y) + \text{edge}(X,Y) \leq 1 \), i.e. where adjacent nodes are too similarly colored.

Consider now graphs \( G_{(a)} \) and \( G_{(b)} \) depicted in Figure 3.1 on page 66. To find a coloring of these graphs, we add facts to \( P_{fgc} \) that describe the graph and find the answer sets of the resulting program (after grounding). The facts \( F_{(a)} \), respectively \( F_{(b)} \) corresponding to these graphs are
3.3. EXAMPLE: FUZZY GRAPH COLORING

\[ n_a: \text{node}(a) \leftarrow 1 \]
\[ n_b: \text{node}(b) \leftarrow 1 \]
\[ f_{a,b}: \text{edge}(a, b) \leftarrow 0.5 \]

\[ n_a: \text{node}(a) \leftarrow 1 \]
\[ n_b: \text{node}(b) \leftarrow 1 \]
\[ f_{b,a}: \text{edge}(b, a) \leftarrow 0.5 \]

and

\[ n_a: \text{node}(a) \leftarrow 1 \]
\[ n_b: \text{node}(b) \leftarrow 1 \]
\[ n_c: \text{node}(c) \leftarrow 1 \]
\[ f_{a,b}: \text{edge}(a, b) \leftarrow 1 \]
\[ f_{b,a}: \text{edge}(b, a) \leftarrow 1 \]
\[ f_{a,c}: \text{edge}(a, c) \leftarrow 1 \]
\[ f_{c,a}: \text{edge}(c, a) \leftarrow 1 \]
\[ f_{b,c}: \text{edge}(b, c) \leftarrow 1 \]
\[ f_{c,b}: \text{edge}(c, b) \leftarrow 1 \]

An answer set for program \( gnd(P_{fgc} \cup F(a)) \) is

\[ A^{(a)} = \{ \text{node}(a)^1, \text{node}(b)^1, \text{edge}(a, b)^{0.5}, \text{edge}(b, a)^{0.5}, \text{white}(a)^1, \text{white}(b)^{0.5}, \]
\[ \text{black}(b)^{0.5}, \text{sim}(a, b)^{0.5}, \text{sim}(b, a)^{0.5} \} \]

This corresponds to a coloring where node \( a \) is white and node \( b \) is exactly gray (i.e. right in between entirely white and black). Of course, many other solutions are possible, and thus many more answer sets exist for this program. For example, the answer sets of the crisp graph coloring problem given in Example 2.27 on page 33 will also be answer sets of \( gnd(P_{fgc} \cup F(a)) \).

Now if we consider the program \( gnd(P_{fgc} \cup F(b)) \), we find that no answer sets exist. This is because the weights are very strict and prohibit any solution where \( \text{sim}(x, y) = 0 \) for any two nodes \( x \) and \( y \) (i.e. where two nodes are entirely dissimilar in color). In the next chapter we show how we can find approximate solutions for such problems and how we can model preference among the edges in other ways than by using weights.

To summarize, in this chapter we introduced the basic elements of fuzzy answer set programming (FASP). We showed how FASP can be used to model continuous problems in an elegant manner, similar to how ASP is capable of declaratively modeling combinatorial problems. We illustrated this on a fuzzy graph coloring problem, but also found that the language misses flexibility: sometimes no solutions could be found, whereas intuitively certain approximate solutions would certainly be admissible. In the next chapter we show how we can alleviate this problem.
Figure 3.1: Example instances for the graph coloring problem
4

Aggregated Fuzzy Answer Set Programming

4.1 Introduction

In the previous chapter we introduced FASP, an extension of ASP that allows to solve continuous problems in a concise, declarative manner. We have also shown that FASP can sometimes be limited in its flexibility, however. For example, the fuzzy graph coloring program $P_{fgc}$ introduced in Section 3.3 on page 64 allowed a continuous range of gray values as colors, but was unable to find a suitable graph coloring for graph $G_{(b)}$ depicted in Figure 3.1 on the preceding page. This is not ideal, as a coloring that colors node $a$ white, node $b$ black, and node $c$ gray may be better than having no solution at all. The inadmissibility is due to the constraint $constr$, which for $G_{(b)}$ removes solutions in which nodes have a similarity that is strictly greater than 0. A possible alternative is to allow solutions in which the similarity degree of two adjacent nodes may be greater than 0. Of course, solutions in which this degree is as small as possible are still preferred. This idea can be implemented by allowing that the last rule, $constr$, may not always be completely
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

satisfied.

Many of the current approaches (for example [28,29,52,95,96,103,104,112,113, 117, 118, 155]) that allow such partial rule satisfaction do so by coupling a residual implicator \( I \) and weight \( w \) to each rule \( r :: a \leftarrow \alpha \). A rule is then satisfied by an interpretation \( I \) if \( I(I(\alpha), I(a)) \geq w \). This means that the degree to which a rule should be satisfied is predefined.

Attaching weights to rules is not an entirely satisfactory solution, however. First of all, having weights puts an additional burden on the programmer, who, moreover, may not always be aware of which weights are suitable. Second, we are not only interested in finding any solution: if multiple solutions can be found, we are especially interested in the solution modeling the rules best. Hence, based on the degree to which the rules of the program are satisfied, it is of interest to define an ordering on the solutions, which cannot be meaningfully done using fixed weights. In [175] the proposed solution is to attach an aggregator expression to a program. This aggregator expression maps each prospective solution to a score, based on the satisfaction degrees of the rules. The latter is obtained by attaching an implicator to each rule, similar to the weighted approaches. We call this aggregated fuzzy answer set programming (AFASP). In this chapter, we further develop this approach. In particular, the main contribution of the chapter is two-fold. First, we decouple the order structure used by the aggregator expression from the lattice underlying the truth values. This ensures that we can define preference orderings on answer sets which may not correspond to complete lattices. Second, our approach is based on a fixpoint semantics, rather than unfounded sets. As we show below, the approach from [175] does not correctly generalize to arbitrary truth lattices, an issue which is solved by our proposed fixpoint semantics. In addition, the fixpoint semantics are also more general, as they are not restricted to formulas that are built from t-norms. Last, the fixpoint semantics (more clearly) reveal the link between FASP with an aggregator expression and FASP with weighted rules.

The structure of the chapter is as follows. In Section 4.2 we develop a fixpoint theory for fuzzy answer set programming with aggregators and investigate its main properties. While many of the properties we find are unsurprising, in the sense that they have a direct counterpart in fixpoint theory for (F)ASP, there are some notable differences as well. For instance, while one of the central properties of (F)ASP is that programs without negation have unique answer sets, it turns out that such programs can have several non-trivial answer sets in our setting, depending on the aggregator that is chosen. We apply AFASP on the reviewer assignment problem in Section 4.3, followed by a detailed overview of the relationship between AFASP
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

and existing approaches in Section 4.4.

4.2 Aggregated Fuzzy Answer Set Programming

In this section we introduce the concepts of AFASP. We begin by introducing the notion of approximate models, which do not satisfy all rules completely. Afterwards we define the semantics of AFASP programs.

4.2.1 Models

Normally, given a FASP program \( P \), one is interested in the interpretations \( I \) satisfying \( I(\alpha) \geq I(a) \) for each \( (r:a \leftarrow \alpha) \in P \). Such interpretations are called models. In the present framework, however, we recognize that rules cannot always be completely fulfilled. This has two main advantages: first, we can tackle problems lacking a “perfect” solution (i.e. a solution satisfying all rules) and second, we can find satisfactory solutions faster if we do not need a “perfect” solution.

The first situation can occur when problems are overconstrained. For example, consider the graph coloring problem introduced in Section 3.3 on page 64. For the graph depicted in Figure 3.1b on page 66, we found that no perfect coloring exists. Hence the problem is overconstrained and we must make some compromises, adhering more strictly to rules that are considered more important. The second situation occurs when approximate solutions can be computed substantially faster and when having a perfect solution is not crucial.

To define what it means for a rule of a FASP program \( P \) to be satisfied to a degree, we attach to each FASP rule \( (r:a \leftarrow \alpha) \in P \) a residual implicator, denoted \( I_r \). We also denote the t-norm of which \( I_r \) is the residual implicator with \( T_r \). An interpretation \( I \) of \( P \) is then said to satisfy \( r \) to the degree \( k \in \mathcal{L}_P \) iff \( I_r(I(\alpha), I(a)) \geq k \). Note that by Proposition 2.54 \( r \) is satisfied to degree 1 iff \( I(\alpha) \leq I(a) \), i.e. iff \( r \) is satisfied by \( I \) according to the definition of a satisfied FASP rule on page 54. Throughout this thesis we use \( r:a \leftarrow_m \alpha \), \( r:a \leftarrow_l \alpha \) and \( r:a \leftarrow_p \alpha \) to denote a rule \( r:a \leftarrow \alpha \) that is respectively associated with the Gödel, Łukasiewicz and product implicator. If no subscript is attached to \( \leftarrow \) either the associated implicator will be clear from the context, or will not be important in the given context.

The behavior of rules that are only partially satisfied depends crucially on the choice of the residual implicator. For example, rule \( r:a \leftarrow \alpha \) is satisfied exactly
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

to degree \( k < 1 \) for \( I(a) = k \) when the Gödel implicator is used, for \( I(a) = k \cdot I(\alpha) \) when the Goguen implicator is used and for \( I(a) = I(\alpha) - (1 - k) \) when the Łukasiewicz implicator is used. Thus, depending on the chosen implicator, the degree to which the head should be satisfied, depends 1) not at all on \( I(\alpha) \); 2) proportionally on \( I(\alpha) \); or 3) linearly on \( I(\alpha) \). Depending on the context, each of these three situations may be required.

To construct the AFASP theory, we first extend interpretations to rules.

**Definition 4.1.** Let \( P \) be a FASP program and let \( r : a \leftarrow \alpha \) be a rule in \( P \). We extend interpretations \( I \) of \( P \) to \( r \) as follows:

\[
I(r) = I_r(I(\alpha), I(a))
\]

To make the presentation clearer, we will often write \( I(a \leftarrow \alpha) \) to denote \( I(r) \) for a FASP rule \( r : a \leftarrow \alpha \). Now we can use this extension to define rule interpretations, which are functions that map rules to a truth value (i.e. a value from the considered lattice).

**Definition 4.2.** Given a FASP program \( P \), a rule interpretation of \( P \) is a \( P \rightarrow \mathcal{L}_P \) mapping. Any interpretation \( I \) of \( P \) induces a rule interpretation \( \rho_I \) defined as \( \rho_I(r) = I(a \leftarrow \alpha) \) for every rule \( r : a \leftarrow \alpha \) in \( P \).

Hence, the difference between an interpretation of a program and a rule interpretation is that the former maps propositional symbols to truth values, whereas the latter maps the rules themselves to a truth value. We define \( \rho_1 \leq \rho_2 \) iff \( \forall r \in P : \rho_1(r) \leq \mathcal{L}_P \rho_2(r) \). The relative importance of rules is encoded in an aggregator function.

**Definition 4.3.** An aggregator over a program \( P \) is an order-preserving \( (P \rightarrow \mathcal{L}_P) \rightarrow P \) function, where \( (P, \leq_P) \) is a preordered set.

Hence, an aggregator maps rule interpretations to preference scores from some preorder. As it is order-preserving, we guarantee that rule interpretations which map the rules to a higher degree receive a higher score. The aggregator typically
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

encodes which rules are deemed more important by the designer, who may be more reluctant to accept solutions that do poorly on some rules while not caring much about failing to fully satisfy others.

Example 4.4. Consider the fuzzy graph coloring program $P_{fgc}$ and the graph from Figure 3.1b introduced in Section 3.3 on page 64. Now, suppose that the dissimilarity of the colors of $a$ and $c$ is more important than the dissimilarity between the colors of $b$ and $c$, which is more important than the dissimilarity between the colors of $a$ and $b$. In terms of the graph this means that edge $(a, c)$ is more important than edge $(b, c)$, which is more important than edge $(a, b)$. If the generate rules or similarity rules are not satisfied to degree 1, the score of an interpretation of the rules should always be 0 since this means we are underestimating the similarity or color value, which is unwanted. If the generate rules and similarity rules are satisfied to degree 1 we can use the weighted sum of the constraint rules to represent the edge preference. Let us denote by $\text{constr}_{(x,y)}$ the grounding of the $\text{constr}$ rule where variable $X$ is replaced by constant $x$ and variable $Y$ by constant $y$. The following aggregator function over the preorder $\mathcal{P} = (\mathbb{R}, \leq)$ models this:

$$A(\rho) = \beta(\rho) \cdot \left( 1.4 \cdot \rho(\text{constr}_{(a,c)}) + 1 \cdot \rho(\text{constr}_{(b,c)}) + 0.8 \cdot \rho(\text{constr}_{(a,b)}) \right)$$

where

$$\beta(\rho) = \text{crisp}\left( \left( n_a \cdot n_b \cdot n_c \cdot f_{(a,b)} \cdot f_{(b,a)} \cdot f_{(a,c)} \cdot f_{(c,a)} \cdot f_{(b,c)} \cdot f_{(c,b)} \right) \cdot \left( \prod_{a \in \text{Nodes}} \rho((\text{gen}_1)_a) \cdot \rho((\text{gen}_2)_a) \cdot \rho(n_a) \right) \cdot \left( \prod_{(a,b) \in \text{Nodes}^2} \rho((\text{sim}_1)_{(a,b)}) \cdot \rho((\text{sim}_2)_{(a,b)}) \right) \right)$$

where $\text{crisp}(x) = 0$ if $x < 1$ and $\text{crisp}(x) = 1$ otherwise. Note that since $\text{constr}_{(x,y)} = \text{constr}_{(y,x)}$, for any $x, y \in \text{Nodes}$, we doubled the weights of the constraints to take the symmetry into account. Now consider two rule interpretations $\rho_1$ and $\rho_2$ of this program such that $\beta(\rho_1) = \beta(\rho_2) = 1$ and...
with the following values for the constraints:

\[
\begin{array}{c|c|c|c}
\text{constraint} & \text{constraint} & \text{constraint} \\
\hline
\rho_1 & 0.3 & 0.7 & 0.7 \\
\rho_2 & 1 & 0.5 & 0.5 \\
\end{array}
\]

Computing \( A(\rho_1) \) and \( A(\rho_2) \) we obtain \( A(\rho_1) = 1.92 \) and \( A(\rho_2) = 2 \). Hence according to this aggregator, a solution satisfying the rules to the degrees specified by rule interpretation \( \rho_2 \) is better than a solution satisfying the rules to the degrees specified by rule interpretation \( \rho_1 \). This corresponds to our intended semantics of the aggregator, since \( \rho_2 \) satisfies the most important edge \((a, b)\) to a much higher degree than \( \rho_1 \), whereas \( \rho_1 \) satisfies the less important edge \((a, c)\) only slightly better than \( \rho_2 \) and satisfies the least important edge \((b, c)\) a bit worse than \( \rho_2 \).

However, depending on the application, this might not be what we want. For example, the fact that edge \((a, c)\) is more important can also mean that \( \rho_1 \) should be more preferred than \( \rho_2 \), since it satisfies this rule better. This can be encoded using an aggregator over the partial order \( \mathcal{P}' = ([0, 1]^3, \leq_{\text{lex}}) \) where by definition \((a_1, a_2, a_3) \leq_{\text{lex}} (a_1', a_2', a_3') \) iff

\[
\begin{align*}
& (a_1 < a_1') \lor (a_1 = a_1' \land a_2 < a_2') \lor (a_1 = a_1' \land a_2 = a_2' \land a_3 < a_3') \\
& \quad \lor (a_1 = a_1' \land a_2 = a_2' \land a_3 = a_3')
\end{align*}
\]

The corresponding aggregator is

\[
A'(\rho) = \begin{cases} 
(0, 0, 0) & \text{if } \beta(\rho) = 0 \\
(\rho(\text{constraint}(a, c)), \rho(\text{constraint}(b, c)), \rho(\text{constraint}(a, b))) & \text{otherwise}
\end{cases}
\]

Note that due to the symmetry between the constraints we do not need to take the constraints \( \text{constraint}(b, a) \), \( \text{constraint}(c, b) \) and \( \text{constraint}(c, a) \) into account. Using \( A' \), we obtain \( A'(\rho_1) = (0.7, 0.7, 0.3) \) and \( A'(\rho_2) = (0.5, 0.5, 1) \), from which we obtain that \( A'(\rho_2) \leq_{\text{lex}} A'(\rho_1) \), i.e. \( \rho_1 \) is strictly preferred over \( \rho_2 \).

Many aggregation strategies have been proposed over the years (for an overview see [41]). Formally, an aggregation operator \( A \) is defined as a function mapping vectors over \( \mathcal{L}^n \), with \( \mathcal{L} \) a complete lattice, to a preordered set \( \mathcal{P} \). It can easily be seen that the aggregator defined in Definition 4.3 above fits this definition as
rule interpretations correspond to vectors in $\mathcal{L}^n$, with $n$ the number of rules in a program. In the following, let $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ be two vectors in $\mathcal{L}^n$. The main task of an aggregation operator is to define an ordering over the vectors in $\mathcal{L}^n$, which we call the induced ordering of an aggregation operator $A$. Formally for an aggregation operator $A$ it is defined as

$$u \leq_A v \equiv A(u) \leq_P A(v)$$

We can distinguish two types of aggregators. In the first case the aggregator maps vectors in $\mathcal{L}^n$ to a single value in $\mathcal{L}$ or $\mathbb{R}$. Well-known aggregation operators of this form are the minimum, maximum, median, product and sum. Though these operators are useful, sometimes, we may consider the satisfaction of some rules to be more important, e.g. expressed using priority levels for each rule. To cope with such priority levels, weighted versions of the basic operators have been proposed, such as the weighted sum used in Example 4.4. For weighted minimum and maximum we refer the reader to [45, 55, 179]. A particularly interesting class of operators with weights are the Ordered Weighted Average (OWA) operators (see e.g. [180, 181]), which encompass a wide range of aggregation operators over vectors in $[0,1]^n$, including the minimum, maximum and median. Formally, given a vector $u = (u_1, \ldots, u_n) \in [0,1]^n$, a collection of weights $(w_1, \ldots, w_n) \in [0,1]^n$ such that $\sum_i w_i = 1$ and a permutation $\sigma$ of $u$ such that $u_{\sigma(1)} \leq \cdots \leq u_{\sigma(n)}$, an OWA operator is defined by

$$\text{OWA}(u) = \sum_{j=1}^n w_j u_{\sigma(j)}$$

(4.2)

By manipulating the weights of the OWA operator, particular aggregation operators are obtained, with the minimum and the maximum as extreme cases, corresponding to the weight vectors $(1,0,\ldots,0)$ and $(0,\ldots,0,1)$ respectively. Note that the aggregated value will always be between the minimum and the maximum of their arguments. Interestingly, the weights of an OWA operator are not associated to a component of the vector, but to an ordered position. This makes it ideal for applications where certain outliers should not be taken into account, such as for example in the judging of some Olympic sports, where the most extreme scores do not count for the final score. Furthermore OWAs can be used to model the demand that most of the rules should be fulfilled, or at least a few rules should be fulfilled.

Two other families of aggregation operators allow to model interaction between values, viz. the discrete Sugeno integral [166] and the Choquet integral [25]. The difference between these aggregators is that the Sugeno integral is more suitable
for ordinal aggregation (where only the order of elements is important) while the Choquet integral is suitable for cardinal aggregation (where the distance between the numbers has a meaning) [41]. Interestingly, the Sugeno integral generalizes the weighted minimum and the weighted maximum, and the Choquet integral generalizes the weighted mean and the OWA operators. The downside of using these operators is the high number of weights that need to be provided by the user. To aggregate $n$ values, in principle, the user needs to supply $2^n$ weights, which clearly is rather cumbersome. However, in some cases one can reduce the number of required weights; see for example [70, 131].

One can also use t-norms and t-conorms for aggregation. However, the aggregated value of these operators is not in between the minimum and the maximum, though this can be useful in certain applications. Two classes of operators that do have this feature can be constructed based on t-norms and t-conorms, viz. the exponential compensatory operators [170] and the convex-linear compensatory operators [108, 170]. Another related class of operators are uninorms [60], which generalize t-norms and t-conorms. Contrary to the two aforementioned compensatory operators, uninorms satisfy the full reinforcement property, i.e. the tendency that when we collect a number of high scores, the aggregated value will be greater than the maximum of these scores, and similarly when we collect a number of low scores, the aggregated value will be lower than the minimum of these scores. In some cases this follows the human aggregation process more closely.

The second type of aggregators are those where the aggregator function is the identity function, and thus $u \leq_A v \equiv u \leq_P v$, with $\leq_P$ an ordering over vectors in $L^n$. This is for example the case with the Pareto aggregator and lexicographical aggregator. Formally the former orders the vectors of $L^n$ with $\leq_{par}$ defined as

$$u \leq_{par} v \equiv \forall i \in \{1, \ldots, n\} \cdot u_i \leq v_i$$

The latter orders the vectors of $L^n$ with the preorder $\leq_{lex}$ defined as

$$u \leq_{lex} v \equiv (u \leq_{par} v) \lor (\exists i \in \{1, \ldots, n\} \cdot (u_i < v_i) \land (\forall j < i \cdot u_j = v_j))$$

Although the min and max operators are useful because they work in a strictly ordinal manner, the ordering induced by these operators can sometimes be too coarse. For example, using the minimum, the vectors $(0.1, 0.5)$ and $(0.1, 0.1)$ would be equally preferred, as $\min(0.1, 0.5) = \min(0.1, 0.1)$. However, the first vector clearly has a better score for the second value to be aggregated. To cope with this, refinements of the induced ordering have been proposed, namely the discrimin and
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

the leximin (see e.g. [46]). Formally the discrimin aggregator orders vectors in \( L^n \)
using \( \leq_{\text{disc}} \) defined as

\[
\begin{align*}
  u \leq_{\text{disc}} v &\equiv \min \{ u_i \mid i \in D(u, v) \} \leq \min \{ v_i \mid i \in D(u, v) \} \\
&= \min \{ u_i \mid i \in \{1, \ldots, n\}, u_i \neq v_i \}.
\end{align*}
\]

where \( D(u, v) = \{ i \mid i \in \{1, \ldots, n\}, u_i \neq v_i \} \). Intuitively, this ordering is based
on the idea that the values on which the two vectors agree are of no importance
when comparing them. Decisions are thus based on the least satisfied discriminating
value. The idea of the leximin aggregator is to represent vectors of satisfaction levels
by ranked multi-sets of satisfaction degrees. Formally it maps a vector in \( L^n \) to the
corresponding element in the structure \( (L^n, \leq_{\text{lexi}}) \) defined as

\[
\begin{align*}
  u \leq_{\text{lexi}} v &\equiv \exists k \leq n \cdot \forall i < k \cdot (u_{\sigma(i)} = v_{\pi(i)}) \land (u_{\sigma(k)} < v_{\pi(k)}) \\
&\lor \forall i \in \{1, \ldots, n\} \cdot u_i = v_i
\end{align*}
\]

where \( \sigma \) and \( \pi \) are permutations of \( u \), resp. \( v \) such that \( u_{\sigma(1)} \leq \ldots \leq u_{\sigma(n)} \)
and \( v_{\pi(1)} \leq \ldots \leq v_{\pi(n)} \). Hence, two vectors are indifferent if the corresponding
reordered vectors are the same. The leximin ordering is a refinement of both the
minimum and the discrimin [42]. Similarly two refinements of the maximum, called
discrimax and leximax, can be defined.

Of course it is also possible to combine a non-trivial aggregation function \( A \)
with a non-trivial ordering, as we have for example done in Example 4.4.

An aggregated FASP program then consists of a FASP program and an aggre-
gator function over this program.

**Definition 4.5.** An aggregated FASP program (AFASP program) \( P \) is a
tuple \( \langle R, A \rangle \), where \( R \) is a FASP program over a lattice \( L \), called the rule
base, and \( A \) is an aggregator function over \( R \). Given an AFASP program \( P \)
we denote its rule base as \( R_P \), its aggregator as \( A_P \), the lattice over which
\( R_P \) is defined as \( L_P \) and the preorder used by the aggregator as \( P_P \). The set
\( B_P = B_{R_P} \) is called the Herbrand base of \( P \). Furthermore, we define the set
\( P_a \), for \( a \in B_P \), as \( P_a = \{ r \mid r \in R_{Pa} \} \). Last, a rule interpretation of an
AFASP program is a rule interpretation of \( R_P \).

In the remainder of this chapter the term program always refers to an AFASP
program. Furthermore, we will use the term interpretation of a program for
the interpretation of the rule base of the program. Similar to FASP we can divide an
AFASP program in different classes:
1. An AFASP program $P$ is called \textbf{constraint-free} if $\mathcal{R}_P$ is constraint-free.

2. An AFASP program $P$ is called \textbf{positive} if $\mathcal{R}_P$ is positive, i.e. if every rule body is increasing in its arguments.

3. An AFASP program $P$ is called \textbf{simple} if $\mathcal{R}_P$ is simple, i.e. if every rule is positive and not a constraint.

Finally, we introduce two types of \textit{approximate models}: $\rho$-\textit{rule models}, which relate interpretations of the program to rule interpretations, and $k$-\textit{models}, which are interpretations that induce rule interpretations whose score is at least $k$.

\begin{definition}
Let $P = \langle \mathcal{R}_P, \mathcal{A}_P \rangle$ be an AFASP program. A $\rho$-\textbf{rule model}, with $\rho$ a rule interpretation of $P$, is an interpretation $I$ of $\mathcal{R}_P$ such that $\rho_1 \geq \rho$. A $\mathbf{k}$-\textbf{model}, $k \in \mathcal{P}_F$, of $P$ is any interpretation $I$ satisfying $\mathcal{A}_P(\rho_1) \geq k$. Lastly, we define the values $\min(P) = \mathcal{A}_P(\rho_{\bot})$ and $\max(P) = \mathcal{A}_P(\rho_{\top})$, where $\forall r \in \mathcal{R}_P \cdot \rho_{\bot}(r) = 0$ and $\forall r \in \mathcal{R}_P \cdot \rho_{\top}(r) = 1$. Intuitively these correspond to the minimal, resp. maximal value the aggregator expression can attain.
\end{definition}

Obviously, any $\rho_1$-rule model $M$, with $\rho_1$ some rule interpretation, is also a $\rho_2$-rule model when $\rho_2 \leq \rho_1$. Similarly any $k_1$-model $M$ of a program $P$, with $k_1 \in \mathcal{P}_F$, is also a $k_2$-model when $k_2 \leq P, k_1$.

\begin{example}
Consider program $P_{fgc}$ and the graph depicted in Figure 3.1b on page 66, together with rule interpretations $\rho_1$ and $\rho_2$ from Example 4.4 on page 71. Furthermore, for this example we interpret the rules using the Łukasiewicz implication. To have a $\rho_1$-rule model, we need an interpretation $I$ such that $\rho_1 \geq \rho_1$. Now consider

$I_1 = \{\text{edge}(a, b)^1, \text{edge}(b, a)^1, \text{edge}(a, c)^1, \text{edge}(c, a)^1, \text{edge}(b, c)^1, \text{edge}(c, b)^1,$

\hspace{1cm} white(a)^1, white(b)^0.7, black(b)^0.3, white(c)^0.7, sim(a, b)^0.7,$

\hspace{1cm} sim(b, a)^0.7, sim(a, c)^0.3, sim(c, a)^0.3, sim(b, c)^0.3, sim(c, b)^0.3, sim(a, a)^1,$

\hspace{1cm} sim(b, b)^1, \text{sim}(c, c)^1\}$

Clearly $I_1(r) = 1$ for every rule not in $\{\text{constr}_{(x, y)} | x, y \in \text{Nodes}\}$, hence $I_1(r) \geq \rho_1(r)$. For the constraint rules we obtain:

$I_1(\text{constr}_{(a, b)}) = I_W(T_W(1, 0.7), 0) = 0.3 = \rho_1(\text{constr}_{(a, b)})$
\end{example}
Likewise for the other constraint rules we obtain that $I_1(\text{constr}_{x,y}) = \rho_1(\text{constr}_{x,y})$ for any $x, y \in \text{Nodes}$. Hence $I_1$ is a $\rho_1$-rule model of $P_{fgc}$. We can also verify that $I_1$ is not a $\rho_2$-rule model of $P_{fgc}$ as $I_1(\text{constr}_{a,b}) = 0.3 < \rho_2(\text{constr}_{b,c})$. Consider now

$$I_2 = \{\text{edge}(a, b)^1, \text{edge}(b, a)^1, \text{edge}(a, c)^1, \text{edge}(c, a)^1, \text{edge}(b, c)^1, \text{white}(a)^1, \text{black}(b)^1, \text{white}(c)^{0.5}, \text{black}(c)^{0.5}, \text{sim}(a, c)^{0.5}, \text{sim}(c, a)^{0.5}, \text{sim}(b, c)^{0.5}, \text{sim}(c, b)^{0.5}, \text{sim}(a, a)^1, \text{sim}(b, b)^1, \text{sim}(c, c)^1\}$$

Again one can easily verify that $I_2(r) = 1$ for any rule not in $\{\text{constr}_{x,y} \mid x, y \in \text{Nodes}\}$. For the constraint rules we obtain:

$$I_2(\text{constr}_{a,b}) = I_W(T_W(1, 0), 0) = 1 \geq \rho_2(\text{constr}_{a,b})$$

Likewise for every other constraint rule $\text{constr}_{x,y}$ with $x, y \in \text{Nodes}$ we obtain $I_2(\text{constr}_{x,y}) \geq \rho_2(\text{constr}_{x,y})$. Hence $I_2$ is a $\rho_2$-rule model of $P_{fgc}$.

Now consider the aggregators $\mathcal{A}$ and $\mathcal{A}'$ from Example 4.4 on page 71. Using $\mathcal{A}$ we obtain that $I_1$ is an $1.92$-model and $I_2$ is a $2$-model of $P_{fgc}$, hence $I_2$ is preferred over $I_1$. However, with $\mathcal{A}'$ we obtain that $I_1$ is an $(0.7, 0.7, 0.3)$-model of $P_{fgc}$, whereas $I_2$ is an $(0.5, 0.5, 1)$-model of $P_{fgc}$. This means that $\rho_1 I_2 \leq \text{lex} \rho_1 I_1$, and thus with this aggregator interpretation $I_1$ is preferred over $I_2$ since it satisfies the more important rules to a better degree.

Hence, the above example shows that by adding an aggregator function to FASP we can order models according to how well they satisfy the program rules.

### 4.2.2 Answer Sets

In this section we introduce $k$-answer sets of AFASP programs $P$, which are approximations of the answer sets of the FASP program $R_P$. Similar to FASP the semantics of general AFASP programs is based on those of positive AFASP programs. Therefore we begin by introducing the semantics of the latter.

#### Positive Programs

To define the answer sets of positive programs we need to extend concepts from FASP to deal with partial rule satisfaction. First we introduce the support of a rule.
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

w.r.t. some truth value representing the minimal degree to which this rule should be satisfied. This captures the idea of partial rule application. Using the support, we then extend the immediate consequence operator in such a way that we can derive knowledge from a program in a forward chaining manner, while still allowing for partial rule satisfaction. Afterwards we define answer sets of positive programs and present a number of propositions.

Definition 4.8 (from [175]). Let \( r : a \leftarrow \alpha \) be a rule defined on the lattice \( L \) and let \( I \) be an interpretation of \( r \). The support of this rule w.r.t. \( I \) and some \( c \in L \) is denoted as \( I_s(r,c) \) and is defined by

\[
I_s(r,c) = \inf \{ y \in L | I_r(I(\alpha),y) \geq c \}
\]

In practice the weight \( c \) will mostly be determined using a rule interpretation. It turns out that a characterization of this operator is easy to find.

Proposition 4.9. Let \( r : a \leftarrow \alpha \) be a rule defined on the lattice \( L \), \( I \) an interpretation of \( r \), and \( c \) a value in \( L \). Then \( I_s(r,c) = T_r(I(\alpha),c) \).

Proof. Suppose \( r : a \leftarrow \alpha \) is defined over a lattice \( L \), then:

\[
I_s(r,c) = \langle \text{Def. } I_s(r,c) \rangle \inf \{ y \in L | I_r(I(\alpha),y) \geq c \}
= \langle \text{Residuation principle} \rangle \inf \{ y \in L | y \geq T_r(I(\alpha),c) \}
= \langle \text{See below} \rangle T_r(I(\alpha),c)
\]

The last step follows from the fact that \( T_r(I(\alpha),c) \) is an element of \( L \) and is a lower bound of \( \{ y \in L | y \geq T_r(I(\alpha),c) \} \).

Note that this property is only valid when the partial mappings of \( T_r \) are supmorphisms, which we assumed in Chapter 2.

Example 4.10. Consider the rule \( r : a \leftarrow_m \alpha \) with interpretations \( I \) and \( I' \) satisfying \( I(\alpha) = I'(\alpha) = 0.5 \), \( I(a) = 1 \) and \( I'(a) = 0.5 \). Recall that for a rule of the aforementioned form we defined that \( I_r = I_M \). The support of this rule w.r.t. \( \rho_T(r) \) is given by \( I_s(r,\rho_T(r)) = T_M(0.5,1) = 0.5 \). Likewise we can compute that \( I'_s(r,\rho_T(r)) = T_M(0.5,1) = 0.5 \). Hence, \( I(a) > I_s(r,\rho_T(r)) \) and \( I'(a) = I'_s(r,\rho_T(r)) \). This means that the interpretation of \( a \) by \( I' \) is
consistent with the support provided by the rule, whereas the interpretation by I is strictly greater. Hence, rule r cannot be used to justify the value of a under interpretation I. Likewise we can see that I_s(r, 0.9) < I(a), meaning I attaches a value to a that is higher than the support that is needed to satisfy this rule to a degree of 0.9.

The support is monotonic, as shown in the following proposition.

**Proposition 4.11.** Let I_1 and I_2 be interpretations of a simple rule r: a ← α defined on a lattice L. If I_1 ≤ I_2 we have for any c ∈ L that (I_1)_s(r, c) ≤ (I_2)_s(r, c).

**Proof.** Using Proposition 4.9 and the monotonicity of t-norms we can easily see that:

\[(I_1)_s(r, c) = T_r(I_1(α), c) \leq T_r(I_2(α), c) = (I_2)_s(r, c)\]

For simple AFASP programs, answer set semantics are based on a “forward chaining” approach, captured in the definition of an extended version of the immediate consequence operator (Definition 3.5 on page 55). This operator ensures that the support of a rule is propagated to its head, which means that we derive exactly the maximal amount of knowledge contained in the program.

**Definition 4.12.** Let P be an AFASP program with ρ a rule interpretation of P. The immediate consequence operator Π_{P, ρ} derived from P and ρ is a mapping from \((B_P \rightarrow L_P)\) to \((B_P \rightarrow L_P)\) defined for any interpretation I of P and a ∈ B_P as:

\[Π_{P, ρ}(I)(a) = \sup\{I_s(r, ρ(r)) \mid r \in P_a\}\]

The following example illustrates the use of this operator.
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

Example 4.13. Let $P$ be an AFASP program with the following rule base $\mathcal{R}_P$:

$$
\begin{align*}
  r_1 &: a \leftarrow_m 0.8 \\
  r_2 &: c \leftarrow_m 0.5 \\
  r_3 &: b \leftarrow_m T_M(a, c) \\
  r_4 &: b \leftarrow_m 0.2
\end{align*}
$$

and aggregator function $A_P(\rho) = \inf\{\rho(r) \mid r \in \mathcal{R}_P\}$. Consider now the interpretation $\emptyset$, which attaches to each atom the value 0. When using the rule interpretation $\rho^\top$ we can compute $\Pi_{P,\rho^\top}(\emptyset)$ as follows (note that we are using Proposition 4.9 in the computation):

1. For $a$: $\Pi_{P,\rho^\top}(\emptyset)(a) = \sup\{\emptyset_s(r, \rho^\top(r)) \mid r \in P_a\} = \emptyset_s(r_1, \rho^\top(r_1)) = T_M(0.8, 1) = 0.8$
2. For $b$: $\Pi_{P,\rho^\top}(\emptyset)(b) = \sup\{\emptyset_s(r, \rho^\top(r)) \mid r \in P_b\} = \sup\{\emptyset_s(r_3, \rho^\top(r_3)), \emptyset_s(r_4, \rho^\top(r_4))\} = \sup\{T_M(0.2, 1)\} = \sup\{0.2\} = 0.2$
3. For $c$: $\Pi_{P,\rho^\top}(\emptyset)(c) = \sup\{\emptyset_s(r, \rho^\top(r)) \mid r \in P_c\} = T_M(0.5, 1) = 0.5$

Hence $\Pi_{P,\rho^\top}(\emptyset) = \{a^{0.8}, b^{0.2}, c^{0.5}\}$. For rule interpretation $\rho = \{r_1^{0.5}, r_2^{0.3}, r_3^{1}, r_4^{1}\}$ the computation of $\Pi_{P,\rho}(\emptyset)$ is as follows:

1. For $a$: $\Pi_{P,\rho}(\emptyset)(a) = \emptyset_s(r_1, \rho(r_1)) = T_M(0.8, 0.5) = 0.5$
2. For $b$: $\Pi_{P,\rho}(\emptyset)(b) = \sup\{\emptyset_s(r_3, \rho(r_3)), \emptyset_s(r_4, \rho(r_4))\} = \sup\{T_M(0.2, 1)\} = \sup\{0.2\} = 0.2$
3. For $c$: $\Pi_{P,\rho}(\emptyset)(c) = \emptyset_s(r_2, \rho(r_2)) = T_M(0.5, 0.3) = 0.3$

Hence $\Pi_{P,\rho}(\emptyset) = \{a^{0.5}, b^{0.2}, c^{0.3}\}$.

This immediate consequence operator is similar to the one proposed by Damásio et al. in [29]. The difference is that we add the weights of the program as a parameter of the operator, where in [29] the weights of the program are fixed in the program itself. The resulting dynamicity of the weights is crucial for our aggregation based framework. However, once we have chosen a particular rule interpretation the two operators coincide. As in [29], our operator is monotonic for simple programs:

**Proposition 4.14.** Let $P$ be a positive AFASP program and $\rho$ a rule interpreta-
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

The immediate consequence operator $\Pi_{P,\rho}$ is monotonically increasing, i.e. for every two interpretations $I_1$ and $I_2$ it holds that

$$I_1 \leq I_2 \Rightarrow \Pi_{P,\rho}(I_1) \leq \Pi_{P,\rho}(I_2)$$

**Proof.** Follows from Theorem 16 in [29].

The following proposition shows that our operator is also monotonic in the rule interpretations, something that is also illustrated in Example 4.13.

**Proposition 4.15.** Let $P$ be a positive AFASP program. The immediate consequence operator is monotonically increasing in the rule interpretations, i.e. for any two rule interpretations $\rho_1$ and $\rho_2$ and interpretation $I$ of $P$ it holds that

$$\rho_1 \leq \rho_2 \Rightarrow \Pi_{P,\rho_1}(I) \leq \Pi_{P,\rho_2}(I)$$

**Proof.**

\[
\Pi_{P,\rho_1}(I)(a) = \langle \text{Def. } \Pi_{P,\rho_1} \rangle \sup\{I_s(r, \rho_1(r)) \mid r \in P_a\} \\
= \langle \text{Prop. 4.9} \rangle \sup\{T_r(I(r_b), \rho_1(r)) \mid r \in P_a\} \\
\leq \langle \text{Monot. t-norm} \rangle \sup\{T_r(I(r_b), \rho_2(r)) \mid r \in P_a\} \\
= \langle \text{Prop. 4.9} \rangle \sup\{I_s(r, \rho_2(r)) \mid r \in P_a\} \\
= \langle \text{Def. } \Pi_{P,\rho_2} \rangle \Pi_{P,\rho_2}(I)(a)
\]

Due to Proposition 2.7 on page 25 it follows that our immediate consequence operator has a least fixpoint $\Pi_{P,\rho}^\star$ for any positive AFASP program $P$ and rule interpretation $\rho$. Similar to ASP and FASP, this least fixpoint can in principle be computed using the iterated fixpoint computation introduced in Definition 2.8 on page 25.

**Definition 4.16.** Let $P$ be a positive AFASP program, and let $\rho$ be a rule interpretation of $P$. Formally, we define the sequence $S(P, \rho) = \langle J_i \mid i \text{ an ordinal} \rangle$ by

$$J_i = \begin{cases} 
\emptyset & \text{if } i = 0 \\
\Pi_{P,\rho}(J_{i-1}) & \text{if } i \text{ is a successor ordinal} \\
\bigcup_{j<i} J_j & \text{if } i \text{ is a limit ordinal}
\end{cases} \quad (4.3)$$

where $\bigcup_{i\in I} J_i = \sup_{i\in I} J_i$. 

81
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

The least fixpoint of $\Pi_{P,\rho}$ is the first element $J_i$ in the sequence $S\langle P, \rho \rangle$ for which $\Pi_{P,\rho}(J_i) = J_i$.

**Example 4.17.** Consider program $P$ from Example 4.13. If we apply $\Pi_{P,\rho_T}$ to the interpretation $J_1 = \Pi_{P,\rho_T}(\emptyset) = \{a^{0.8}, b^{0.2}, c^{0.5}\}$ of the sequence $S\langle P, \rho_T \rangle$ we obtain $J_2$:

1. For $a$: $J_2(a) = \Pi_{P,\rho_T}(J_1)(a) = \sup\{(J_1)_s(r, \rho_T(r)) \mid r \in P_a\} = (J_1)_s(r_1, \rho_T(r_1)) = T_M(0.8, 1) = 0.8 = J_1(a)$
2. For $b$: $J_2(b) = \Pi_{P,\rho_T}(J_1)(b) = \sup\{(J_1)_s(r, \rho_T(r)) \mid r \in P_b\} = \sup\{(J_1)_s(r_3), (J_1)_s(r_4)\} = \sup\{J_1(T_W(T_M(a, c), 1)), T_M(0.2, 1)\} = \sup\{0.5, 0.2\} = 0.5$
3. For $c$: $J_2(c) = \Pi_{P,\rho_T}(J_1)(c) = \sup\{(J_1)_s(r, \rho_T(r)) \mid r \in P_c\} = (J_1)_s(r_2, \rho_T(r_2)) = T_M(0.5, 1) = 0.5$

Hence $J_2 = \{a^{0.8}, b^{0.5}, c^{0.5}\}$. One can readily verify that $J_3 = \Pi_{P,\rho_T}(J_2) = J_2$. Hence $J_2$ is a fixpoint of $\Pi_{P,\rho_T}$ and, as it is the first element of the sequence $S\langle P, \rho_T \rangle$ that is a fixpoint of $\Pi_{P,\rho_T}$, it is its least fixpoint. In other words $J_2 = \Pi_{P,\rho_T}^*$.

**Example 4.18.** Consider program $P$ from Example 4.13 with the rule interpretation $\rho = \{r_1^{0.5}, r_2^{0.3}, r_3^{0.1}\}$. If we apply $\Pi_{P,\rho}$ to the interpretation $J_1 = \Pi_{P,\rho}(\emptyset) = \{a^{0.5}, b^{0.2}, c^{0.3}\}$ of the sequence $S\langle P, \rho \rangle$ we obtain $J_2$:

1. For $a$: $J_2(a) = \Pi_{P,\rho}(J_1)(a) = \sup\{(J_1)_s(r, \rho(r)) \mid r \in P_a\} = (J_1)_s(r_1, \rho(r_1)) = T_M(0.8, 0.5) = 0.5$
2. For $b$: $J_2(b) = \Pi_{P,\rho}(J_1)(b) = \sup\{(J_1)_s(r, \rho(r)) \mid r \in P_b\} = \sup\{(J_1)_s(r_3, \rho(r_3)), (J_1)_s(r_4, \rho(r_4))\} = \sup\{J_1(T_W(T_M(a, c), 1)), T_M(0.2, 1)\} = \sup\{0.3, 0.2\} = 0.3$
3. For $c$: $J_2(c) = \Pi_{P,\rho}(J_1)(c) = \sup\{(J_1)_s(r, \rho(r)) \mid r \in P_c\} = (J_1)_s(r_2, \rho(r_2)) = T_M(0.5, 0.3) = 0.3$

Again one can readily verify that $\Pi_{P,\rho}(J_2) = J_2$ and hence $J_2$ is the least fixpoint of $\Pi_{P,\rho}$, i.e. $J_2 = \Pi_{P,\rho}^*$. Note that $\rho \leq \rho_T$ and $\Pi_{P,\rho} \leq \Pi_{P,\rho_T}$, i.e. rule interpretations that put stricter requirements on the satisfaction of rules will lead to greater fixpoints.
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

Similar to FASP, the least fixpoint may not necessarily be found after a finite number of steps, as can easily be seen by combining program \( \text{Inf} \) in Example 3.11 on page 57 with rule interpretation \( r_\top \).

The following proposition shows us that smaller rule interpretations yield smaller (least) fixpoints. Hence if we increase the lower bounds imposed on the rules of a program \( P \), the resulting knowledge that we can derive from \( P \) using forward chaining increases as well. This corresponds to our intuition, as in general derivable knowledge monotonically increases with tighter constraints.

**Proposition 4.19.** Let \( \rho_1 \leq \rho_2 \) be rule interpretations of a positive AFASP program \( P \). Then \( \Pi^\star_{P,\rho_1} \leq \Pi^\star_{P,\rho_2} \).

**Proof.** It is straightforward to show, using transfinite induction and Propositions 4.14 and 4.15, that, for any ordinal \( i \), \( J^1_i \leq J^2_i \), where \( S(P,\rho_1) = \{ J^1_i \mid i \text{ an ordinal} \} \) and \( S(P,\rho_2) = \{ J^2_i \mid i \text{ an ordinal} \} \).

Indeed: \( J^1_0 = J^2_0 = \emptyset \). Propositions 4.14 and 4.15 then ensure that, for a successor ordinal \( i \), \( J^1_i = \Pi_{P,\rho_1}(J^1_{i-1}) \leq \Pi_{P,\rho_2}(J^2_{i-1}) = J^2_i \) follows from \( \rho_1 \leq \rho_2 \) and the induction hypothesis. For a limit ordinal \( i \), \( J^1_i = \bigcup_{j<i} J^1_j \leq \bigcup_{j<i} J^2_j = J^2_i \) is immediate from the induction hypothesis. \( \square \)

Consider program \( P \) from Example 4.13 on page 80 and its interpretation \( J_2 \) from Example 4.18 on the preceding page. Note that for \( J_2 \) we obtain that

1. \( J_2(r_1) = I_M(0.8, 0.5) = 0.5 \)
2. \( J_2(r_2) = I_M(0.5, 0.3) = 0.3 \)
3. \( J_2(r_3) = I_M(T_M(0.5, 0.3), 0.3) = 1 \)
4. \( J_2(r_4) = I_M(0.2, 0.3) = 1 \)

Hence, according to Definition 4.6, \( J_2 \) is a \( \rho \)-rule model with \( \rho \) defined as in Example 4.18, i.e. \( \rho = \{ r^{0.5}_1, r^{0.3}_2, r^{1}_3, r^{1}_4 \} \). It turns out that this is a general property, i.e. that fixpoints of the immediate consequence operator are \( \rho \)-rule models. Note that this property holds for all constraint-free programs and thus in particular also for non-positive programs.

**Proposition 4.20.** Let \( P \) be a constraint-free AFASP program, \( \rho \) a rule interpretation of \( P \) and \( M \) a fixpoint of \( \Pi_{P,\rho} \). Then \( M \) is a \( \rho \)-rule model of \( P \).
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

Proof. Since $P$ is constraint-free, we know that $R_P = \bigcup_{a \in B_P} P_a$. Using this property we obtain the stated as follows:

$$M = \Pi_{P,\rho}(M)$$

$\equiv$  (Def. $\Pi_{P,\rho}(M)$)  $\forall a \in B_P \cdot M(a) = \sup\{M_s(r,\rho(r)) \mid r \in P_a\}$

$\Rightarrow$  (sup is upper bound)  $\forall a \in B_P \cdot \forall r \in P_a \cdot M(a) \geq M_s(r,\rho(r))$

$\equiv$  (Prop. 4.9, (*))  $\forall r \in R_P \cdot M(r_h) \geq \check{T}_r(M(r_b),\rho(r))$

$\equiv$  (Residuation principle)  $\forall r \in R_P \cdot I_r(M(r_b),M(r_h)) \geq \rho(r)$

$\equiv$  (Def. $\rho$-rule model)  $M$ is a $\rho$-rule model of $P$

where the (*) justification is that $R_P = \bigcup_{a \in B_P} P_a$.

The converse is not true in general, as one can see from the following example.

Example 4.21. Consider the following simple AFASP program:

$$r : a \leftarrow 0.5$$

with rule interpretation $\rho_\top$ and interpretation $I = \{a^1\}$. As $I(r) = 1 = \rho_\top(r)$, $I$ is a $\rho_\top$-rule model of $P$. But $\Pi_{P,\rho_\top}(I)(a) = \check{T}_M(0.5,1) = 0.5 < I(a)$ and thus $I$ is not a fixpoint of $\Pi_{P,\rho_\top}$.

However, the converse of Proposition 4.20 turns out to hold for minimal (w.r.t. Zadeh inclusion$^1$) $\rho$-rule models and simple programs.

Proposition 4.22. Let $P$ be a simple AFASP program, $\rho$ a rule interpretation of $P$ and $M$ a minimal $\rho$-rule model of $P$. Then $M$ is a fixpoint of $\Pi_{P,\rho}$.

Proof. Since $P$ is constraint-free, we know that $R_P = \bigcup_{a \in B_P} P_a$. Now, suppose that $M$ is a minimal $\rho$-rule model of $P$ and not a fixpoint of $\Pi_{P,\rho}$. Then we first show that $\forall a \in B_P \cdot M(a) \geq \sup_{r \in P_a} M_s(r,\rho(r))$ as follows:

$$M \text{ is a } \rho\text{-rule model of } P$$

$\equiv$  (Def. $\rho$-rule model)  $\forall r \in R_P \cdot I_r(M(r_b),M(r_h)) \geq \rho(r)$

$\equiv$  (Residuation principle)  $\forall r \in R_P \cdot M(r_h) \geq \check{T}_r(M(r_b),\rho(r))$

$\equiv$  (Prop. 4.9, $R_P = \bigcup_{a \in B_P} P_a$)  $\forall a \in B_P \cdot \forall r \in P_a \cdot M(a) \geq M_s(r,\rho(r))$

$\equiv$  (Def. upper bound, sup)  $\forall a \in B_P \cdot M(a) \geq \sup_{r \in P_a} M_s(r,\rho(r))$

$^1$See Definition 2.36 on page 41.
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

As $M$ is not a fixpoint of $\Pi_{P,\rho}$, for some $a \in B_P$ it must hold that $M(a) > \sup_{r \in P_a} M_s(r, \rho(r))$. Consider then the interpretation $M'$ defined as

$$M'(x) = \begin{cases} M(x) & \text{if } x \neq a \\ \sup_{r \in P_a} M_s(r, \rho(r)) & \text{otherwise} \end{cases}$$

Clearly $M' < M$. We will show that $M'$ is a $\rho$-rule model, leading to a contradiction with the minimality of $M$. For any $x \in (B_P \setminus \{a\})$ we show that

$$\forall r \in P_x \cdot I_r(M'(r_b), M'(x)) \geq \rho(r) \quad (4.4)$$

Indeed:

$$M' < M \Rightarrow \langle \text{Body is increasing} \rangle \quad M'(r_b) \leq M(r_b)$$

$$\Rightarrow \langle \text{Anti-monoton. } I \rangle \quad I_r(M'(r_b), M(x)) \geq I_r(M(r_b), M(x))$$

$$\Rightarrow \langle M \text{ is } \rho \text{-rule model} \rangle \quad I_r(M'(r_b), M(x)) \geq \rho(r)$$

$$\equiv \langle \text{Def. } M', \ r_b = x \neq a \rangle \quad I_r(M'(r_b), M'(x)) \geq \rho(r)$$

For $a$ we show that

$$\forall r \in P_a \cdot I_r(M'(r_b), M'(a)) \geq \rho(r) \quad (4.5)$$

as follows:

$$M'(a) = \sup_{r \in P_a} M_s(r, \rho(r))$$

$$\Rightarrow \langle \text{sup is upper bound} \rangle \quad \forall r \in P_a \cdot M'(a) \geq M_s(r, \rho(r))$$

$$\equiv \langle \text{Prop. 4.9} \rangle \quad \forall r \in P_a \cdot M'(a) \geq I_r(M(r_b), \rho(r))$$

$$\Rightarrow \langle M' < M, \text{ Monot. } \wedge_r, (\star) \rangle \quad \forall r \in P_a \cdot M'(a) \geq I_r(M'(r_b), \rho(r))$$

$$\equiv \langle \text{Residuation principle} \rangle \quad \forall r \in P_a \cdot I_r(M'(r_b), M'(a)) \geq \rho(r)$$

The (\star) justification is that $r_b$ contains an increasing function. By combining (4.4) and (4.5), we obtain that $M'$ is a $\rho$-rule model, contradicting the assumption that $M$ is a minimal $\rho$-rule model. Hence $M$ must be a fixpoint of $\Pi_{P,\rho}$.

Finally, we define $k$-answer sets of a program $P$ as those least fixpoints of the immediate consequence operator that are $k$-models of $P$.

**Definition 4.23.** Let $P$ be a positive AFASP program. An interpretation $M$ is a $k$-answer set ($k \in P_P$) of $P$ iff $M = \Pi_{P,\rho,M} \text{ and } A_P(\rho_M) \geq k$. 
Chapter 4. Aggregated Fuzzy Answer Set Programming

Example 4.24. Consider program $P$ from Example 4.13 on page 80 and the least fixpoints $\Pi_{P,\rho}$ and $\Pi_{P,\rho}$ (with $\rho$ as in Example 4.18 on page 82) computed in Example 4.17 on page 82 and Example 4.18 on page 82 respectively. For convenience we refer to $\Pi_{P,\rho}$ as $A_1$ and to $\Pi_{P,\rho}$ as $A_2$. Then $A_1$ is a 1-answer set of $P$ as $A_P(\rho_{A_1}) = \inf \{ r_1, r_2, r_3, r_4 \} (\rho_{A_1}) = 1$ and $A_2$ is an 0.3-answer set as $A_P(\rho_{A_2}) = \inf \{ r_1, r_2, r_3, r_4 \} (\rho_{A_2}) = 0.3$. Note that $A_1$ is also an 0.3-answer set, since according to Definition 4.6 it is an 0.3-model, i.e. $\rho_{A_1}(A_P) \geq 0.3$.

Similar to FASP, the idea behind this definition is that answer sets represent the knowledge inferable from a program $P$ without resorting to external knowledge, i.e. knowledge not contained in the program. This is reflected in the definition since the least fixpoint of $\Pi_{P,\rho}$ corresponds to the result of applying forward chaining on the minimal interpretation. Furthermore, the knowledge expressed by an answer set is also maximal as it is a fixpoint of the immediate consequence operator. Hence using forward chaining on this model will not yield new knowledge.

The $k$-prefix allows to distinguish between approximate answer sets, i.e. answer sets that do not fulfill the rules of the program completely. This allows us to handle conflicting information, or to find approximate solutions to problems encoded as fuzzy answer set programs, when the computation of perfect solutions is too costly.

Note that answer sets for simple programs, contrary to the classical case and non-aggregated FASP approaches, are not necessarily unique. This is illustrated in the following example.

Example 4.25. Consider a program $P_{ex4.25}$ with the following rules:

$$r_1: \quad a \leftarrow_{m} 1$$
$$r_2: \quad b \leftarrow_{m} 1$$

and aggregator function $A_{P_{ex4.25}}(\rho) = \inf \{ \rho(r) | r \in \mathcal{R}_{P_{ex4.25}} \}$. Consider now two interpretations of $P_{ex4.25}$, viz. $I_1 = \{ a^{0.5}, b^1 \}$ and $I_2 = \{ a^1, b^{0.5} \}$. It is easy to see that both of them are least fixpoints of the immediate consequence operator with their induced rule interpretations. As $\rho_{I_1}(r_1) = 0.5 = \rho_{I_2}(r_2)$ and $\rho_{I_1}(r_2) = 1 = \rho_{I_2}(r_1)$, both of them are 0.5-answer sets.

There is a strong connection between minimal $\rho$-rule models and the answer sets.
Proposition 4.26. Let $P$ be a simple AFASP program and $\rho$ a rule interpretation of $P$. Then $\Pi_{P,\rho}^*$ is the unique minimal $\rho$-rule model of $P$.

Proof. First, we show that $\Pi_{P,\rho}^*$ is a minimal $\rho$-rule model of $P$. Due to Proposition 4.20 we know that $\Pi_{P,\rho}^*$ must be a $\rho$-rule model of $P$. Suppose $M$ is a $\rho$-rule model such that $M \leq \Pi_{P,\rho}^*$. Without loss of generalization, we can assume that $M$ is a minimal $\rho$-rule model. Now from Proposition 4.22 we know that $M$ must be a fixpoint of $\Pi_{P,\rho}$ and hence, as $\Pi_{P,\rho}^*$ is the least fixpoint of $\Pi_{P,\rho}$, that $M = \Pi_{P,\rho}^*$. Thus $\Pi_{P,\rho}^*$ is a minimal $\rho$-rule model of $P$.

Second, we show that no other minimal $\rho$-rule models of $P$ exist. Suppose $M$ is a minimal $\rho$-rule model of $P$. From Proposition 4.22 we know that $M$ must be a fixpoint of $\Pi_{P,\rho}$ and hence $\Pi_{P,\rho}^* \leq M$ as $\Pi_{P,\rho}^*$ is the least fixpoint of $\Pi_{P,\rho}$. From this it follows that $M = \Pi_{P,\rho}^*$.

From this proposition we can show that our answer sets correspond to minimal rule models.

Corollary 4.27. Let $P$ be a simple AFASP program. Then $A$ is a $A_P(\rho_A)$-answer set of $P$ iff $A$ is the unique minimal $\rho_A$-rule model of $P$.


One may wonder whether every rule interpretation $\rho$ for a simple AFASP program $P$ can be used to generate an answer set $M = \Pi_{P,\rho}^*$ such that $\rho_M = \rho$. The answer is negative, as one can see from the following example:

Example 4.28. Consider the program $P$ with aggregator $A_P(\rho) = \inf\{\rho(r) \mid r \in \mathcal{R}_P\}$ and the following rule base $\mathcal{R}_P$:

$$
\begin{align*}
& r_1 : a \leftarrow m 0.2 \\
& r_2 : a \leftarrow m gt(a, 0)
\end{align*}
$$

where $gt(x, y) = 1$ if $x > y$ and $gt(x, y) = 0$ otherwise. Computing $\Pi_{P,\rho}^*$ for $\rho = \{r_1^{0.2}, r_2\}$ yields $\Pi_{P,\rho}^* = \{a^1\} = M$, which induces $\rho_M = \{r_1^1, r_2^1\} \neq \rho$. One can easily verify that $M = \Pi_{P,\rho_M}^*$ and $A_P(\rho_M) \geq 1$, thus $M$ is an 1-answer set of $P$. 


CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

As one can see, the least fixpoint of $\Pi_{P,\rho}$ from Example 4.28 turned out to be a 1-answer set of $P$, where $1 > A_P(\rho)$, since $A_P(\rho) = 0.2$. The following propositions show that in general the least fixpoint of $\Pi_{P,\rho}$ for some program $P$ and an arbitrary rule interpretation $\rho$ of this program will always be a $k$-answer set, for all $k \geq A_P(\rho)$.

This means that to obtain a $k$-answer set of a positive program $P$ for an arbitrary $k \in P_P$, we only need to compute $\Pi_{P,\rho}^*$ for an arbitrary rule interpretation $\rho$ satisfying $A_P(\rho) \geq k$.

Lemma 4.29. Let $P$ be an AFASP program and $I$ an interpretation of this program, then for each $a \in B_P$ it holds that $I(a)$ is an upper bound of the set $\{I_s(r, \rho_I(r)) \mid r \in P_a\}$.

Proof. We show that for any $a \in B_P$ and $r \in P_a$ it holds that $I(a) \geq I_s(r, \rho_I(r))$, from which the stated readily follows.

\[
I_s(r, \rho_I(r)) = \langle \text{Def. } I_s(r, \rho_I(r)) \rangle = \inf\{y \in L_P \mid I_r(I(r_b), y) \geq \rho_I(r)\} \leq \langle \text{Def. } \rho_I \rangle = \inf\{y \in L_P \mid I_r(I(r_b), y) \geq I_r(I(r_b), I(a))\} \leq \langle \text{Def. } \inf \rangle = I(a)
\]

\[\square\]

Proposition 4.30. Let $P$ be a simple AFASP program, $\rho$ a rule interpretation of $P$, and $M = \Pi_{P,\rho}^*$. Then $\Pi_{P,\rho_M} = M$.

Proof. First, note that from Proposition 4.20 we can immediately see that $\rho \leq \rho_M$ as $M$ is a fixpoint of $\Pi_{P,\rho}$. Second, we show that $\Pi_{P,\rho_M}(M) = M$:

\[
M = \Pi_{P,\rho}^* \\
\Rightarrow \langle \text{Def. fixpoint} \rangle \quad \forall a \in B_P \cdot M(a) = \Pi_{P,\rho}(M)(a) \\
\Rightarrow \langle \rho \leq \rho_M, \text{Prop. 4.15} \rangle \quad \forall a \in B_P \cdot M(a) \leq \Pi_{P,\rho_M}(M)(a) \\
\equiv \langle \text{Def. } \Pi_{P,\rho_M} \rangle \quad \forall a \in B_P \cdot M(a) \leq \sup_{r \in P_a} M_s(r, \rho_M(r)) \\
\Rightarrow \langle \text{Lemma 4.29} \rangle \quad \forall a \in B_P \cdot M(a) = \sup_{r \in P_a} M_s(r, \rho_M(r)) \\
\equiv \langle \text{Def. } \Pi_{P,\rho_M} \rangle \quad \forall a \in B_P \cdot M(a) = \Pi_{P,\rho_M}(M)(a)
\]

By Proposition 4.19 and the fact that $\rho \leq \rho_M$ we obtain that $\Pi_{P,\rho}^* \leq \Pi_{P,\rho_M}^*$ and thus by definition of $M$ that $M \leq \Pi_{P,\rho_M}^*$. As we have shown that $M$ is a fixpoint of $\Pi_{P,\rho_M}$, and thus $\Pi_{P,\rho_M}^* \leq M$, this means that $M = \Pi_{P,\rho_M}^*$. \[\square\]
Proposition 4.31. Let \( P \) be a simple AFASP program and \( \rho \) a rule interpretation of \( P \). Then \( M = \Pi_{P,\rho}^\ast \) is a \( A_P(\rho) \)-answer set.

Proof. Due to Proposition 4.30 we already know that \( M = \Pi_{P,\rho_M}^\ast \). We thus only need to show that \( A_P(\rho_M) \geq A_P(\rho) \). From Proposition 4.20, we know that \( M \) is a \( \rho \)-rule model, i.e. \( \forall r \in R_P \cdot \rho_M(r) \geq \rho(r) \). This implies \( A_P(\rho_M) \geq A_P(\rho) \) because \( A_P \) is increasing. \( \square \)

We obtain two immediate corollaries, the first of which shows that a model of a simple program is a \( k \)-answer set iff it is produced by some \( \rho \)-rule interpretation with \( A_P(\rho) \geq k \).

Corollary 4.32. \( M \) is a \( k \)-answer set of a simple AFASP program \( P \) iff there is some rule interpretation \( \rho \) for which \( A_P(\rho) \geq k \), such that \( M = \Pi_{P,\rho}^\ast \).

Proof. First, suppose \( M \) is a \( k \)-answer set of a simple AFASP program \( P \). By definition of \( k \)-answer sets it must then hold that \( A_P(\rho_M) \geq k \) and that \( M = \Pi_{P,\rho_M}^\ast \), hence some rule interpretation \( \rho \) exists such that \( M = \Pi_{P,\rho}^\ast \) and \( A_P(\rho_M) \geq k \).

Second, suppose there is some rule interpretation \( \rho \) for which \( A_P(\rho) \geq k \) and \( M = \Pi_{P,\rho}^\ast \). From Proposition 4.31 we then know that \( M \) is a \( A_P(\rho) \)-answer set of \( P \), from which we know that \( A_P(\rho_M) \geq A_P(\rho) \) and hence, as \( A_P(\rho) \geq k \), it follows that \( A_P(\rho_M) \geq k \). Thus \( M \) is a \( k \)-answer set. \( \square \)

The following corollary shows that it is easy to obtain a suitable rule interpretation for simple AFASP programs.

Corollary 4.33. Every simple AFASP program \( P \) has a \( \text{max}(P) \)-answer set, with \( \text{max}(P) \) as defined in Definition 4.6.

Proof. Let \( P \) be a simple AFASP program. The desired answer set is obtained by applying Proposition 4.30 to the rule interpretation \( \rho_\top \). \( \square \)

Hence, when constructing a \( k \)-answer set of a simple program \( P \), with \( k \in \mathcal{P}_P \) and \( k \leq \text{max}(P) \), we can simply use \( \rho_\top \) and compute \( \Pi_{P,\rho_\top}^\ast \). As any \( k_1 \)-answer set of \( P \) is a \( k_2 \)-answer set of \( P \) for \( k_1 \geq k_2 \), \( \Pi_{P,\rho_\top}^\ast \) is a \( k \)-answer set for any \( k \leq \text{max}(P) \).
Example 4.34. Consider an AFASP program $P$ with rule base $\mathcal{R}_P$:

\[
\begin{align*}
    r_1 : a & \leftarrow m 0.8 \\
    r_2 : b & \leftarrow m 0.4 \\
    r_3 : c & \leftarrow \mathcal{T}_M(a, b)
\end{align*}
\]

The aggregator is $A_P(\rho) = \rho(r_1) + \rho(r_2) + \rho(r_3)$, defined over the preorder $(\mathcal{R}, \leq)$. As $A_P(\rho_{\top}) = 3$, we know that $\Pi_{\mathcal{R}_P, \rho_{\top}}^* = \{ a^{0.8}, b^{0.4}, c^{0.4} \}$ is a 3-answer set of $P$. Since the least fixpoint of $\Pi_{\mathcal{R}_P, \rho_{\top}}$ is unique we know from the aggregator definition that this is also the unique 3-answer set of $P$.

General programs

Recall that for a FASP program $\mathcal{R}_P$ and interpretation $M$ we defined the reduct of $\mathcal{R}_P$ w.r.t. $M$, denoted as $\mathcal{R}_P^M$, in Definition 3.14 on page 59. The semantics of general AFASP programs will be based on this reduct definition. However, different to FASP we don't require answer sets to be models, but require them to be $k$-models. Hence they are approximate answer sets.

Definition 4.35. Let $P$ be an AFASP program. An interpretation $M$ is a $k$-answer set of $P$ ($k \in \mathcal{P}_P$) iff $M = \Pi_{\mathcal{R}_P^M, \rho_M}^*$ and $A_P(\rho_M) \geq k$.

Example 4.36. Consider program $P_{fgc}$ from Section 3.3 on page 64 together with interpretations $I_1$ and $I_2$ from Example 4.7 on page 76. It turns out that both $I_1$ and $I_2$ are approximate answer sets of this program. Indeed, for $I_1$ the reduct $\mathcal{R}_{P_{fgc}}^{I_1}$ is:

\[
\begin{align*}
    (\text{gen}_1)_a : \text{white}(a) & \leftarrow 1 \\
    (\text{gen}_1)_b : \text{white}(b) & \leftarrow 0.7 \\
    (\text{gen}_1)_c : \text{white}(c) & \leftarrow 0.3 \\
    (\text{gen}_2)_a : \text{black}(a) & \leftarrow 0 \\
    (\text{gen}_2)_b : \text{black}(b) & \leftarrow 0.3 \\
    (\text{gen}_2)_c : \text{black}(c) & \leftarrow 0.7
\end{align*}
\]
4.2. AGGREGATED FUZZY ANSWER SET PROGRAMMING

\[
\begin{align*}
(sim_1)_{(a,a)} &: \text{sim}(a,a) \leftarrow T_M(\mathcal{I}_M(1, \text{white}(a)), \mathcal{I}_M(1, \text{white}(a))) \\
(sim_1)_{(a,b)} &: \text{sim}(a,b) \leftarrow T_M(\mathcal{I}_M(0.7, \text{white}(a)), \mathcal{I}_M(1, \text{white}(b))) \\
(sim_1)_{(a,c)} &: \text{sim}(a,c) \leftarrow T_M(\mathcal{I}_M(0.3, \text{white}(a)), \mathcal{I}_M(1, \text{white}(c))) \\
(sim_1)_{(b,a)} &: \text{sim}(b,a) \leftarrow T_M(\mathcal{I}_M(1, \text{white}(b)), \mathcal{I}_M(0.7, \text{white}(a))) \\
(sim_1)_{(b,b)} &: \text{sim}(b,b) \leftarrow T_M(\mathcal{I}_M(0.7, \text{white}(b)), \mathcal{I}_M(0.7, \text{white}(b))) \\
(sim_1)_{(b,c)} &: \text{sim}(b,c) \leftarrow T_M(\mathcal{I}_M(0.3, \text{white}(b)), \mathcal{I}_M(0.7, \text{white}(c))) \\
(sim_1)_{(c,a)} &: \text{sim}(c,a) \leftarrow T_M(\mathcal{I}_M(1, \text{white}(c)), \mathcal{I}_M(0.3, \text{white}(a))) \\
(sim_1)_{(c,b)} &: \text{sim}(c,b) \leftarrow T_M(\mathcal{I}_M(0.7, \text{white}(c)), \mathcal{I}_M(0.3, \text{white}(b))) \\
(sim_1)_{(c,c)} &: \text{sim}(c,c) \leftarrow T_M(\mathcal{I}_M(0.3, \text{white}(c)), \mathcal{I}_M(0.3, \text{white}(c))) \\
(sim_2)_{(a,a)} &: \text{sim}(a,a) \leftarrow T_M(\mathcal{I}_M(0, \text{black}(a)), \mathcal{I}_M(0, \text{black}(a))) \\
(sim_2)_{(a,b)} &: \text{sim}(a,b) \leftarrow T_M(\mathcal{I}_M(0.3, \text{black}(a)), \mathcal{I}_M(0, \text{black}(b))) \\
(sim_2)_{(a,c)} &: \text{sim}(a,c) \leftarrow T_M(\mathcal{I}_M(0.7, \text{black}(a)), \mathcal{I}_M(0, \text{black}(c))) \\
(sim_2)_{(b,a)} &: \text{sim}(b,a) \leftarrow T_M(\mathcal{I}_M(0, \text{black}(b)), \mathcal{I}_M(0.3, \text{black}(a))) \\
(sim_2)_{(b,b)} &: \text{sim}(b,b) \leftarrow T_M(\mathcal{I}_M(0.3, \text{black}(b)), \mathcal{I}_M(0.3, \text{black}(b))) \\
(sim_2)_{(b,c)} &: \text{sim}(b,c) \leftarrow T_M(\mathcal{I}_M(0.7, \text{black}(b)), \mathcal{I}_M(0.3, \text{black}(c))) \\
(sim_2)_{(c,a)} &: \text{sim}(c,a) \leftarrow T_M(\mathcal{I}_M(0, \text{black}(c)), \mathcal{I}_M(0.7, \text{black}(a))) \\
(sim_2)_{(c,b)} &: \text{sim}(c,b) \leftarrow T_M(\mathcal{I}_M(0.3, \text{black}(c)), \mathcal{I}_M(0.7, \text{black}(b))) \\
(sim_2)_{(c,c)} &: \text{sim}(c,c) \leftarrow T_M(\mathcal{I}_M(0.7, \text{black}(c)), \mathcal{I}_M(0.7, \text{black}(c))) \\
\text{constr}_{(a,a)} &: 0 \leftarrow T_W(\text{edge}(a,a), \text{sim}(a,a)) \\
\text{constr}_{(a,b)} &: 0 \leftarrow T_W(\text{edge}(a,b), \text{sim}(a,b)) \\
\text{constr}_{(a,c)} &: 0 \leftarrow T_W(\text{edge}(a,c), \text{sim}(a,c)) \\
\text{constr}_{(b,a)} &: 0 \leftarrow T_W(\text{edge}(b,a), \text{sim}(b,a)) \\
\text{constr}_{(b,b)} &: 0 \leftarrow T_W(\text{edge}(b,b), \text{sim}(b,b)) \\
\text{constr}_{(b,c)} &: 0 \leftarrow T_W(\text{edge}(b,c), \text{sim}(b,c)) \\
\text{constr}_{(c,a)} &: 0 \leftarrow T_W(\text{edge}(c,a), \text{sim}(c,a)) \\
\text{constr}_{(c,b)} &: 0 \leftarrow T_W(\text{edge}(c,b), \text{sim}(c,b)) \\
\end{align*}
\]
One can easily verify that $\Pi_{\mathcal{R}_{P_{fgc}}^g}^{\rho_{I_1}} I_1 = I_1$. Hence, combining this with the observation that $I_1$ is an 1.92-model of $P_{fgc}$ using aggregator $A$ from Example 4.7, we obtain that $I_1$ is a 1.92-answer set of $P$. With aggregator $A'$ from Example 4.7 it would be a $(0.7, 0.7, 0.3)$-answer set. Likewise we obtain that $I_2 = \Pi_{\mathcal{R}_{P_{fgc}}^g}^{\rho_{I_2}} I_2$. This means that $I_2$ is a 2-answer set with aggregator $A$ and a $(0.5, 0.5, 1)$-answer set with aggregator $A'$.

Intuitively, answer sets need to be self-producing, i.e. by assuming the knowledge in the answer set and starting from the empty interpretation of a program, we should only be able to infer the same set. Note that Definition 4.35 supports programs with constraints, as these only influence the $k$-score obtained in the aggregator, and not the fixpoint computation of the reduct program; therefore constraints can only restrict the results and cannot add atoms to solutions. A first proposition shows that answer sets are models of a program, as we would expect.

**Proposition 4.37.** Let $M$ be a $k$-answer set of an AFASP program $P$. Then $M$ is a $k$-model of $P$.

**Proof.** From Definition 4.35, it immediately follows that $A_P(\rho_M) \geq k$. □

A second proposition shows that answer sets are minimal rule models of a (constraint-free) program.

**Proposition 4.38.** Let $M$ be a $k$-answer set of a constraint-free AFASP program $P$. Then $M$ is a minimal $\rho_M$-model of $P$.

**Proof.** Let $M$ be a $k$-answer set of $P$, then by definition of answer sets we know that $M = \Pi_{\mathcal{R}_{P_{M}}^g}^{\rho_{M}}$. We will show by contradiction that no $M' \subset M$ can exist such that $M'$ is a $\rho_M$-rule model of $P$. Suppose $M' \subset M$ such that $M'$ is a $\rho_M$-rule model of $P$. From this it follows that for any rule $r \in P$ we have $M'(r_b) \geq M'(r_b^M)$ as only the arguments in which the body function is decreasing are replaced by their...
value in $M$. We can then proceed as follows:

\[
\begin{align*}
\equiv & \quad \langle \text{Def. } \rho_M\text{-rule model} \rangle \quad \forall r \in R \cdot I_r(M'(r_b), M'(r_h)) \geq \rho_M(r) \\
\equiv & \quad \langle \text{Property reduct} \rangle \quad \forall r \in R \cdot I_r(M'(r_b^M), M'(r_h^M)) \geq \rho_M(r^M) \\
\Rightarrow & \quad \langle M'(r_b) \geq M'(r_b^M) \rangle \quad \forall r \in R \cdot I_r(M'(r_b^M), M'(r_h^M)) \geq \rho_M(r^M) \\
\equiv & \quad \langle r_h \in B_P, \text{ Def. reduct} \rangle \quad \forall r \in R \cdot I_r(M'(r_b^M), M'(r_h^M)) \geq \rho_M(r) \\
\equiv & \quad \langle \text{Def. } \mathcal{R}_P^M \rangle \quad \forall r \in R \cdot I_r(M'(r_b^M), M'(r_h^M)) \geq \rho_M(r) \\
\end{align*}
\]

$M'$ is a $\rho_M$-rule model of $P$.

From Proposition 4.26 and the fact that $M = \Pi_{R_P, \rho_M}$, we know that $M$ is the unique minimal $\rho_M$-rule model of $\mathcal{R}_P^M$, leading to a contradiction with the fact that $M'$ is a $\rho_M$-rule model of $\mathcal{R}_P^M$ and $M' \subset M$.

As is the case for classical answer set programming and FASP, the reverse of this proposition does not hold:

**Example 4.39.** Consider the $\rho_\top$-rule model and interpretation $M = \{a^0, b^1\}$ of the following program $P$:

\[
\begin{align*}
r : a & \leftarrow_m N_W(b)
\end{align*}
\]

with aggregator $A_P(\rho) = \inf\{\rho(r) \mid r \in P\}$. As any $M' \subset M$ must satisfy $M'(a) = 0$ and $M'(b) < 1$, we obtain that $M'(r) = I_M(1 - M'(b), M'(a)) = I_M(1 - M'(b), 0)$. Since $I_M(1 - M'(b), 0) \geq 1$ only if $1 - M'(b) = 0$, or $M'(b) = 1$, we obtain $M'(r) < 1$. This means that $M$ is a minimal $\rho_\top$-rule model of $P$. $M$ is not a 1-answer set of $P$, however, as $\Pi_{\rho_M, \rho_\top} = \emptyset \neq M$.

Similar to Proposition 3.21 for FASP on page 62, there is a correspondence with minimal fixpoints of the immediate consequence operator and answer sets of (constraint-free) AFASP programs.

**Lemma 4.40.** Let $P$ be an AFASP program, then an interpretation $I$ is a fixpoint of $\Pi_{\rho, \rho}$ iff it is a fixpoint of $\Pi_{\mathcal{R}_P, \rho}$.

**Proof.** First, remark that for any expression $\alpha$ and interpretation $I$ we have that
I(\alpha) = I(\alpha^I). Then we proceed as follows:

\[ I = \Pi_{p, \rho}(I) \]
\[ \equiv \langle \text{Equality of functions} \rangle \quad \forall a \in B_p \cdot I(a) = \Pi_{p, \rho}(I)(a) \]
\[ \equiv \langle \text{Def. } \Pi_{p, \rho} \rangle \quad \forall a \in B_p \cdot I(a) = \sup_{r \in P_a} I_s(r, \rho(r)) \]
\[ \equiv \langle \text{Prop. 4.9} \rangle \quad \forall a \in B_p \cdot I(a) = \sup_{r \in P_a} T_r(I(r_b), \rho(r)) \]
\[ \equiv \langle I(\alpha) = I(\alpha^I) \rangle \quad \forall a \in B_p \cdot I(a) = \sup_{r \in P_a} T_r(I(r_b^I), \rho(r)) \]
\[ \equiv \langle \text{Def. } \Pi_{p, \rho}, \text{ Def. } R_{p, \rho}^I \rangle \quad \forall a \in B_p \cdot I(a) = \Pi_{R_{p, \rho}^I}(I)(a) \]

Hence I is a fixpoint of \( \Pi_{p, \rho} \).

□

**Proposition 4.41.** Let P be a constraint-free AFASP program with a k-answer set \( M \). Then \( M \) is a minimal fixpoint of \( \Pi_{p, \rho, M} \).

**Proof.** From Lemma 4.40, we know that any answer set \( M \) must be a fixpoint of \( \Pi_{p, \rho, M} \). Suppose \( M \) is not a minimal fixpoint and thus, that some \( N \) exists, \( N \subset M \), such that \( N \) is a fixpoint of \( \Pi_{p, \rho, M} \). We will show that any such \( N \) is also a \( \rho_M \)-rule model of \( R_{p, \rho}^M \), leading to \( N \supseteq M \) due to Proposition 4.26, a contradiction.

First, as \( N \subset M \) and since the reduct only substitutes negative subexpressions for their corresponding values, it follows that, for any expression \( \alpha \):

\[ N(\alpha^N) \geq N(\alpha^M) \]

and thus, since \( N(\alpha) = N(\alpha^N) \) that

\[ N(\alpha) \geq N(\alpha^M) \]

We can now show that \( N \) is a \( \rho_M \)-rule model of \( R_{p, \rho}^M \) as follows:

\[ N = \Pi_{p, \rho, M}(N) \]
\[ \Rightarrow \langle \text{Prop. 4.20} \rangle \quad \forall r \in R_p \cdot \mathcal{I}_r(N(r_b), N(r_h)) \geq \rho_M(r) \]
\[ \Rightarrow \langle \text{(*)} \rangle \quad \forall r \in R_p \cdot \mathcal{I}_r(N(r_b^M), N(r_h)) \geq \rho_M(r) \]
\[ \equiv \langle \text{Def. reduct} \rangle \quad \forall r \in R_{p, \rho}^M \cdot \mathcal{I}_r(N(r_b), N(r_h)) \geq \rho_M(r) \]
\[ \equiv \langle \text{Def. } N(r) \rangle \quad \forall r \in R_{p, \rho}^M \cdot N(r) \geq \rho_M(r) \]

where \( (*) \) follows from the anti-monotonicity of \( \mathcal{I} \) in its second argument and the fact that \( N(r_b) \geq N(r_b^M) \).

□

The converse of Proposition 4.41 does not hold, however, as witnessed by the following example.
Example 4.42. Consider program $P_{ex4.42}$:

$$
\begin{align*}
  r_1 & : a \leftarrow a \\
  r_2 & : p \leftarrow T_M(gt(N_W(p), 0), gt(N_W(a), 0))
\end{align*}
$$

and the following aggregator over $([0, 1], \leq)$: $A_P = T_M(p(r_1), p(r_2))$. Note that $gt(x, y)$ is defined as in Example 4.28. It is easy to verify that $M = \{a^1, p^0\}$, with $\rho_M = \rho^\top$, is the only, and thus minimal, fixpoint of $\Pi_{P_{ex4.42}, \rho^\top}$. However, $R^M_{P_{ex4.42}}$ is

$$
\begin{align*}
  r^1_M & : a \leftarrow a \\
  r^2_M & : p \leftarrow T_M(1, gt(0, 0))
\end{align*}
$$

It thus holds that $\Pi^{*\rho_M}_{R^M_{P_{ex4.42}}, \rho^\top} = \{a^0, p^0\} \neq M$. From this it follows that the minimal fixpoint of $\Pi_{P, \rho^\top}$ is not an answer set of $P_{ex4.42}$.

Note that $P_{ex4.42}$ is very similar to $P_{ex2.32}$ from Example 2.32 on page 38, which showed that answer sets do not correspond to minimal models of the completion. The above example illustrates our intuition about answer sets: the minimal interpretation $M$ contains an atom $a$ that is self-motivating and therefore unwanted. Hence, not every minimal fixpoint is intuitively suitable as an answer set.

Finally, we would like to point out that constraints can be simulated in the presented framework using decreasing functions, meaning the aforementioned results are generally applicable. The details of this simulation will be discussed at length in Chapter 5.

4.3 Illustrative Example

In this section we illustrate how the features of the AFASP framework can be useful for building real-life applications. The example we use is the “paper distribution” problem, which attracted quite some attention from the research community (see e.g., [63]). Specifically, we assume that there is a set of papers (named Papers) about a certain set of topics (named Topics) that need to be assigned to reviewers (from the set Reviewers) with a certain expertise on the aforementioned topics. When assigning these papers, care must be taken to ensure that there are no conflicts between reviewers and authors; furthermore, each paper should have enough reviewers
and no reviewer should be burdened with a high review workload. We assume that
the expertise of reviewers, the topics of papers and the affiliations of both authors
and reviewers are known and thus need not be calculated with an AFASP program,
but are given by a set of fact rules \( \mathcal{F} \). For example to denote that reviewer \( r_1 \)
is an expert of degree 0.4 on topic \( t_3 \) we add the fact \( f_{r_1,t_3} : \text{expert}(r_1, t_3) \leftarrow 0.4 \).
The rule base \( \mathcal{R}_{\text{paper}} \) of the program \( P_{\text{paper}} \) solving this problem is defined over the
lattice \( ([0, 1], \leq) \) and consists of the set \( \mathcal{F} \) together with the following rules:

\[
\begin{align*}
\text{confl} : & \quad \text{conflict}(R, P) \leftarrow_m S_M(\text{author}(R, P), \\
& \quad T_M(\text{author}(R', P), \text{university}(R, U), \\
& \quad \text{university}(R', U'), \text{close}(U, U'))) \\
\text{appr} : & \quad \text{appropriate}(R, P) \leftarrow_m T_M(\text{Nw}(\text{appropriate}(R, P)), \\
& \quad T_W(\text{about}(P, T), \text{expert}(R, T))) \\
\text{inappr} : & \quad \text{inappropriate}(R, P) \leftarrow_l \text{Nw}(\text{appropriate}(R, P)) \\
\text{qualr} : & \quad \text{overworked}(R) \leftarrow_l f_1(\sum_{p \in \text{Papers}} \text{assign}(R, p)) \\
\text{enough} : & \quad \text{enough}(P) \leftarrow_m f_2(\sum_{r \in \text{Reviewers}} \text{assign}(r, P)) \\
\text{qualp} : & \quad 0 \leftarrow_l \text{Nw}(\text{enough}(P)) \\
\text{assign} : & \quad \text{assign}(R, P) \leftarrow_m \text{geq}(T_M(\text{Nw}(\text{inappropriate}(R, P)), \\
& \quad \text{Nw}(\text{overworked}(R))), 1)
\end{align*}
\]

where \( \text{geq}(x, y) = 1 \) if \( x \geq y \) and \( \text{geq}(x, y) = 0 \) otherwise; furthermore \( f_1 \) is:

\[
f_1(x) = \begin{cases} 
0 & \text{if } x \leq 3 \\
\frac{x - 3}{7} & \text{if } x \in [4, 9] \\
1 & \text{if } x \geq 10 
\end{cases}
\]

and \( f_2 \) is:

\[
f_2(x) = \begin{cases} 
\frac{x}{4} & \text{if } x \leq 3 \\
1 & \text{otherwise}
\end{cases}
\]

Note that the application of t-norms to more than one argument, such as in rule
\text{confl}, poses no problem due to their associativity. Furthermore we recall that, due
to the process of grounding (see e.g. [5]), a rule like \text{inappr} actually denotes the set
of rules \( \{ \text{inappr}_{r, p} : \text{inappropriate}(r, p) \leftarrow_l (1 - \text{appropriate}(r, p)) \mid r \in \text{Reviewers}, \\
p \in \text{Papers} \} \).
The intuition of the rules is as follows. The confl rules determine when there are potential conflicts-of-interest with reviewer neutrality, i.e. \( \text{conflict}(R, P) \) quantifies the degree of conflict that diminishes the suitability of person \( R \) reviewing paper \( P \). To keep the discussion simple, we opted to only consider the degree to which universities are close (by which we mean both geographical proximity and the affiliations between universities) to determine these conflicts. Furthermore we assumed that the set of authors is equivalent to the set of potential reviewers. The appr rules determine the degree to which an assignment is appropriate, based on the expertise of the reviewer (\( \text{expert}(R, T) \)), the topic of the paper (\( \text{about}(P, T) \)) and potential conflicts. We use the Łukasiewicz t-norm for combining the reviewer knowledge and paper topic to ensure that reviewers have enough knowledge about the paper content. The inappr rules determine the inappropriateness of a given paper assignment. The qualr rules determine when a reviewer is overworked. By our definition of \( f_1 \) the degree to which a reviewer is overworked scales linearly with the number of assigned papers, where less than three papers means that he is not overworked at all and ten or more papers means that he has too many papers to review. The combination of the enough rules and qualp constraints are used in the aggregator to score an answer set based on the number of reviews papers have. Due to our definition of \( f_2 \) the degree to which a paper is considered to have enough reviewers scales linearly, where 4 or more reviewers is considered to be optimal. Last, the assgn rules assign papers to reviewers based on the suitability of the match between reviewer and paper and bearing in mind the workload of reviewers.

Note that the inappr, qualr, and qualp rules are all evaluated with a Łukasiewicz implicator. The reason for this is that the former rules are allowed to be partially fulfilled and we want their fulfillment to change gradually, while the other rules should be completely fulfilled, hence an arbitrary residual implicator can be used for their evaluation. The effect of allowing the inappr to be partially satisfied is that we can underestimate the inappropriateness score when this leads to better reviewer assignments. For example, if a certain paper has a low number of reviewers, we can opt to also include a reviewer that is less familiar with the topics of the paper in this way. Furthermore, note that there is a strong interaction between the assgn, inappr, and qualr rules: if no reviewer is assigned a paper, the assgn rule is triggered for that reviewer and a paper is correspondingly assigned to him; this in turn leads to an increasing overworked score, leading to either fewer remaining assignments or a violation of the overworked constraint if this is needed to ensure that each paper has enough reviews for example.
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

One example of an aggregator function for $P_{\text{paper}}$ is

$$A_{P_{\text{paper}}} (\rho) = E_1(\rho) \cdot E_2(\rho) \cdot (E_3(\rho) + 10 \cdot E_4(\rho) + 20 \cdot E_5(\rho)) \quad (4.6)$$

where

$$E_1(\rho) = \min \{ \rho(f) \mid f \in F \} \geq 1$$

$$E_2(\rho) = \min \{ \min \{ \rho(\text{confl}_{r,p}), \rho(\text{appr}_{r,p}), \rho(\text{enough}_p), \rho(\text{assgn}_{r,p}) \} \mid r \in \text{Reviewers}, p \in \text{Papers} \} \geq 1$$

$$E_3(\rho) = \sum \{ \rho(\text{inappr}_{r,p}) \mid r \in \text{Reviewers}, p \in \text{Papers} \}$$

$$E_4(\rho) = \sum \{ \rho(\text{qualp}_p) \mid p \in \text{Papers} \}$$

$$E_5(\rho) = \sum \{ \rho(\text{qualr}_r) \mid r \in \text{Reviewers} \}$$

The preorder for the aggregator is $(\mathbb{R}, \leq)$. The aggregation expression ensures that the confl, appr, enough, and assgn rules are completely fulfilled and allows partial fulfillment of the inappr, qualr, and qualp rules. The weights in the expression state that solutions in which reviewers are not overworked are better than solutions in which some reviewer is assigned a paper about a topic the reviewer is not familiar with or where papers do not have a lot of reviews. This can be seen from the fact that the qualr rules are satisfied to a lower degree when we underestimate overworked$(R)$, i.e. when we attach a lower value to overworked$(R)$ than the one warranted, when this allows us to find a solution in which papers have an appropriate number of reviewers. Thus the qualr rules are less fulfilled when reviewers are actually more overworked, meaning that giving the highest weight to these rules in the aggregator makes it more important to estimate the overworked$(R)$ values correctly.

As an example, suppose $\text{Papers} = \{ p_1, \ldots, p_{10} \}$, $\text{Authors} = \{ a_1, \ldots, a_{10} \}$ such that $\text{author}(a_i, p_i) = 1$ for any $i \in 1 \ldots 10$ and $\text{Universities} = \{ u_1, \ldots, u_{10} \}$ such that $\text{university}(a_i, u_i) = 1$ for any $i \in 1 \ldots 10$. Furthermore, suppose $\text{Reviewers} = \{ r_1, \ldots, r_5 \}$, $\text{Universities}' = \{ u'_1, \ldots, u'_5 \}$ such that $\text{university}(r_i, u'_i) = 1$ for any $i \in 1 \ldots 5$. Tables 4.1 to 4.3 on the facing page then respectively give the about, expert and close scores. Note that for any answer set of this program, the value of these atoms must be those from the aforementioned tables as they are added to the program as fact rules that must be completely satisfied, i.e. things that must be true in any answer set. In Tables 4.4 to 4.5 on page 100 the corresponding conflict and appropriateness scores are shown. Note that the degree of conflict $\text{confl}(r_i, p_j)$ of reviewer $r_i$ and paper $p_j$ is equivalent to the degree of closeness $\text{close}(u'_i, u_i)$ of
4.3. ILLUSTRATIVE EXAMPLE

<table>
<thead>
<tr>
<th>$about(P, T)$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>0.8</td>
<td>1</td>
<td>0.2</td>
<td>0.4</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$p_2$</td>
<td>0</td>
<td>0.2</td>
<td>0.7</td>
<td>0.5</td>
<td>0.9</td>
<td>0.4</td>
<td>0.3</td>
</tr>
<tr>
<td>$p_3$</td>
<td>1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0</td>
</tr>
<tr>
<td>$p_4$</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.2</td>
<td>0.4</td>
<td>0.8</td>
</tr>
<tr>
<td>$p_5$</td>
<td>0.3</td>
<td>0.9</td>
<td>0.8</td>
<td>0.9</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>$p_6$</td>
<td>0.6</td>
<td>0.8</td>
<td>0.3</td>
<td>0.7</td>
<td>0.8</td>
<td>0.3</td>
<td>0</td>
</tr>
<tr>
<td>$p_7$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>$p_8$</td>
<td>0.9</td>
<td>0.2</td>
<td>0.1</td>
<td>0.7</td>
<td>0.1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$p_9$</td>
<td>0.1</td>
<td>0.7</td>
<td>0.1</td>
<td>0.2</td>
<td>0.1</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>0.7</td>
<td>0.6</td>
<td>0.6</td>
<td>0.3</td>
<td>0.2</td>
<td>1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 4.1: $about(P, T)$ scores

<table>
<thead>
<tr>
<th>expert($R, T$)</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>0.8</td>
<td>0.5</td>
<td>0.2</td>
<td>0.5</td>
<td>0.9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$r_2$</td>
<td>0.2</td>
<td>0.9</td>
<td>1</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$r_3$</td>
<td>0.5</td>
<td>0.2</td>
<td>0.8</td>
<td>0.7</td>
<td>0.4</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>$r_4$</td>
<td>0</td>
<td>0.3</td>
<td>0.2</td>
<td>0.4</td>
<td>0.5</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>$r_5$</td>
<td>1</td>
<td>0.7</td>
<td>0.8</td>
<td>0.6</td>
<td>0.9</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 4.2: expert($R, T$) scores

<table>
<thead>
<tr>
<th>close($U, U'$)</th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
<th>$u_5$</th>
<th>$u_6$</th>
<th>$u_7$</th>
<th>$u_8$</th>
<th>$u_9$</th>
<th>$u_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>0.5</td>
<td>0.3</td>
<td>0.4</td>
<td>0.8</td>
<td>0.1</td>
<td>1</td>
<td>0.4</td>
<td>0.3</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0.5</td>
<td>0.8</td>
<td>0.5</td>
<td>0.4</td>
<td>0.4</td>
<td>0.5</td>
<td>0.1</td>
<td>0</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.3</td>
<td>0.6</td>
<td>0.3</td>
<td>0.9</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>$u_4$</td>
<td>1</td>
<td>0.9</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
<td>1</td>
<td>0.5</td>
<td>0.2</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td>$u_5$</td>
<td>0.8</td>
<td>0.1</td>
<td>0.1</td>
<td>0.4</td>
<td>0.2</td>
<td>0.3</td>
<td>1</td>
<td>0.3</td>
<td>0.1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 4.3: close($U, U'$) scores
### CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

\[ \text{conflict} (R, P) \]

<table>
<thead>
<tr>
<th></th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
<th>( p_6 )</th>
<th>( p_7 )</th>
<th>( p_8 )</th>
<th>( p_9 )</th>
<th>( p_{10} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>0.5</td>
<td>0.3</td>
<td>0.4</td>
<td>0.8</td>
<td>0.1</td>
<td>1</td>
<td>0.4</td>
<td>0.3</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>0.5</td>
<td>0.8</td>
<td>0.5</td>
<td>0.4</td>
<td>0.4</td>
<td>0.5</td>
<td>0.1</td>
<td>0</td>
<td>0.4</td>
<td>0.1</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0.3</td>
<td>0.6</td>
<td>0.3</td>
<td>0.9</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>1</td>
<td>0.9</td>
<td>0.4</td>
<td>0.2</td>
<td>0.1</td>
<td>1</td>
<td>0.5</td>
<td>0.2</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td>( r_5 )</td>
<td>0.8</td>
<td>0.1</td>
<td>0.1</td>
<td>0.4</td>
<td>0.2</td>
<td>0.3</td>
<td>1</td>
<td>0.3</td>
<td>0.1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 4.4: \( \text{conflict} (R, P) \) scores

\[ \text{appropriate} (R, P) \]

<table>
<thead>
<tr>
<th></th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
<th>( p_4 )</th>
<th>( p_5 )</th>
<th>( p_6 )</th>
<th>( p_7 )</th>
<th>( p_8 )</th>
<th>( p_9 )</th>
<th>( p_{10} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>0.5</td>
<td>0.7</td>
<td>0.6</td>
<td>0.2</td>
<td>0.4</td>
<td>0</td>
<td>0.5</td>
<td>0.7</td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>0.5</td>
<td>0.2</td>
<td>0.2</td>
<td>0.4</td>
<td>0.6</td>
<td>0.5</td>
<td>0.9</td>
<td>0.1</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>0.3</td>
<td>0.5</td>
<td>0.5</td>
<td>0</td>
<td>0.6</td>
<td>0.4</td>
<td>0.7</td>
<td>0.1</td>
<td>0.6</td>
<td>0.6</td>
</tr>
<tr>
<td>( r_4 )</td>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.7</td>
<td>0.3</td>
<td>0</td>
<td>0.4</td>
<td>0.8</td>
<td>0.1</td>
<td>0</td>
</tr>
<tr>
<td>( r_5 )</td>
<td>0.2</td>
<td>0.8</td>
<td>0.9</td>
<td>0.6</td>
<td>0.6</td>
<td>0.7</td>
<td>0</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Table 4.5: \( \text{appropriate} (R, P) \) scores
the affiliation $u_i$ of $r_i$ and the affiliation $u_j$ of the author of paper $p_j$. Furthermore note that, as the confl and appr scores directly depend on the atoms given as facts and since the rules defining these atoms always need to be completely satisfied, these atoms must also have the same score in any answer set of $P_{paper}$. Although the inappropriate scores are also only dependent on the appropriate atoms, the rules defining these atoms can be partially satisfied, meaning the inappropriate scores can be lower than the corresponding $N_W(appropriate)$ scores. Hence the difference between answer sets will be in the overworked, inappropriate, enough and assign scores.

Now, consider an approximate answer set $A_1$ with assignments as given in Table 4.6. The corresponding inappropriate scores can be found in Table 4.7 on the following page. One can check that $A_1$ indeed is an answer set by checking whether $\Pi^{A_1}_{p^{A_1}, \rho^{A_1}} = A_1$. For example, for $inappropriate(r_1, p_1)$ we can see that there is only one rule with $inappropriate(r_1, p_1)$ in its head, viz. $inappr_{r_1,p_1}$. Now the reduct of this rule is

$$inappr_{r_1,p_1}^A : inappropriate(r_1, p_1) \leftarrow l A_1(N_W(appropriate(r_1, p_1)))$$

which, by Table 4.5 on the preceding page, is equivalent to

$$inappr_{r_1,p_1}^A : inappropriate(r_1, p_1) \leftarrow l 0.5 \quad (4.7)$$

We can also compute $\rho_{A_1}(inappr_{r_1,p_1})$ as

$$\rho_{A_1}(inappr_{r_1,p_1}) = A_1(inappropriate(r_1, p_1) \leftarrow l 1 - appropriate(r_1, p_1))$$

which, by Table 4.5 and Table 4.7, is equal to

$$\rho_{A_1}(inappr_{r_1,p_1}) = T_W(0.5, 0) = 0.5 \quad (4.8)$$

Now using (4.7) and (4.8) we know that

$$\Pi^{A_1}_{p^{A_1}, \rho^{A_1}}(inappropriate(r_1, p_1)) = T_W(A_1(0.5), \rho_{A_1}(inappr_{r_1,p_1}))$$

$$= T_W(0.5, 0.5)$$

$$= 0$$

$$= A_1(inappropriate(r_1, p_1))$$

One can check that for all other atoms $l \in B_{P_{paper}}$ we also obtain that $\Pi^{A_1}_{p^{A_1}, \rho^{A_1}}(l) = A_1(l)$ and thus that $A_1$ is an answer set of $P_{paper}$.  

101
### CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

<table>
<thead>
<tr>
<th>assignments</th>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$r_1, r_2, r_3$</td>
<td>$r_1, r_2, r_4$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$r_1, r_3, r_5$</td>
<td>$r_1, r_3, r_4$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$r_1, r_3, r_5$</td>
<td>$r_1, r_4, r_5$</td>
</tr>
<tr>
<td>$p_4$</td>
<td>$r_2, r_4, r_5$</td>
<td>$r_2, r_4, r_5$</td>
</tr>
<tr>
<td>$p_5$</td>
<td>$r_2, r_3, r_5$</td>
<td>$r_2, r_3, r_5$</td>
</tr>
<tr>
<td>$p_6$</td>
<td>$r_2, r_3, r_5$</td>
<td>$r_2, r_3, r_5$</td>
</tr>
<tr>
<td>$p_7$</td>
<td>$r_1, r_2, r_3$</td>
<td>$r_1, r_2, r_3$</td>
</tr>
<tr>
<td>$p_8$</td>
<td>$r_1, r_4, r_5$</td>
<td>$r_1, r_4, r_5$</td>
</tr>
<tr>
<td>$p_9$</td>
<td>$r_2, r_3, r_5$</td>
<td>$r_2, r_3, r_5$</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>$r_1, r_3, r_5$</td>
<td>$r_1, r_3, r_5$</td>
</tr>
</tbody>
</table>

Table 4.6: Assignments for answer sets $A_1$ and $A_2$

<table>
<thead>
<tr>
<th>$\text{inappropriate}(R,P)$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
<th>$p_8$</th>
<th>$p_9$</th>
<th>$p_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8</td>
<td>0.6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>0</td>
</tr>
<tr>
<td>$r_2$</td>
<td>0</td>
<td>0.8</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>$r_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$r_4$</td>
<td>1</td>
<td>0.9</td>
<td>0.9</td>
<td>0</td>
<td>0.7</td>
<td>1</td>
<td>0.6</td>
<td>0</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td>$r_5$</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.7: $\text{inappropriate}(R,P)$ scores for $A_1$

<table>
<thead>
<tr>
<th>$\text{inappropriate}(R,P)$</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
<th>$p_4$</th>
<th>$p_5$</th>
<th>$p_6$</th>
<th>$p_7$</th>
<th>$p_8$</th>
<th>$p_9$</th>
<th>$p_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.8</td>
<td>0.6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>0</td>
</tr>
<tr>
<td>$r_2$</td>
<td>0</td>
<td>0.8</td>
<td>0.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>0.4</td>
</tr>
<tr>
<td>$r_3$</td>
<td>0.7</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.9</td>
<td>0</td>
</tr>
<tr>
<td>$r_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.7</td>
<td>1</td>
<td>0.6</td>
<td>0</td>
<td>0.9</td>
<td>1</td>
</tr>
<tr>
<td>$r_5$</td>
<td>0.8</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.8: $\text{inappropriate}(R,P)$ scores for $A_2$
Now, from the inappropriate scores in Table 4.7, we know that $\rho_{A_1}(\text{inappr}_{r_1,p_1}) = 0.5$ by (4.8). Likewise we can see from Table 4.5 and Table 4.7 that

$$A_1(\text{inappr}_{r_1,p_4}) = \mathcal{I}_W(\mathcal{N}_W(A_1(\text{appropriate}(r_1, p_4))), A_1(\text{inappropriate}(r_1, p_4))) = \mathcal{I}_W(0.8, 0.8) = 1$$

The values of $\text{inappr}_{r,p}$ for any $r \in \text{Reviewers}$ and $p \in \text{Papers}$ follow in the same fashion and thus we can compute

$$E_3(\rho_{A_1}) = \sum \{A_1(\text{inappr}_{r,p}) \mid r \in \text{Reviewers}, p \in \text{Papers}\} = 385/10$$

As the enough rules should always be completely satisfied, we have that $\text{enough}(p)$ for a certain paper $p$ should always be equal to $f_2(\sum \{\text{assign}(r,p) \mid r \in \text{Reviewers}\})$. Due to each paper having three reviewers in $A_1$ we then know that $A_1(\text{enough}(p))$ for any paper $p$ will be equal to $f_2(3) = 3/4$. This means that for any $\text{qual}_{p}$ rule we have that $A_1(\text{qual}_{p}) = 3/4$. As there are 10 papers in total we can thus compute:

$$E_4(\rho_{A_1}) = \sum \{A_1(\text{qual}_{p}) \mid p \in \text{Papers}\} = \sum \{(3/4) \mid p \in \text{Papers}\} = 10 \cdot (3/4) = 15/2$$

Now, in this answer set we have that, even though reviewer $r_1$ has 6 reviews, $A_1(\text{overworked}(r)) = 0$ for each assigned reviewer $r$; thus for some reviewers we underestimate their degree of being overworked to ensure that papers have enough reviews. From the foregoing, we can compute

$$E_5(\rho_{A_1}) = \sum \{A_1(\text{qual}_{r}) \mid r \in \text{Reviewers}\} = \sum \{(f_1(\sum \{A_1(\text{assign}(r,p)) \mid p \in \text{Papers}\}))\mathcal{I}(l)0 \mid r \in \text{Reviewers}\} = \sum \{1 - (f_1(\sum \{A_1(\text{assign}(r,p)) \mid p \in \text{Papers}\})) \mid r \in \text{Reviewers}\} = 19/7$$

Hence we obtain that

$$A_{p\text{aper}}(\rho_{A_1}) = E_1(\rho_{A_1}) \cdot E_2(\rho_{A_1}) \cdot (E_3(\rho_{A_1}) + 10 \cdot E_4(\rho_{A_1}) + 20 \cdot E_5(\rho_{A_1})) = 2349/14 \approx 167.8$$
There is room for improvement, however, as this answer set clearly contains a high work burden for some reviewers, while creating a minimal workload for others. Due to the nature of our aggregator expression, we can spread the papers among reviewers, potentially giving papers to reviewers with a lower knowledge of the domain, to obtain a better answer set. Answer set $A_2$, for which the assignments are given in Table 4.6, relieves the burden of reviewers $r_3$ and $r_5$ by assigning more reviews to reviewer $r_4$. Computing the value of the aggregator expression, we first obtain from the inappropriate scores in Table 4.8 that $E_3(\rho_{A_2}) = \sum \{A_2(\text{inappr}_{r,p}) \mid r \in \text{Reviewers}, p \in \text{Papers}\} = 371/10$; furthermore we can compute in a similar fashion as for $A_1$ that $E_4(\rho_{A_2}) = \sum \{A_2(\text{qual}_{p}) \mid p \in \text{Papers}\} = 15/2$ as every paper again has three reviewers assigned. Now, by calculating the assignments per reviewer we obtain $E_5(\rho_{A_2}) = \sum \{A_2(\text{qual}_{r}) \mid r \in \text{Reviewers}\} = 20/7$. The foregoing shows $A_{\text{paper}}(\rho_{A_2}) = 11847/70 \approx 169.2$, hence answer set $A_2$ is more suitable than $A_1$, as we expected.

### 4.4 Relationship to Existing Approaches

The combination of answer set programming and logic programming with uncertainty and many-valued theories has received a great deal of attention over the past years. Among others, there have been extensions of logic programming using probabilistic reasoning [30, 61, 110, 111, 129, 130, 163], possibilistic reasoning [1, 132, 133], fuzzy reasoning [22, 77, 112–115, 117, 118, 147, 163, 175–177], and more general many-valued or uncertainty reasoning [28, 29, 31–33, 52, 59, 87, 88, 91, 92, 94–96, 103, 104, 107, 128, 155, 159, 160, 164, 165]. Roughly, one can divide these approaches in two classes, viz. implication-based (IB) and annotation-based (AB) frameworks.

In the implication-based setting a rule is generally of the form

$$a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m)$$

where $a$ is an atom, $f$ is a total, finitely computable $\mathcal{L}^{n+m} \rightarrow \mathcal{L}$ function that is increasing in its $n$ first arguments and decreasing in its $m$ last and $w \in \mathcal{L}$, with $\mathcal{L}$ the lattice used for truth values. For convenience, we will use $\alpha$ as a short-hand for $f(b_1, \ldots, b_n; c_1, \ldots, c_m)$. Intuitively, such a rule denotes that in any model of the program the truth degree of the implication $\mathcal{I}(a, a)$ must be greater than or equal to the weight $w$. In the annotation-based approaches one considers annotations, which are either constants from the truth lattice $\mathcal{L}$, variables ranging over this truth lattice, or functions over elements of this truth lattice applied to annotations. A
rule is then of the form

\[ a : \mu \leftarrow b_1 : \mu_1, \ldots, b_n : \mu_n \]

where \(a, a_1, \ldots, a_n\) are atoms and \(\mu, \mu_1, \ldots, \mu_n\) are annotations. Intuitively, an annotated rule denotes that if the certainty of each \(b_i\), \(1 \leq i \leq n\), is at least \(\mu_i\), then the certainty of \(a\) is at least \(\mu\). The links between these two approaches are well-studied in e.g. \([33, 88, 95, 96]\) and we will therefore not repeat these results. In this section, we give an overview of these related approaches and study the links between our framework and related proposals.

### 4.4.1 Fuzzy and Many-Valued Logic Programming Without Partial Rule Satisfaction

Many proposals for fuzzy and many-valued logic programming with rules that have to be completely fulfilled have been published. In this category one finds most annotation-based (AB) approaches, e.g. \([22, 87, 147, 160, 165]\) and some implication-based (IB) approaches where the weight of each rule is 1, e.g. \([31–33, 92]\). The latter includes the FASP framework we discussed in Chapter 3. Some of these proposals only contain monotonic functions in rules (e.g. the AB approach from \([22, 88]\) and the IB approaches from \([32, 33, 92]\)), while others feature negation (e.g. the AB approaches from \([87, 147, 160, 165]\)) or even arbitrary decreasing functions (e.g. the IB approach from \([31]\)). These proposals differ from ours as they do not incorporate the idea of partial rule satisfaction.

We can readily embed the IB approaches in our framework by supplying these programs with the infimum aggregator. When modeled like this, the 1-answer sets of the embedding will correspond exactly to the answer sets of programs from the aforementioned frameworks. Thanks to this embedding we also inherit the modeling power that is already present in some of these proposals. For example, from the embeddings shown in \([30, 33]\), and using the fact that we can embed \([31]\) in our approach, we inherit the capacity to model Generalized Annotated Logic Programs \([88]\), Probabilistic Deductive Databases \([93]\), Possibilistic Logic Programming \([43]\), Hybrid Probabilistic Logic Programs \([40]\)\(^2\) and Fuzzy Logic Programming \([176]\).

The AB approach from \([22]\) is interesting in that annotations are actually fuzzy sets, which allows for an intuitive modeling language. Whether the semantics of this

\(^2\)Note that the translation process in \([30]\) is exponential in the size of the program, but, as the authors point out, this is to be expected as reasoning in these programs in most cases is exponential.
specific framework can be truthfully embedded in our approach is not immediately clear, but as the family of all fuzzy sets in a given universe forms a complete lattice, when equipped with the Zadeh intersection and union\(^3\), the use of fuzzy sets, together with functions over fuzzy sets is certainly possible in the AFASP language.

Another interesting approach is the use of bilattices for giving the semantics of logic programs, as was initiated by Fitting in [59]. It turns out that by using bilattices an elegant characterization of many-valued answer set programming can be shown [106] that clarifies the role of the closed world assumption in (many-valued) logic programming [105]. Furthermore, this characterization can be used for defining a top-down query procedure over many-valued logic programs [161,162].

4.4.2 Weighted Rule Satisfaction Approaches

Some IB approaches to fuzzy and many-valued logic programming feature partial rule fulfillment by adding “weights” to rules (e.g. [28, 29, 52, 95, 96, 103, 104, 112, 113, 117–120, 122, 123, 155]). These weights are specified manually and they reflect the minimum degree of fulfillment required for a rule. Formally, such a rule takes on the form of

\[ a \overset{w}{\leftarrow} \alpha \]

where \( a \) is an atom, \( \alpha \) is a body expression, \( \overset{w}{\leftarrow} \) is a residual implicator over \([0, 1]\) and \( w \) is a value of \([0, 1]\). We will use \( r_h \) and \( r_b \) to refer to the head, resp. the body of a rule \( r \) as usual, and \( r_w \) to refer to the weight \( w \). In the case of [52,112,113,117,118,122,123,155], the bodies of rules are restricted to combinations of triangular norms, possibly with negation-as-failure literals in [112,113,117–120], whereas in [28,29,95,96,103,104] the bodies consist of monotonically increasing functions, where some approaches do not feature non-monotonic negation [28,29,95,96] and others feature negation under the well-founded semantics [103,104]. Furthermore, [113,122,123] allow disjunctions in the head of rules and [28] allows a combination of multiple lattices to be used. This last feature is obtained in the AFASP setting by using the Cartesian product of all these lattices as the lattice for program rules and using the corresponding projections to extend the operators to this product lattice.

\(^3\)For two mappings \( A \) and \( B \) from \( X \) to \([0, 1]\) the Zadeh union and intersection are the \( X \rightarrow [0, 1] \) mappings \( \cup \) and \( \cap \) respectively defined as \( (A \cup B)(x) = \max(A(x), B(x)) \) and \( (A \cap B)(x) = \min(A(x), B(x)) \).
4.4. RELATIONSHIP TO EXISTING APPROACHES

The semantics of a program consisting of weighted rules without negation-as-failure is defined in two ways in the literature. We will take [113] and [28] as examples of these two methods, but the following discussion equally applies to all the approaches mentioned earlier, barring some minor syntactical issues. In the case of [113], an interpretation \( M \) is called a model of a program \( P \) when for all \( r \) in \( P \) it holds that \( M(r_h) \geq T(r(M(r_b), r_w)) \). Answer sets of these programs are then defined as minimal models. In [28], answer sets are defined as the least fixpoints of an immediate consequence operator, defined for a program \( P \), interpretation \( I \) of \( P \) and atom \( l \in B_P \), as:

\[
\Pi_P(I)(l) = \sup\{T_r(I(r_b), r_w) \mid r \in P_l\}
\]

It is known that these two semantics coincide, which can also be shown using the results on AFASP, as demonstrated below.

Note that due to Proposition 4.9, it holds that \( \Pi_P = \Pi_{P, \rho_w} \), where \( \rho_w(r) = r_w \). Hence, for simple AFASP programs, the semantics of [28] can be obtained by taking the least fixpoint w.r.t. the rule interpretation corresponding to the rule weights. Furthermore, from Proposition 4.26, we know that the least fixpoint of \( \Pi_{P, \rho_w} \) corresponds to the minimal \( \rho_w \)-rule model of \( P \). An interpretation \( M \) is a \( \rho_w \)-rule model of \( P \) iff for each rule \( r \in P \) it holds that \( M(r_h) \geq \rho_w(r) \) and hence, by the residuation principle, that \( M(r_h) \geq T_r(M(r_b), \rho_w(r)) \). This means that \( \rho_w \)-rule models correspond to models in the sense of [113]. Hence the semantics of [113] and [28] coincide and can both be generated from a simple AFASP program.

Furthermore, due to the equivalence between \( \Pi_P \) and \( \Pi_{P, \rho_w} \), we can use the termination conditions from [28] to determine structural conditions that ensure that the computation of the least fixpoint of \( \Pi_{P, \rho} \) ends.

Note that, different to AFASP, programs with weight rules without negation-as-failure only have a single answer set corresponding to the minimal model satisfying the rules to the stated weights. This means that the weight of a rule should not simply be seen as the minimal degree of satisfaction, but that the semantics of these programs correspond to a cautious use of the rules and their weights.

At first, one might think that these semantics can easily be embedded in the AFASP framework by moving the weights into the aggregator expression and using them as lower bounds on the satisfaction of their corresponding rules. Formally, this means that we define for a program \( P \) in the sense of [28,113] and rule base \( \{r_1, \ldots, r_n\} \) with corresponding weights \( w_1, \ldots, w_n \), the program \( P' \) with rule base \( R_{P'} = \{r_i : r_h \leftarrow r_b \mid i \in 1 \ldots n\} \) and aggregator expression \( A_{P'}(\rho) = (\rho(r_1) \geq w_1) \land \ldots \land (\rho(r_n) \geq w_n) \).
... ∧ (ρ(r) ≥ w). Note that in the former we regard ≥ as a boolean expressing and identify 0 with False and 1 with True. The semantics of programs P and P' do not coincide, however, as shown in the following example.

**Example 4.43.** Let P be a program in the sense of [28,113] with rule base \( R_P = \{ r : a \leftarrow_m 0.5 \} \). It is easy to see that the unique answer set of this program is \( \{ a^{0.5} \} \). The corresponding AFASP program \( P' \) with rule base \( R_{P'} = \{ r : a \leftarrow_m 1 \} \) and aggregator expression \( A_{P'}(\rho) = \rho(r) \geq 0.5 \), however, has multiple k-answer sets for varying \( k \in [0,1] \), such as the 1-answer set \( \{ a^1 \} \). The original answer set of P is only an 0.5-answer set of \( P' \).

A proper embedding of these semantics in the AFASP framework can be obtained as follows. Suppose P is a program with weighted rules, then the corresponding AFASP program \( P_{\text{weight}} \) is defined as

\[
P_{\text{weight}} = \{ r_h \leftarrow \mathcal{T}_r(r_b, r_w) \mid r \in P \}
\]

The lattice to be used for evaluating the rules is then \(([0,1], \leq)\), the aggregator lattice is \((\{0,1\}, \leq)\). The aggregator expression is given as

\[
A_{P_{\text{weight}}}(\rho) = \forall r \in P_{\text{weight}} \cdot (\rho(r) = 1)
\]

**Example 4.44.** Consider program \( P \) from Example 4.43. Using the proper embedding discussed above we obtain an AFASP program \( P_{\text{weight}} \) with rule base \( R_{P_{\text{weight}}} = \{ r : a \leftarrow_m \mathcal{T}_M(1, 0.5) \} \) and aggregator expression \( A_{P_{\text{weight}}}(\rho) = (\rho(r) = 1) \). The 1-answer set of \( P_{\text{weight}} \) is now \( \{ a^{0.5} \} \), which corresponds to the answer set of \( P \).

The following proposition shows that for simple AFASP programs, this is indeed a truthful embedding of these semantics:

**Proposition 4.45.** Let \( P \) be a simple AFASP program in the sense of [28]. Then A is a 1-answer set of \( P_{\text{weight}} \) iff \( A = \Pi^*_P \).

**Proof.** First, remark that there is only a single rule interpretation \( \rho \) such that \( \forall r \in P_{\text{weight}} \cdot \rho(r) = 1 \) holds, viz. \( \rho^+ \). As \( A \) is only a 1-answer set of \( P_{\text{weight}} \) iff \( A = \Pi^*_P \rho \), for some rule interpretation satisfying \( \forall r \in P_{\text{weight}} \cdot \rho(r) = 1 \) due to
4.4. RELATIONSHIP TO EXISTING APPROACHES

Corollary 4.32, it must hold that there is a unique 1-answer set \( A \) of \( P_{\text{weight}} \) where
\[ A = \Pi_{P,\rho^\top}^\star. \]
We can now show that \( \Pi_P = \Pi_{P,\rho^\top} \), leading to the stated equivalence, as follows. Suppose \( I \) is some interpretation of \( P \) (and hence also of \( P_{\text{weight}} \)) and \( l \in B_P \) (and hence also \( l \in B_{P_{\text{weight}}} \)), then:

\[
\Pi_P(I)(l) = \langle \text{Def. } \Pi_P \rangle \sup \{ T_r(I(r_b), r_w)I \mid r \in P_l \}
= \langle \text{Def. } P_{\text{weight}} \rangle \sup \{ I(r_b) \mid r \in (P_{\text{weight}})_l \}
= \langle \text{Def. } \rho^\top \rangle \sup \{ T_r(I(r_b), \rho^\top(r))I \mid r \in (P_{\text{weight}})_l \}
= \langle \text{Def. } \Pi_{P_{\text{weight}},\rho^\top} \rangle \Pi_{P_{\text{weight}},\rho^\top}(I)(l)
\]

From this, the stated readily follows.

For approaches featuring negation-as-failure under the answer set semantics in the body of rules, we will again take [113] as a representative example. Although the approaches in [117,118] are slightly different in that the reduct operation moves the value of the negated literals in the weight of the rule instead of directly substituting it in the rule body, the net result is the same. Negation-as-failure in this approach is denoted as \( \text{not}_N l \), where \( l \) is an atom and \( N \) a negator over \([0,1]\). If \( P \) is a program with negation-as-failure in the sense of [113], then an interpretation \( A \) is called an answer set of \( P \) if \( A \) is the answer set of \( P_{A} \), where \( P_{A} \) is a generalized Gelfond-Lifschitz transformation replacing all negation-as-failure literals \( \text{not}_N l \) by the value \( N(A(l)) \). It is easy to see that this Gelfond-Lifschitz transformation is a special case of the reduct we introduced for FASP programs. From this, it easily follows that the embedding mentioned before still works in the presence of negation-as-failure. Hence we obtain the following proposition.

**Proposition 4.46.** Let \( P \) be a program in the sense of [113]. Then \( A \) is a 1-answer set of \( P_{\text{weight}} \) iff \( A \) is an answer set of \( P \) in the sense of [113].

**Proof.** The proof follows easily from Proposition 4.45 and the fact that the Gelfond-Lifschitz reduct introduced in [113] is a special case of our reduct.

Although these proposals feature partial rule satisfaction and are therefore better equipped to model real-life phenomena than previous proposals, the use of weights introduces the new problem of “weight-guessing”. The AFASP framework eliminates the weight-guessing problem by using an aggregator expression, which encodes which combinations of partially satisfied rules are more desirable than others. In this light, one could think of AFASP programs as programs with variables as rule-weights instead of fixed weights, where the variables must be chosen according
to the aggregator expression. In effect, due to Corollary 4.32 and the fact that rule interpretations are analogous to weights on rules, this means that semantically, a single AFASP program corresponds to a set of programs with weights. This shows that the aggregator expression has a substantial modeling advantage over attaching arbitrary weights.

Interestingly, in [119,120], a measure is introduced that shows how the weights of the rules of a program without stable models should be increased or decreased to obtain a new program with stable models. This is related our approach, where rule weights are variable and can be changed to obtain approximate answer sets, if none exists. Our approach differs in the fact that we do not have predefined weights on rules and assume that the best solution is the solution satisfying all the rules best. Furthermore, due to the aggregator, we can compare approximate answer sets in our framework with a preference order that is chosen by the programmer.

4.4.3 Van Nieuwenborgh et al.

In [175], a language that is similar to AFASP is introduced. Our presented approach generalizes this framework by allowing a much richer vocabulary of expressions to use in rules and by allowing more sophisticated aggregator expressions. Specifically, we allow arbitrary monotonic functions in rule bodies, whereas [175] only allows t-norms and negators in bodies. Furthermore, we base our semantics on fixpoints, which more clearly shows the link between other FASP approaches, and also fixes a problem with the semantics of [175] when generalizing to arbitrary lattices as truth values, as demonstrated below. Moreover, it is not clear how the semantics of [175] can be extended to deal with arbitrary monotonic functions in rule bodies, where this is straightforward in our fixpoint approach.

Semantically, in [175], an answer set \( M \) has a degree \( k \), which, as in the present approach, reflects the value of an aggregator function that combines the degree of satisfaction of the rules in the program. However, as opposed to the present approach, this aggregator must have a value in the same lattice as the one used for the rules. As shown throughout the examples in this chapter, our more general aggregator can be advantageous for modeling certain real-life problems. Furthermore, an answer set is defined in [175] as a model that is free from unfounded sets. Intuitively, the concept of unfounded set provides a direct formalization of "badly motivated" (as described in Section 4.2.2) conclusions.
Definition 4.47. A set $X$ of atoms is called **unfounded** w.r.t. an interpretation $I$ of a program $P$ iff for all $a \in X$, every rule $r \in P_a$ satisfies either

(i) $X \cap r^+_b \neq \emptyset$, or
(ii) $I_s(a, \rho_I(r)) < I(a)$, or
(iii) $I(r_b) = 0$

where for $r_b = T(b_1, \ldots, b_n, N_1(c_1), \ldots, N_m(c_m))$, with $T$ an arbitrary t-norm and $N_i$ arbitrary negators, we define $r_b^+ = \{b_1, \ldots, b_n\}$.

Intuitively, condition (i) above describes a circular motivation while (ii) asserts that $a$ is overvalued w.r.t. $r$. Condition (iii) is needed to ensure that the semantics are a proper generalization of the classical semantics. An interpretation $I$ is called **unfounded-free** iff $\text{supp}(I) \cap X = \emptyset$ for any set $X$ that is unfounded w.r.t. $I$. Answer sets to a degree $k$ are then defined in [175] as those $k$-models that are unfounded-free. A nice feature is that this single definition covers any program $P$, regardless of whether it is positive, or has constraint rules. One may wonder whether the concept of unfounded set can be generalized to AFASP programs. For example, a natural generalization would simply replace the circularity definition $X \cap r_b \neq \emptyset$ above by “some element of $X$ occurs as an argument in which the body expression $r_b$ is increasing”. However, this approach fails, as illustrated by the program $P$ from Example 4.28 where it can easily be verified that $\{a\}$ would be unfounded w.r.t. the interpretation $\{a^1\}$ since $r_1 : a \leftarrow_m 0.2$ satisfies (ii) above while $r_2 : a \leftarrow_m \text{gt}(a, 0)$ satisfies (i). Note that this failure is only due to the presence of $\text{gt}(a, 0)$ in rule bodies, which are not allowed in the framework presented in [175].

We now show the novel result that when a total ordering is used in the lattice, the semantics of [175] correspond to the semantics of our fixpoint definition (when obeying the syntactic restrictions noted above). First, note that the concept of $k$-models in [175] coincides with Definition 4.6 on page 76. Hence, we only need to show that an interpretation $I$ of $P$ is unfounded-free iff $I = \Pi_{P, \rho_1} I$.

**Lemma 4.48.** Let $P$ be an AFASP program. For any interpretation $I$ of $P$ it holds that $I = \Pi_{P, \rho_1} I$ iff $I = \Pi_{R^I_{P, \rho_1}} I$.

**Proof.** Follows trivially by the construction of $R^I_{P, \rho_1}$. 

\footnote{We recall that for an interpretation $I$ of a program $P$ the set $\text{supp}(I)$ contains all atoms $a \in B_P$ for which $I(a) > 0$. See Definition 2.37 on page 41.}
Chapter 4. Aggregated Fuzzy Answer Set Programming

Lemma 4.49. Let $P$ be an AFASP program with only t-norms and negators in rule bodies. Then any unfounded-free interpretation $I$ of $P$ is a fixpoint of $\Pi_{P,\rho_1}$.

Proof. Let $I$ be an unfounded-free interpretation of $P$. We show that $I(a) = \sup\{I_s(r,\rho_1(r)) \mid r \in P_a\} = \Pi_{P,\rho_1}(I)(a)$ for any $a \in B_P$, from which the stated readily follows. The proof is split into the case for $a \in \text{supp}(I)$ and $a \notin \text{supp}(I)$.

For any $a \in \text{supp}(I)$ it must hold that $\{a\}$ is not unfounded w.r.t. $I$, meaning that $P_a \neq \emptyset$ and there is some $r \in P_a$ such that $I(a) \leq I_s(r,\rho_1(r))$. Due to Proposition 4.9 on page 78 this means

$$I(a) \leq \tau_r(I(r_b),\rho_1(r))$$

By definition of $\rho_1$ we know that for any $r' \in P_a$ we have that $\tau_r(I(r'_b),I(a)) \geq \rho_1(r')$. Due to Proposition 2.52 on page 46 this means $I(a) \geq \tau_r(I(r'_b),\rho_1(r'))$. Combining this with (4.9) and Proposition 4.9 we obtain that $I_s(r,\rho_1(r)) = I(a) \geq I_s(r',\rho_1(r'))$ for any $r' \in P_a$. Hence $I_s(r,\rho_1(r)) = I(a)$ is the supremum of $\{I_s(r',\rho_1(r')) \mid r' \in P_a\}$.

The case for $a \notin \text{supp}(I)$ is as follows. First, remark that in the former case we already showed that for any $r \in P_a$ we have that $I(a) \geq \tau_r(I(r_b),\rho_1(r))$. Since $I(a) = 0$ this means that $I_s(r,\rho_1(r)) = 0$ for any $r \in P_a$ due to Proposition 4.9. Hence $I(a) = \sup\{I_s(r,\rho_1(r)) \mid r \in P_a\}$.

Proposition 4.50. Let $P$ be an AFASP program with only t-norms and negators in rule bodies. An interpretation $I$ of $P$ is unfounded-free iff $I = \Pi_{R_{P,\rho_1}}^*$.

Proof. In [175] it was shown that the least fixpoint of $\Pi_{R_{P,\rho_1}}^*$ must necessarily be unfounded-free (Proposition 4). Hence we only need to show that if $I$ is unfounded-free, it is the least fixpoint of $\Pi_{R_{P,\rho_1}}^*$.

Suppose $I \neq \Pi_{R_{P,\rho_1}}^*$. Then, since any unfounded-free interpretation is a fixpoint of $\Pi_{R_{P,\rho_1}}^*$ due to Lemmas 4.48 and 4.49, it holds that some set $I' \subset I$ exists such that $I' = \Pi_{R_{P,\rho_1}}^*$.

Consider then $U = \{u \in B_P \mid I'(u) < I(u)\}$. Surely $U \subseteq \text{supp}(I)$ and hence $U \cap \text{supp}(I) \neq \emptyset$. We now show that $U$ is unfounded with respect to $I$, leading to a contradiction. First, we show that for any atom $u \in U$ and rule $r \in P_u$ it holds that

$$r^+_b \cap U = \emptyset \Rightarrow I_s(r,\rho_1(r)) < I(u)$$

(4.10)
4.4. RELATIONSHIP TO EXISTING APPROACHES

as follows

\[ r_b^+ \cap U = \varnothing \]

\[ \equiv \langle \text{Def. } \cap \rangle \]

\[ \neg \exists l \in r_b^+ \cdot l \in U \]

\[ \equiv \langle \text{Duality } \forall, \exists \rangle \]

\[ \forall l \in r_b^+ \cdot l \not\in U \]

\[ \equiv \langle \text{Def. } U \rangle \]

\[ \forall l \in r_b^+ \cdot I(l) = I'(l) \]

\[ \Rightarrow \langle I(l) = I(l') \rangle \]

\[ I(r_b^+) = I'(r_b^+) \]

\[ \equiv \langle \text{Leibniz} \rangle \]

\[ \mathcal{T}_r(I(r_b^+), \rho_1(r^l)) = \mathcal{T}_r(I'(r_b^+), \rho_1(r^l)) \]

\[ \equiv \langle \text{Prop. 4.9} \rangle \]

\[ I_s(r, \rho_1(r^l)) = I'_s(r', \rho_1(r')) \]

\[ \Rightarrow \langle \text{Monotonicity sup} \rangle \]

\[ I_s(r, \rho_1(r^l)) \leq \sup_{r' \in \mathcal{R}_P} I'_s(r', \rho_1(r')) \]

\[ \equiv \langle I' = \Pi_{\mathcal{R}_P} \rangle \]

\[ I_s(r, \rho_1(r^l)) \leq I'(u) \]

\[ \Rightarrow \langle u \in U, \text{ Def. } U \rangle \]

\[ I_s(r, \rho_1(r^l)) < I(u) \]

Thus, since it follows from the Definition of \( r^l \) that \( I(r_b^+) = I(r_b) \) and \( \rho_1(r^l) = \rho_1(r) \), we have shown that (4.10) holds. From this equation we obtain that

\[ \left( r_b^+ \cap U \neq \varnothing \right) \lor \left( I_s(r, \rho_1(r)) < I(u) \right) \]

Hence

\[ \left( r_b^+ \cap U \neq \varnothing \right) \lor \left( I_s(r, \rho_1(r)) < I(u) \right) \lor \left( I(r_b) = 0 \right) \]

Which means \( U \) is unfounded with respect to \( I \), a contradiction. \( \Box \)

When the ordering used is not total, however, this equivalence is no longer valid. For example, consider the lattice \((\mathcal{B} \times \mathcal{B}, \leq)\) such that \((1, 1)\) is the top element of the lattice, \((0, 0)\) is the bottom element and \((0, 0) \leq (0, 1) \leq (1, 1) \) and \((0, 0) \leq (1, 0) \leq (1, 1)\). Now consider an AFASP program \( P \), with \( \mathcal{A}_P(\rho) = \inf\{\rho(r) | r \in P\} \), over this lattice:

\[ r_1 : a \leftarrow (1, 0) \]

\[ r_2 : a \leftarrow (0, 1) \]

According to Proposition 2.54 on page 47, any interpretation \( I \) of \( P \) that satisfies rule \( r_1 \) to the degree \((1, 1)\) must obey \( I(a) \geq (1, 0) \). Likewise any interpretation \( I \) that satisfies \( r_2 \) to the degree \((1, 1)\) must obey \( I(a) \geq (0, 1) \). Hence the only 1-model of \( P \) is \( I = \{ a^{(1,1)} \} \). However, according to rule (ii) of Definition 4.47 \( \{ a \} \) is an unfounded set, which means that under the unfounded-based semantics \( I \) is not an answer set of \( P \). On the other hand, \( I = \Pi_{\mathcal{R}_P}^{+} \rho_1 \), and thus \( I \) is an answer set of \( P \) according to the fixpoint semantics. If we consider rules as constraints that need to be fulfilled, the fixpoint semantics correspond better to our intuition.
4.4.4 Valued Constraint Satisfaction Problems

A classical constraint satisfaction problem (CSP) consists of a set of variables $X = \{x_1, \ldots, x_n\}$, a set of finite domains $D = \{d_1, \ldots, d_n\}$ such that variable $x_i$ ranges over domain $d_i$, and a set of constraints $C$ of the form $c = (X_c, R_c)$ such that $X_c \subseteq X$ is a set of variables and $R_c$ is a relation between the variables in $X_c$. In Valued Constraint Satisfaction Problems (VCSPs) [150], a CSP is augmented with a cost function $\phi$, which associates a cost to every constraint. A solution to a VCSP is then an assignment of values to the variables in $X$ such that the aggregated cost of all violated constraints is minimal. Typically, costs are represented as real numbers, and the maximum or sum is used to aggregate.

In the crisp case it has been noted that answer set programming can be used for solving constraint satisfaction problems [121,135]. The idea is to write an answer set program containing generate rules\(^5\) that generate possible assignments of values to each of the variables, and constraints which remove those assignments that violate any constraints. In this way the resulting answer set program models a constraint satisfaction problem in the sense that answer sets of the program correspond to the solutions of the problem under consideration. It should come as no surprise that the AFASP framework can likewise be used for modeling VCSPs, as VCSPs can be seen as CSPs with an added aggregation operator. Basically, a VCSP corresponds to an AFASP program that only uses choice rules (which are the fuzzy equivalent of generate rules, i.e. rules assigning a random truth value to a certain atom) and constraints. Hard constraints correspond to rules that are required to be greater than 1 in the aggregator, whereas soft constraints are rules whose valuation in the aggregator can be lower than 1, such as rules aggregated using the infimum. An example of the use of this paradigm is the fuzzy graph coloring program introduced in Section 4.1, where we model the constraint satisfaction problem of coloring a graph with continuous colors, given some soft and hard constraints.

4.4.5 Answer Set Optimization

In [19, 20], a framework for answer set optimization is proposed. The basic idea is that one can state preference rules which are combined to define an ordering over answer sets. For example, if we have a program that generates a class room

\(^5\)These are rules involving cyclic negation such as the ASP program [{\text{a} \leftarrow \text{not b}. \text{b} \leftarrow \text{not a}}]. The answer sets of this program are \{\text{a}\} and \{\text{b}\}, hence this program allows us to choose one of two options \text{a} and \text{b} in a solution of the modeled problem.
schedule, this framework allows to state that if teacher John is teaching Math, we prefer John to also teach Physics as follows:

\[
\text{teaches(John, Physics) : 0 > teaches(Mark, Physics) : 1} \leftarrow \text{teaches(John, Math)}
\]

A rule is of the general form

\[
C_1 : p_1 > \ldots > C_k : p_k \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m
\]

where \( C_i \) : \( p_i \) encodes that the penalty associated with the rule is \( p_i \) if \( i \) is the lowest index for which the atom set \( C_i \) is true. However, the penalty of the rule is only taken into account if the conditions on the left hand side, expressed as a conjunction of atoms, are true. Different rules can then be combined using strategies, which encode importance among these preference rules. Formally, [19] defines a Preference Description Language PDL in which one can for example write that answer sets should be ordered using the Pareto ordering on rules \( r_1 \) and \( r_2 \) as \((\text{pareto } r_1, r_2)\). Many other complex (combinations of) orderings can be written in this language, such as \((\text{lex } (\text{pareto } r_1, r_2), r_3)\) which denotes that two answer sets first need to be compared using the Pareto ordering on rules \( r_1 \) and \( r_2 \); if they are Pareto-equal, then one must try to discriminate between them on the basis of rule \( r_3 \).

It is clear that the ideas of this approach and the one we proposed in this chapter are very similar. In [82] we showed how this framework could be generalized to AFASP, using an appropriate aggregator. For a practical implementation of AFASP it seems interesting to adopt the same strategy of having a fixed language for specifying the aggregator. For example, we could then write an aggregator defining a lexicographical ordering over the program rules \( r_1, r_2, r_3 \) as \((\text{lex } r_1, r_2, r_3)\). Furthermore, the idea of computing optimal answer sets by means of a generating program and optimality checking program could also be generalized to AFASP.

## 4.5 Summary

When using FASP to solve continuous optimization problems it is natural to consider approximate solutions, which correspond to answer sets that satisfy some of the rules partially. Current approaches require users to annotate rules with fixed weights, however, which is not flexible and leaves the programmer with the problem of guessing the right weights.
CHAPTER 4. AGGREGATED FUZZY ANSWER SET PROGRAMMING

In this chapter we have introduced aggregated fuzzy answer set programming (AFASP), which uses aggregation functions to combine the degrees to which the rules are satisfied to a single value from a preordered set. Essentially, this approach attaches variable weights to rules, where the aggregator function determines the most desirable combinations of these weights. In contrast to languages with fixed weights this means that simple AFASP programs can have multiple answer sets.

Different to a previous proposal for fuzzy answer set programming with aggregators we base our semantics on fixpoints instead of unfounded sets, allow arbitrary monotonic functions instead of only t-norms in rule bodies and decouple the aggregator from the lattice underlying the program. We have shown that the unfounded-based semantics cannot easily be generalized to programs with arbitrary monotonic functions in rule bodies, whereas this is trivial with the fixpoint semantics. Furthermore, we proved that the fixpoint semantics coincide with the unfounded-based semantics when the lattice that is used in the program is total. This is an important result that will be used extensively in Chapter 6. For non-total lattices we have moreover shown that, in contrast to the fixpoint semantics, the unfounded-based semantics produce counter-intuitive results.

We also studied AFASP itself in great detail. Most importantly we demonstrated the relations that exist between our notion of approximate models and the fixpoints of our generalization of the immediate consequence operator of FASP. For example, in Proposition 4.22 we proved the relation between minimal approximate models and the fixpoints of the the AFASP immediate consequence operator and in Proposition 4.38 we proved that answer sets correspond to certain minimal approximate models. Furthermore we showed a number of interesting properties that are also relevant for FASP languages with fixed weights. For example, in Proposition 4.19 we proved that lower rule weights will lead to smaller answer sets.

Finally, we illustrated the use of AFASP on the reviewer assignment problem. The AFASP program that solves this problem was built in the usual generate-define-test pattern and used partial rule satisfaction to rank solutions according to the workload of each reviewer and the number of reviewers that each paper has.

From the results in this chapter we can conclude that the addition of partial rule satisfaction to FASP is very useful when modeling continuous optimization problems. However, as we have seen throughout the chapter, this also makes AFASP harder to reason about than FASP. In the next chapter we therefore investigate whether AFASP and other extensions of FASP can be simulated using a simpler core language.
5.1 Introduction

The study of extensions of classical ASP has received a great deal of attention over the past years, including the efforts of the European Working Group on Answer Set Programming (WASP) [134]. The main objectives of such a study are (1) researching the complexity and additional expressivity which certain extensions bring; (2) investigating whether extensions can be compiled to a core language that is easy to implement, or is already implemented. Certain interesting links have been brought to light in this research. For example, it has been shown that nested expressions can be translated to disjunctive logic programs [139] and that aggregates can be translated to normal logic programs [140]. Next to these general extensions of ASP, the translation of other frameworks to ASP has also been studied. For example, DLV supports abduction with penalization [141] through its front-end by compiling this framework to a logic program with weak constraints [21]. For preferences in ASP a common implementation method is to use a meta-formalism and first
generate all answer sets for a program, and then filter the most preferred ones. Though the preference extensions have a higher complexity, this method ensures that programs with preferences can still be solved using off the shelf ASP solvers such as Smodels [156] and DLV [53].

Over the years many different FASP formalisms have been proposed. Some of these only allow arbitrary monotonic functions (e.g. [88]) in rule bodies, whereas others have negation-as-failure (e.g. [117, 160]) or decreasing functions (e.g. [31] and the FASP framework that we described in Chapter 3). In contrast to ASP, the study of whether these formalisms can be compiled to a core language has not received much attention. One notable exception is [31], which has been shown to be capable of simulating Generalized Annotated Logic Programs [88], Probabilistic Deductive Databases [93], Possibilistic Logic Programming [43], Hybrid Probabilistic Logic Programs [30] and Fuzzy Logic Programming [176]. However, this study addresses neither constructs in existing FASP formalisms nor their extensions. Furthermore, it compiles the aforementioned languages to a FASP formalism that is quite involved.

In this chapter we investigate the expressivity of different constructs and extensions of existing FASP formalisms and show that many of them can be simulated in a language that is considerably simpler. This creates a bridge between the desire to have a rich and expressive FASP language on one hand and the wish to have a small core theory on the other hand. The advantage of the former is that it removes the burden from programmers to write the simulations by hand, making the language easier to use. The advantage of the latter is that (i) this makes it easier to reason about the language (ii) it makes it easier to investigate links to other theories and (iii) it facilitates the implementation of the backend of a FASP solver.

In Section 5.2, we identify a core language for FASP that is sufficient to express FASP constructs and extensions. The FASP constructs that can be simulated are the following:

1. **Constraints.** One of the important constructs in ASP are constraint rules, which state that their body can never be true in a valid solution of the problem under consideration. In Section 5.3 we show that a well-known procedure for eliminating constraints in ASP can be generalized to the fuzzy case.

2. **Monotonically decreasing functions.** When generalizing ASP to a many-valued setting, various types of functions may serve as generalizations of logical connectives, ranging from t-norms and t-conorms to averaging operators, as well as problem-specific hedges. In the FASP framework introduced in
Chapter 3 we have therefore allowed arbitrary functions whose partial mappings are increasing or decreasing. It is easy to see that this class covers all commonly used operators from fuzzy logic. In Section 5.4, we show that any function with partial mappings that are decreasing can be simulated using increasing functions and negators.

Furthermore, we show that the following extensions of FASP can be simulated in our core language:

1. **Rule aggregation.** In the previous chapter we introduced an extension of FASP allowing partial rule satisfaction, called AFASP. In Section 5.5 we show how AFASP can be simulated using only rules that are required to be completely satisfied.

2. **S-implicators.** The AFASP framework introduced in the preceding chapter limits rules to correspond to residual implicators. However, there might still be some contexts in which an S-implicator is more natural than a residual implicator. This is further motivated in Section 5.6, where we show how to simulate rules based on S-implicators.

3. **Strong negation.** In ASP, two types of negation are used intertwiningly, resp. called negation-as-failure and strong negation. In Section 5.7 we show that the simulation of classical negation in classical ASP can be generalized to the fuzzy case.

## 5.2 The FASP Core Language

We introduce a core language for the FASP language we described in Chapter 3, called **core fuzzy answer set programming (CFASP)**, which will be shown to be sufficient to express many constructs and extensions of (A)FASP. CFASP is a subset of FASP with a more restricted syntax for rules: (i) constraints are removed from the language, hence each rule has an atom in its head; (ii) rule bodies only contain monotonically increasing functions and negators. The arguments of negators are also restricted to be atoms or values from a lattice, i.e. negators can not be applied to arbitrary expressions.
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

Definition 5.1. Consider a set of atoms \( A \). An extended literal is either an atom \( a \in A \), a value from a lattice \( \mathcal{L} \), or a naf-literal of the form \( \mathcal{N}(a) \), where \( \mathcal{N} \) corresponds to a negator.

Definition 5.2. Given a set of atoms \( A \), a CFASP rule on a complete lattice \( \mathcal{L} \) is a FASP rule of the form

\[
\begin{align*}
    r : a & \leftarrow f(b_1, \ldots, b_n) \\
\end{align*}
\]  

(5.1)

where \( a \) is an atom, \( f \) is an increasing \( \mathcal{L}^n \to \mathcal{L} \) function and \( b_i \) (\( 1 \leq i \leq n \)) are extended literals.

CFASP programs are defined as sets of CFASP rules.

Definition 5.3. A core FASP program (CFASP program) is a FASP program consisting of CFASP rules.

Note that by their definition in Section 3.2 on page 52, simple CFASP programs do not contain naf-literals.

5.3 Constraints

As mentioned in Section 2.2, in ASP there are special rules called constraints. Constraints differ from regular rules by the omission of a head literal and are used to specify that in any valid solution, the body of the rule should not be satisfied. For example, in program \( P_{gc} \) from Example 2.27 on page 33 the constraint \( \text{constr} \) specifies that two adjacent nodes should be differently colored. This is an important aspect of answer set programming and a necessary feature to elegantly describe many problem domains. In FASP, constraints are generalized by allowing rules of the following form:

\[
\text{fconstr} : l \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m)
\]

where \( l \) is an element of some complete lattice \( \mathcal{L} \) and \( f \) is a \( \mathcal{L}^{n+m} \to \mathcal{L} \) function that is increasing in its \( n \) first and decreasing in its \( m \) last arguments. Such a
5.3. CONSTRAINTS

constraint is satisfied by an interpretation $I$ when $l \geq I(f(b_1, \ldots, b_n; c_1, \ldots, c_m))$, i.e. when the truth value of the body is lower than $l$.

In this section we show how constraints in FASP programs on $([0, 1], \leq)$ can be simulated in CFASP and furthermore explain how they can be used to lock the truth value of an atom in a certain interval. To do so we extend CFASP with constraints and show that any extended program can be translated to an equivalent CFASP program. The extension is defined as follows:

**Definition 5.4.** Consider a set of atoms $A$. A CFASP $\bot$ rule on a complete lattice $\mathcal{L}$ is a FASP rule on $\mathcal{L}$ of the form

$$r : a \leftarrow f(b_1, \ldots, b_n)$$

where $f$ is an increasing $\mathcal{L}^n \rightarrow \mathcal{L}$ function, $a$ is either an atom or a value from $\mathcal{L}$ and $b_i$, for $1 \leq i \leq n$ is an extended literal.

**Definition 5.5.** A CFASP $\bot$ program on a complete lattice $\mathcal{L}$ is a FASP program consisting only of CFASP $\bot$ rules.

**5.3.1 Implementing Constraints**

The program $P_{\text{empty}} = \{c : p \leftarrow \text{not } p\}$\(^1\) is well-known in answer set programming because it has no classical answer sets, as shown in Example 2.26 on page 33. In fact, any program containing rule $c$ will have no answer sets $[5]$. This peculiarity actually turns out to be useful in eliminating answer sets under certain conditions. For example, consider the following classical program $P_{\text{nondet}}$:

$$r_1 : a \leftarrow \text{not } b$$
$$r_2 : b \leftarrow \text{not } a$$

which has answer sets $\{a\}$ and $\{b\}$, as shown in Example 2.25 on page 32. If we would like to eliminate the answer sets in which $b$ holds, we can add $b$ to the body

\(^1\)For convenience, we have taken the liberty of extending ASP with rule labels.
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

of rule $c$ and add the resulting rule to the program:

\[
\begin{align*}
  r_1: & \quad a \leftarrow \neg b \\
  r_2: & \quad b \leftarrow \neg a \\
  c_b: & \quad p \leftarrow \neg p, b
\end{align*}
\]

Suppose now that $A$ is an interpretation of $P_{\text{nondet}} \cup \{c_b\}$ such that $b \in A$. If $p \not\in A$, then $A$ is not a model of $P_{\text{nondet}} \cup \{c_b\}$ and thus $A$ is not an answer set. If $p \in A$, then rule $c_b$ is removed in $(P_{\text{nondet}} \cup \{c_b\})^A$. However, from this we can easily see that $A$ is not the least fixpoint of $\Pi_{(P_{\text{nondet}} \cup \{c_b\})^A}$, hence $A$ is not an answer set. This means that \{a\} is the only answer set of $P_{\text{nondet}} \cup \{c_b\}$ and the addition of rule $c_b$ effectively eliminated answer set \{b\}. Hence, adding the $c_b$ rule in $P_{\text{nondet}}$ has the same effect as adding the ASP constraint $\leftarrow b$. In fact, this works in general: any constraint $\leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m$ in an ASP program $P$ can be replaced by the ASP rule $p \leftarrow \neg p, b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m$ without changing the semantics of the program (provided that $p \not\in B_P$) [5].

The FASP program $F_{\text{empty}} = \{ c : p \leftarrow \mathcal{N}(p) \}$ corresponding to $P_{\text{empty}}$ does have answer sets, however, meaning that its useful capacity to eliminate undesired answer sets has not directly been preserved in the fuzzy setting. Using the Łukasiewicz negation for $\mathcal{N}$, for instance, it is not hard to see that \{p_{0.5}\} is the unique answer set. Therefore, an adaptation of program $F_{\text{empty}}$ is needed to eliminate undesirable answer sets in the fuzzy case. To this end, consider the following program $Z$:

\[
Z = \{ r : p \leftarrow \text{gt}(\mathcal{N}_M(p), 0) \}
\]

where rule $r$ is defined on $(]0, 1], \leq)$. The function in the body is defined as $\text{gt}(x, y) = 1$ if $x > y$ and $\text{gt}(x, y) = 0$ otherwise. It is easy to see that this function is increasing in $x$, so rule $r$ is a CFASP rule. One can easily see that the only models of $Z$ are $M_l = \{p^l\}$, $l \in ]0, 1]$. None of these models are answer sets, however, since for $l > 0$ we get

\[
Z^{M_l} = \{ r^{M_l} : p \leftarrow \text{gt}(0, 0) \}
\]

hence $\Pi^*_{Z^{M_l}} = \{p^0\} \neq M_l$.

It turns out that, similar to ASP, our program $Z$ can be used to simulate constraints in CFASP$^\perp$ programs. As an example, consider the CFASP$^\perp$ program $P$:

\[
\begin{align*}
  r_1: & \quad a \leftarrow \mathcal{N}_W(b) \\
  r_2: & \quad b \leftarrow \mathcal{N}_W(a) \\
  c: & \quad 0.5 \leftarrow a
\end{align*}
\]
5.3. CONSTRAINTS

The only answer sets of this program are of the form \( M_l = \{a^l, b^{1-l}\} \), with \( l \in [0,0.5] \), as rule \( c \) eliminates all solutions \( M \) where \( M(a) > 0.5 \). Now consider the CFASP program \( P' \):

\[
\begin{align*}
  r_1 & : \quad a \leftarrow \mathcal{N}_W(b) \\
  r_2 & : \quad b \leftarrow \mathcal{N}_W(a) \\
  r_c & : \quad \bot \leftarrow \mathcal{T}_M(\text{gt}(\mathcal{N}_W(\bot),0),\text{gt}(a,0.5))
\end{align*}
\]

with \( \bot \) a fresh atom. Note that \( P' \) is constraint-free and that for any \( l \in [0,0.5] \), \( M'_l = M_l \cup \{\bot\} \) is an answer set of \( P' \). Note that these are the only answer sets of \( P' \) as well. Indeed, suppose there is some fixpoint \( N \) of \( \Pi_{P'} \) such that \( N(a) > 0.5 \). It then follows that \( N(\bot) \) needs to satisfy

\[
N(\bot) = \Pi_{P'}(N)(\bot) = \sup\{N(r_b) \mid r \in P'_\bot\} = N((r_c)_b) = N(\mathcal{T}_M(\text{gt}(\mathcal{N}_W(\bot),0),\text{gt}(a,0.5))) = N(\text{gt}(\mathcal{N}_W(\bot),0))
\]

The equality \( N(\bot) = N(\text{gt}(\mathcal{N}_W(\bot),0)) \) has no solution, however, since we know that \( N(\text{gt}(\mathcal{N}_W(\bot),0)) \) takes on a value in \( \{0,1\} \) but for \( N(\bot) = 0 \) we get \( N(\text{gt}(\mathcal{N}_W(\bot),0)) = 1 \) and for \( N(\bot) = 1 \) we get \( N(\text{gt}(\mathcal{N}_W(\bot),0)) = 0 \). In general, one can verify that \( P \) and \( P' \) have corresponding answer sets, i.e. if \( M \) is an answer set of \( P \) then \( M \cup \{\bot\} \) is an answer set of \( P' \) and, conversely, if \( M' \) is an answer set of \( P' \), then \( M' \cap \mathcal{B}_P \) is an answer set of \( P \).

We now show that the construction used in the preceding example can be applied to arbitrary CFASP\(_\bot\) programs on the lattice \(([0,1], \leq)\). Formally, the general transformation is defined as follows:

**Definition 5.6.** Let \( P \) be a CFASP\(_\bot\) program on the lattice \(([0,1], \leq)\) and let \( \mathcal{C}_P \) be the set of constraint rules in \( P \). The corresponding CFASP program \( P' \) of \( P \) then contains the following rules:

\[
P' = \{r': a \leftarrow a \mid (r:a \leftarrow a) \in P \setminus \mathcal{C}_P\} \\
\cup \{r_c: \bot \leftarrow \mathcal{T}(\text{gt}(\mathcal{N}(\bot),0),\text{gt}(a,k)) \mid (c:k \leftarrow a) \in \mathcal{C}_P\}
\]

where \( \mathcal{T} \) is an arbitrary t-norm, \( \mathcal{N} \) is an arbitrary negator and \( \bot \notin \mathcal{B}_P \).
The following propositions show that the answer sets of the CFASP\( \perp \) program \( P \) and its corresponding CFASP program \( P' \) coincide.

**Proposition 5.7.** Let \( P \) be a CFASP\( \perp \) program and let \( P' \) be its corresponding CFASP program as defined by Definition 5.6. If \( M \) is an answer set of \( P \), then \( M' = M \cup \{ \perp^0 \} \) is an answer set of \( P' \).

**Proof.** Suppose that \( M \) is an answer set of \( P \). To show that \( M' = M \cup \{ \perp^0 \} \) is an answer set of \( P' \) we need to prove that 1. it is a model of \( P \); 2. that it is the least fixpoint of \( \Pi_{P'M'} \).

1. Since \( M \) is a model of \( P \) it follows by construction of \( P' \) that for each \( r \in P \setminus C_P \) and corresponding \( r' \in P' \) we have that \( M'(r') = M(r) = 1 \). For each constraint \( (r:a \leftarrow \alpha) \in C_P \) and corresponding rule \( c_r \in P' \) we have that \( M'(c_r) = 1 \) iff \( M'(\text{gt}(\text{gt}(N(\perp),0),\text{gt}(\alpha,k))) = 0 \). That this is true follows easily from the fact that \( M(r) = 1 \) and thus \( M(\alpha) \leq k \).

2. We show that (a) \( M' \) is a fixpoint of \( \Pi_{P'M'} \); (b) \( M' \) is the least fixpoint of \( \Pi_{P'M'} \).

(a) For \( a \in B_P \) we easily obtain that \( \Pi_{P'M'}(M')(a) = M'(a) \) using the definition of \( M' \) and \( P' \) and the fact that \( M \) is a fixpoint of \( \Pi_{P'M} \). Now, consider a rule \( c_r : \perp \leftarrow \text{gt}(\text{gt}(N(\perp),0),\text{gt}(\alpha,k)) \in P_{\perp} \). Since \( M \) is a model of \( P \) we know that for the corresponding rule \( (r:k \leftarrow a) \in P \) we have that \( M(a) \leq k \) and thus \( M'((c_r)_b) = 0 \). By definition of \( \Pi_{P'M'} \) this means \( \Pi_{P'M'}(M')(\perp) = 0 \). Hence, combining these two cases we obtain that \( M' \) is a fixpoint of \( \Pi_{P'M'} \).

(b) Suppose \( M' \neq \Pi_{P'M'}^* \). Then there is some \( M'' \subset M' \) such that \( M'' = \Pi_{P'M'}^* \). Consider then \( M''' = M'' \cap B_P \). From the definition of \( P' \), \( M' \) and \( M''' \) we can easily see that for each \( (r:a \leftarrow \alpha) \in P \setminus C_P \) and corresponding \( (r':a \leftarrow \alpha) \in P' \) we have that \( M(a) = M'(a) \) and \( M''(a) = M'''(a) \). Using the definition of \( \Pi_{P'M} \) and the reduct together with the fact that \( M'' \) is assumed to be a fixpoint of \( \Pi_{P'M} \) we then
5.3. CONSTRAINTS

obtain for any \( a \in B_P \) that:

\[
\Pi_{PM}(M''')(a) = \sup \{ M''(a) \mid (r : a \leftarrow \alpha) \in P^M \} = \sup \{ M''(a) \mid (r : a \leftarrow \alpha) \in P'^{M'} \} = M''(a) = M'''(a)
\]

Hence, \( M''' \) is a fixpoint of \( \Pi_{PM} \) and \( M''' \subset M \), which violates the assumption that \( M \) is an answer set of \( P \).

\[\square\]

**Lemma 5.8.** Let \( P \) be a CFASP\(^{-}\) program and let \( P' \) be its corresponding CFASP program as defined by Definition 5.6. If \( M' \) is an answer set of \( P' \) it holds that \( M'(\bot) = 0 \).

**Proof.** Since \( M' \) is an answer set of \( P' \), it must be a fixpoint of \( \Pi_{P'M'} \). By construction of \( P' \) we however know that if \( M' \) is a fixpoint of \( \Pi_{P'M'} \) it follows that \( M'(\bot) \in \{0,1\} \). Suppose \( M'(\bot) = 1 \). Then \( gt(N(M'(\bot)), 0) = gt(0,0) = 0 \). Hence for every \( r_c \in P'_\bot \) we obtain that \( M'((r_c)b) = 0 \). By definition of \( \Pi_{P'M'} \) it then follows that \( \Pi_{P'M'}(M'(\bot)) = 0 \neq M'(\bot) \), which is a contradiction. \[\square\]

**Proposition 5.9.** Let \( P \) be a CFASP\(^{-}\) program and let \( P' \) be its corresponding CFASP program as defined by Definition 5.6. If \( M' \) is an answer set of \( P' \), then \( M = M' \cap B_P \) is an answer set of \( P \).

**Proof.** Suppose \( M' \) is an answer set of \( P' \). To show that \( M = M' \cap B_P \) is an answer set of \( P \) we need to prove that (i) it is a model of \( P \); (ii) it is the least fixpoint of \( \Pi_{PM} \).

1. For any \( (r : a \leftarrow \alpha) \in P \setminus C_P \) we easily obtain by the fact that \( M'(r') = 1 \) for the corresponding \( (r' : a \leftarrow \alpha) \in P' \) and the construction of \( M \) that \( M(r) = 1 \). For \( (r : k \leftarrow \alpha) \in C_P \) we know by construction of \( P' \) and the fact that \( M' \) is a fixpoint of \( \Pi_{P'M'} \) that \( M'(\bot) \geq T(gt(N(M'(\bot)), 0), gt(M'(a), k)) \).

From Lemma 5.8 we however know that \( M'(\bot) = 0 \) and hence

\[
0 \geq T(gt(1,0), gt(M'(a), k)) = gt(M'(a), k)
\]

From the construction of \( M \) it then follows that \( M(r) = 1 \).
2. We show that (i) $M$ is a fixpoint of $\Pi_{PM}$; (ii) $M$ is the least fixpoint of $\Pi_{PM}$.

(a) For any $a \in B_P$ we easily obtain by the definition of $M$ and $P'$ that:

\[
\Pi_{PM}(M)(a) = \sup\{M(a) \mid (r:a \leftarrow a) \in P^M\}
= \sup\{M'(a) \mid (r':a \leftarrow a) \in P'^M\} \\
= \Pi_{PM'}(M')(a) \\
= M'(a) \\
= M(a)
\]

Hence $M$ is a fixpoint of $\Pi_{PM}$.

(b) Suppose $M \neq \Pi_{PM}'$. Then there is some $M'' \subset M$ such that $M'' = \Pi_{PM}'$. Consider then $M''' = M'' \cup \{\perp^0\}$. For $a \in B_P$ we easily obtain from the definition of $P'$ and the fact that $M''$ is a fixpoint of $\Pi_{PM}$ that $\Pi_{PM'}(M''')(a) = M'''(a)$. For $\perp$, by the fact that $M''' \subset M'$, we obtain the following:

\[
\Pi_{PM'}(M''')(\perp) \\
= \sup\{M'''(T(gt(N(M'(\perp)),0),gt(a^{M'},k))) \mid (r:k \leftarrow a) \in C_P\} \\
= \sup\{M'''(T(gt(N(0),0),gt(a^{M'},k))) \mid (r:k \leftarrow a) \in C_P\} \\
= \sup\{M'''(T(1,gt(a^{M'},k))) \mid (r:k \leftarrow a) \in C_P\} \\
= \sup\{M'(gt(a^{M'},k)) \mid (r:k \leftarrow a) \in C_P\} \\
\leq \sup\{M'(gt(a^{M'},k)) \mid (r:k \leftarrow a) \in C_P\} \\
= \Pi_{PM'}(M')(\perp) \\
= M'(\perp) \\
= 0
\]

Hence, $\Pi_{PM'}(M''')(\perp) = M'''(\perp)$ and thus $M'''$ is a fixpoint of $\Pi_{PM'}$, contradicting with our assumption that $M'$ is an answer set of $P'$.

\[\square\]
5.3. CONSTRAINTS

5.3.2 Locking the Truth Value

In some applications we might want to lock the truth value of an atom in a certain sub-interval of \([0, 1]\). We can define an extension of \(\text{CFASP}^{\perp}\), called \(\text{CFASP}^{\perp\perp}\), that supports this.

**Definition 5.10.** Consider a set of atoms \(A\). A \(\text{CFASP}^{\perp\perp}\) rule is either a \(\text{CFASP}^{\perp}\) rule on \(([0, 1], \leq)\) or a rule of the form

\[
\begin{align*}
  r : a \in [l, u] & \leftarrow \quad (5.3)
\end{align*}
\]

where \(a \in A\), \(l \in \mathcal{L}\) and \(u \in \mathcal{L}\). We call such a rule an interval-locking rule.

**Definition 5.11.** A \(\text{CFASP}^{\perp\perp}\) program \(P\) is a set of \(\text{CFASP}^{\perp\perp}\) rules. An interpretation \(I\) of \(P\) is said to satisfy an interval-locking rule of the form (5.4) above iff \(I(a) \in [l, u]\).

Now, it turns out that we can translate any \(\text{CFASP}^{\perp\perp}\) program to an equivalent \(\text{CFASP}^{\perp}\) program. First, note that a constraint introduces an upper bound on the value of a body function. We can use this to simulate interval-locking rules. Consider a \(\text{CFASP}^{\perp\perp}\) program \(P\) and an atom \(a \in B_P\) such that there is an interval-locking rule \(r : a \in [0.3, 0.8] \leftarrow \) in \(P\). If we wish to simulate \(r\), we can add the following rules to \(P\):

\[
\begin{align*}
  \text{constr} & : 0.8 \leftarrow a \\
  \text{constr}' & : 0.7 \leftarrow \mathcal{N}_W(a)
\end{align*}
\]

It is easy to see that any model \(M\) of \((P \setminus \{r\}) \cup \{\text{constr}, \text{constr}'\}\) satisfies \(0.8 \geq M(a)\) and \(0.3 \leq M(a)\), hence \(M(a) \in [0.3, 0.8]\). In general we can define the following transformation:

**Definition 5.12.** Let \(P\) be a \(\text{CFASP}^{\perp\perp}\) program over \(([0, 1], \leq)\) with \(\mathcal{I}\) the set of interval-locking rules in \(P\). Its corresponding \(\text{CFASP}^{\perp}\) program \(P'\) is defined as the following set of rules

\[
\begin{align*}
  P' & = (P \setminus \mathcal{I}) \cup \{\text{low}_a : (1 - l) \leftarrow \mathcal{N}_W(a) \mid (r : a \in [l, u] \leftarrow) \in \mathcal{I}\} \\
  & \quad \cup \{\text{upp}_a : u \leftarrow a \mid (r : a \in [l, u] \leftarrow) \in \mathcal{I}\}
\end{align*}
\]
The following proposition shows that the simulation of interval-locking rules works in general.

**Proposition 5.13.** Let $P$ be a CFASP\footnote{Definition 5.12} program over $([0, 1], \leq)$. Then $M$ is a model of $P$ iff $M$ is a model of its corresponding CFASP\footnote{Definition 5.12} program $P'$ defined in Definition 5.12.

**Proof.** We show this in two steps.

1. Suppose $M$ is a model of $P$. Then for each rule of the form $(r : a \in [l, u] \leftarrow) \in P$ we have that $M(a) \in [l, u]$. Hence $M(a) \geq l$ and thus $1 - M(a) \leq 1 - l$, meaning the corresponding rule $\text{low}_a \in P'$ is satisfied by $M$. Likewise we obtain that $M(a) \leq u$ and thus $M$ satisfies the rule $\text{upp}_a \in P'$. Since the other rules in $P'$ are equivalent to those in $P$ we obtain that $M$ is a model of $P'$.

2. Suppose $M'$ is a model of $P'$. Then by construction of $P'$ for each rule of the form $(r : a \in [l, u] \leftarrow) \in P$ there are two corresponding rules $\text{upp}_a$ and $\text{low}_a$ in $P'$. Since $M'$ is a model of $P'$ both $\text{low}_a$ and $\text{upp}_a$ are satisfied, meaning $M'(a) \leq u$ and $M'(a) \geq l$. By definition of interval-locking rules this means $M'$ satisfies the rule $(r : a \in [l, u] \leftarrow)$. Since the other rules in $P$ are equivalent to those in $P'$ we obtain that $M'$ is a model of $P$.

Note that, since answer sets are models, it follows trivially that answer sets also obey the interval-locking rules.

### 5.4 Monotonically Decreasing Functions

The FASP framework introduced in Chapter 3 not only allows functions that are increasing in rule bodies, but also functions with partial mappings that are monotonically decreasing. Functions with decreasing partial mappings in fact generalize negation-as-failure to functions with more than one argument: if $f(x_1, \ldots, x_n)$ decreases in its $i$th argument, the function increases when $x_i$ decreases. Since $x_i$ decreases when the maximal value that we can derive for $x_i$ decreases, this means that $f(x_1, \ldots, x_n)$ increases when the support for $x_i$ decreases. This corresponds to the idea underlying negation-as-failure. However, it turns out that generalizing negation-as-failure to functions with decreasing partial mappings does not lead to a
higher expressiveness. We show in this section that any program with monotonically decreasing functions can be translated to a program in which the only decreasing functions are negators that are applied to atoms, i.e. to CFASP programs. Similar to the previous section, we first define an extension of CFASP with decreasing functions and then show that any extended program can be translated to an equivalent CFASP program.

**Definition 5.14.** A CFASP \( f \) rule on a complete lattice \( L \) is a FASP rule of the form

\[
\rho : a \leftarrow f(b_1, \ldots, b_n; c_1, \ldots, c_m)
\]

(5.4)

where \( a, b_i \) and \( c_j \) are atoms and \( f \) is a \( L^{n+m} \rightarrow L \) function that is increasing in its \( n \) first and decreasing in its \( m \) last arguments. A CFASP \( f \) program on a complete lattice \( L \) is a set of CFASP \( f \) rules on \( L \).

Note that we only allow atoms, and not extended literals in CFASP \( f \) programs. This is not a problem, however, as it is easy to see that literals of the form \( \mathcal{N}(s) \) are \( L \rightarrow L \) functions that have no increasing arguments. We now show that any CFASP \( f \) program can be simulated using a CFASP program. Intuitively the procedure works as follows. Given a rule of the form (5.4) above, we replace the function \( f \) in its body with a new function \( f' \) defined by

\[
f'(b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m) = f(b_1, \ldots, b_n; \mathcal{N}(\text{not } c_1), \ldots, \mathcal{N}(\text{not } c_m)),
\]

where \( \text{not } c \) is a new atom supported by the rule \( n_c : \text{not } c \leftarrow \mathcal{N}(c) \), with \( \mathcal{N} \) an involutive negator. In this way, we have replaced the decreasing function with two CFASP rules. Formally, this procedure is defined as follows:

**Definition 5.15.** Let \( P \) be a CFASP \( f \) program over a lattice \( L \). Then its corresponding CFASP program \( P' \) contains the following rules:

\[
P' = \{ \rho' : a \leftarrow a' \mid (\rho : a \leftarrow a) \in P \} \cup \{ n_l : \text{not } l \leftarrow \mathcal{N}_i(l) \mid l \in \mathcal{F}_P \}
\]

where \( a = f(b_1, \ldots, b_n; c_1, \ldots, c_m) \), \( a' = f'(b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m) \), \( \mathcal{N}_i \) is an involutive negator on \( L \), \( \mathcal{F}_P \) is the set of all atoms \( a \) for which a naf-literal \( \mathcal{N}(a) \) occurs in the body of some rule in \( P \), and \( f' \) is defined by \( f'(b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m) = f(b_1, \ldots, b_n; \mathcal{N}_i(\text{not } c_1), \ldots, \mathcal{N}_i(\text{not } c_m)) \).
Also for each \( l \in B_P \) it must hold that \( \text{not}_l \notin B_P \), i.e. the \( \text{not}_l \) literal is a “fresh” literal.

As an example, consider a CFASP\(^f\) program \( P \) with the following rules:

\[
\begin{align*}
    r_1: & \quad a \leftarrow_m \frac{1}{1 + b \cdot c} \\
    r_2: & \quad b \leftarrow_m 0.8 \\
    r_3: & \quad c \leftarrow_m 0.5
\end{align*}
\]

An answer set of \( P \) is \( M = \{a^{10/14}, b^{0.8}, c^{0.5}\} \). If we apply the transformation of Definition 5.15 on \( P \) we obtain \( P' \) with rules:

\[
\begin{align*}
    r'_1: & \quad a \leftarrow_m f'(\text{not}_b, \text{not}_c) \\
    r'_2: & \quad b \leftarrow_m 0.8 \\
    r'_3: & \quad c \leftarrow_m 0.5 \\
    n_b: & \quad \text{not}_b \leftarrow_m \mathcal{N}_W(b) \\
    n_c: & \quad \text{not}_c \leftarrow_m \mathcal{N}_W(c)
\end{align*}
\]

where \( f'(\text{not}_b, \text{not}_c) = \frac{1}{1 + (\mathcal{N}_W(\text{not}_b)) \cdot (\mathcal{N}_W(\text{not}_c))} \). We can then show that \( M' = M \cup \{\text{not}_b^{\mathcal{N}_W(M(b))}, \text{not}_c^{\mathcal{N}_W(M(c))}\} \) is an answer set of \( P' \). The reduct \( P'_M \) contains the following rules:

\[
\begin{align*}
    r'_1 M': & \quad a \leftarrow_m f'(\text{not}_b, \text{not}_c) \\
    r'_2 M': & \quad b \leftarrow_m 0.8 \\
    r'_3 M': & \quad c \leftarrow_m 0.5 \\
    n'_b M': & \quad \text{not}_b \leftarrow_m 0.2 \\
    n'_c M': & \quad \text{not}_c \leftarrow_m 0.5
\end{align*}
\]

where \( f' \) is defined as above. From the reduct we can see that \( \Pi^*_{p_{p'M'}}(b) = 0.8, \Pi^*_{p_{p'M'}}(c) = 0.5, \Pi^*_{p_{p'M'}}(\text{not}_b) = 0.2 \) and \( \Pi^*_{p_{p'M'}}(\text{not}_c) = 0.5 \). This leads to \( \Pi^*_{p_{p'M'}}(a) = 10/14 \) and thus \( M' \) is the least fixpoint of \( \Pi_{p_{p'M'}} \), which is what we expected.

The following propositions show that this transformation preserves the answer set semantics.
5.4. MONOTONICALLY DECREASING FUNCTIONS

**Proposition 5.16.** Let $P$ be a CFASP$^r$ program and let $P'$ be its corresponding CFASP program as defined by **Definition 5.15.** If $M$ is an answer set of $P$, then $M' = M \cup \{not_1^SN_i(M(l)) \mid l \in F_P\}$ is an answer set of $P'$.

**Proof.** We have to show that $M'$ is the least fixpoint of $\Pi_{p_{M'}}$. First, note that for any $l \in F_P$ by definition of $P'$ and $M'$

$$\Pi_{p_{M'}}(M')(not_1) = N(M'(l)) = N(M(l)) = M'(not_1) \quad (5.5)$$

Now, for $a \in B_P$ each rule $r:a \leftarrow f(b_1,\ldots,b_n;c_1,\ldots,c_m)$ in $P$ is replaced by $r:a \leftarrow f'(b_1,\ldots,b_n,not_{c_1},\ldots,not_{c_m})$, with $f'$ as in **Definition 5.15.** Hence since $M'(not_1) = N_i(M(l))$ we obtain that

$$M(f(b_1,\ldots,b_n;c_1,\ldots,c_m)^{M}) = M'(f'(b_1,\ldots,b_n,not_{c_1},\ldots,not_{c_m})^{M'}) \quad (5.6)$$

and also

$$M(r) = M'(r') \quad (5.7)$$

since $N_i$ is involutive, $M = M' \cap B_P$ and the atoms occurring in $R_P$ are all atoms of $B_P$. As $M$ is a fixpoint of $\Pi_{p_M}$, it then follows from (5.5), (5.6), (5.7) and by definition of $M'$ that $M'$ must be a fixpoint of $\Pi_{p_{M'}}$ (one only has to work out the definition of $\Pi_{p_{M'}}$ together with the formerly mentioned equations to see this).

Suppose now there is some $M'' < M'$ that is also a fixpoint of $\Pi_{p_{M'}}$. By definition of $P'$ it is easy to see that for each $l \in F_P$ it then must hold that $M''(not_1) = N_i(M'(l)) = N_i(M(l))$. From this it follows that for each $r \in P$ we obtain $(M'' \cap B_P)(r_b^M) = M''(r_b')$ by definition of $P'$ and thus we obtain for each $a \in B_P$:

$$\Pi_{p_M}(M'' \cap B_P)(a) = \sup\{M'' \cap B_P(a^M) \mid (r:a \leftarrow a) \in P\}
= \sup\{M''(a') \mid r':a \leftarrow a' \in P_{r_{M'}}\}
= \Pi_{p_{M'}}(M')(a)
= M''(a)$$

Hence $M'' \cap B_P$ is a fixpoint of $\Pi_{p_M}$. However, since for each $l \in F_P$ we have $M''(not_1) = M'(not_1)$ and as $M'' < M'$, it holds that $M'' \cap B_P < M$. This however contradicts the fact that $M$ is the least fixpoint of $\Pi_{p_M}$, showing no such $M''$ can exist and thus $M'$ is the least fixpoint of $\Pi_{p_{M'}}$. 

\[\square\]
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

Proposition 5.17. Let $P$ be a CFASP\textsuperscript{f} program and let $P'$ be its corresponding CFASP program as defined by Definition 5.15. If $M'$ is an answer set of $P'$, then $M = M' \cap \mathcal{B}_P$ is an answer set of $P$.

Proof. By definition of $P'$ it is not hard to see that for each rule $r \in P$ we have

$$M((r_b^M)) = M'(r_b')$$

(5.8)

We show that $M$ is the least fixpoint of $\Pi_{pm}$. From (5.8) we can easily see that $M$ must be a fixpoint of $\Pi_{pm}$. Now suppose some $M''$ (with $M'' < M$) is also a fixpoint of $\Pi_{pm}$. Then we can construct $M''' = M'' \cup \{\text{not}_I^{N_i(M(i))} \mid I \in \mathcal{F}_P\}$. It is not hard to see that for each rule $r \in P$:

$$M''((r_b)^M) = M'''(r_b')$$

Hence, as $M''$ is a fixpoint of $\Pi_{pm}$, we obtain that $M'''$ is also a fixpoint of $\Pi_{pm'}$, contradicting the fact that $M'$ is an answer set of $P'$.

Note that when we combine the results introduced in this section with those from Section 5.3, we find that any FASP program can be translated to an equivalent CFASP program. In the following sections we show that for certain extensions of FASP, translations to CFASP can also be defined.

5.5 Aggregators

In Chapter 4 we introduced an extension of FASP that allowed rules to be partially fulfilled, called AFASP. At first sight, one might be tempted to think that an AFASP program $P$ with an aggregator over $\mathcal{L}_P$ can be replaced by a CFASP\textsuperscript{f} program $P'$ such that $P' = \mathcal{R}_P \cup \{r_{aggr}^{aggr} : aggr \leftarrow f(I_{r_1}((r_1)_b,(r_1)_h),\ldots,I_{r_n}((r_n)_b,(r_n)_h))\}$, where $f$ corresponds to the function defined by the aggregator of $P$. The intended meaning is such that $M$ is an m-answer set of $P$ iff $M \cup \{aggr^m\}$ is an answer set of $P'$.

This trivial translation is not correct, however, as it does not correctly incorporate the notion of partial rule satisfaction. For example, consider the AFASP program $P$ with rule base $\mathcal{R}_P = \{(r_1: a \leftarrow 1),(r_2: b \leftarrow 1)\}$ and with aggregator $A_P(\rho) = \inf\{\rho(r) \mid r \in \mathcal{R}_P\}$. Using the transformation proposed above, we obtain the CFASP\textsuperscript{f} program $P' = \{(r_1': a \leftarrow 1),(r_2': b \leftarrow 1),(r_{aggr}': aggr \leftarrow \}$
inf(\mathcal{I}(1,a), \mathcal{I}(1,b)))$. For the 0.7-answer set $M = \{a^{0.7}, b^1\}$ of $P$ the corresponding interpretation $M' = M \cup \{ (r'_{\text{aggr}})^{0.7} \}$ is not an answer set of $P'$, however, as $P'$ only has one answer set, viz. $\{a^1, b^1, (r'_{\text{aggr}})^1\}$. This problem can be solved using the following alternate simulation:

**Definition 5.18.** Let $P$ be an AFASP program with rule base $\mathcal{R}_P = \{ r_1, \ldots, r_n \}$ on the lattice $\mathcal{L}$. If there is an involutive negator $\mathcal{N}_i$ on $\mathcal{L}$ we can construct a CFASP\textsuperscript{f} program corresponding to $P$, denoted as $P'$, that contains the following rules:

$$
P' = \{ r': a \leftarrow T_r(a,r'_1) \mid (r:a) \in \mathcal{R}_P, a \in B_P \}
\cup \{ r'_a: \text{not}_a \leftarrow \mathcal{N}_i(a) \mid a \in B_P \}
\cup \{ r'_i: I_r(a', \mathcal{N}_i(\text{not}_a)) \mid (r:a) \in \mathcal{R}_P, a \in B_P \}
\cup \{ r'_i: I_r(a', k) \mid (r:k) \in \mathcal{R}_P, k \in \mathcal{L} \}
\cup \{ r'_{\text{aggr}}: \text{aggr} \leftarrow f(r'_1, \ldots, r'_{ni}) \}
$$

Furthermore, none of the atoms $\text{not}_a$ for $a \in B_P$, $r'_i$ for $r \in \mathcal{R}_P$ and $\text{aggr}$ occur in $B_P$. Also, the function $f$ corresponds to $A_P$ in the sense that $A_P(\rho) = f(\rho(r_1), \ldots, \rho(r_n))$ and for a given $\alpha = \mathcal{g}(b_1, \ldots, b_n; c_1, \ldots, c_m)$ we have that $\alpha' = \mathcal{g}'(b_1, \ldots, b_n, \text{not}_{c_1}, \ldots, \text{not}_{c_m})$, with $\mathcal{g}'$ defined as $\mathcal{g}'(b_1, \ldots, b_n, \text{not}_{c_1}, \ldots, \text{not}_{c_m}) = \mathcal{g}(b_1, \ldots, b_n, \mathcal{N}_i(\text{not}_{c_1}), \ldots, \mathcal{N}_i(\text{not}_{c_m}))$.

Note that due to the $r'_\rho$ rules in Definition 5.18, we translate an AFASP program to a CFASP\textsuperscript{f} program, rather than a CFASP program. This is not a problem however, as we have shown in Section 5.4 that any CFASP\textsuperscript{f} program can be translated to a corresponding CFASP program. We summarized this in Figure 5.1 on page 156.

Now consider the program $P$ from above again. If we use the translation defined in Definition 5.18 we obtain

$$
P' = \{(r'_1: a \leftarrow T(1,r'_{1i})), (r'_2: b \leftarrow T(1,r'_{2i}))\}
\cup \{(r'_a: \text{not}_a \leftarrow \mathcal{N}_W(a)), (r'_b: \text{not}_b \leftarrow \mathcal{N}_W(b))\}
\cup \{(r'_{1p}: r'_{1i} \leftarrow I(1, \mathcal{N}_W(\text{not}_a)))(r'_{2p}: r'_{2i} \leftarrow I(1, \mathcal{N}_W(\text{not}_b)))\}
\cup \{(r'_{\text{aggr}}: \text{aggr} \leftarrow \text{inf}(r'_{1i}, r'_{2i}))\}
$$
One can easily verify that $M' = \{a^{0.7}, b^1, \text{not}_{a}^{0.3}, \text{not}_{b}^{0.7}, r_{1i}^{1}, r_{2i}^{1}, \text{aggr}^{0.7}\}$ is an answer set of $P'$, hence the translation in Definition 5.18 preserves the AFASP semantics.

**Example 5.19.** Consider $P$ with $A_{P}(\rho) = \inf\{\rho(r) \mid r \in R_{P}\}$ and $R_{P}$:

\[
\begin{align*}
r_1 & : a \leftarrow_{m} N_{W}(b) \\
r_2 & : b \leftarrow_{m} 0.7
\end{align*}
\]

Applying Definition 5.18, we obtain the CFASP$^{f}$ program $P'$ with rules:

\[
\begin{align*}
r_1' & : a \leftarrow_{m} T_{M}(N_{W}(b), r_{1i}) \\
r_2' & : b \leftarrow_{m} T_{M}(0.7, r_{2i}) \\
r_{a} & : \text{not}_{a} \leftarrow_{m} N_{W}(a) \\
r_{b} & : \text{not}_{b} \leftarrow_{m} N_{W}(b) \\
r_{1p} & : r_{1i} \leftarrow_{m} I_{M}(N_{W}(\text{not}_{b})), N_{W}(\text{not}_{a})) \\
r_{2p} & : r_{2i} \leftarrow_{m} I_{M}(0.7, N_{W}(\text{not}_{b})) \\
r_{\text{aggr}} & : \text{aggr} \leftarrow_{m} \inf(r_{1i}', r_{2i}')
\end{align*}
\]

Consider now the 1-answer set $M = \{a^{0.3}, b^{0.7}\}$ of $P$. Definition 5.18 is constructed in such a way that $M' = \{a^{0.3}, b^{0.7}, r_{1i}'^{1}, r_{2i}'^{1}, \text{not}_{a}^{0.7}, \text{not}_{b}^{0.3}, \text{aggr}^{1}\}$ is an answer set of $P'$.

The construction with the involutive negator and $\text{not}_{a}$ for any $a \in B_{P}$ is needed to correctly preserve the semantics. To see this, consider the following alternative CFASP$^{f}$ translation $P''$ of program $P$ in Example 5.19:

\[
\begin{align*}
r_1' & : a \leftarrow_{m} T_{M}(N_{W}(b), r_{1i}) \\
r_2' & : b \leftarrow_{m} T_{M}(0.7, r_{2i}) \\
r_{1p} & : r_{1i} \leftarrow_{m} I_{M}(N_{W}(b), a) \\
r_{2p} & : r_{2i} \leftarrow_{m} I_{M}(0.7, b) \\
r_{\text{aggr}} & : \text{aggr} \leftarrow_{m} \inf(r_{1i}', r_{2i}')
\end{align*}
\]

Now consider the 1-answer set $M = \{a^{0.3}, b^{0.7}\}$ of $P_1$. We wish the aggregator-free version of $P$ to be constructed in such a way that $M' = M \cup \{r_{1i}^{A_{P}(\rho)} \mid r \in R_{P}\} \cup \{\text{aggr}^{A_{P}(\rho_{M})}\}$ is an answer set of $P''$. Hence in the case of $P''$, we find that $M' = M \cup \{r_{1i}'^{1}, r_{2i}'^{1}\} \cup \{\text{aggr}^{1}\}$ should be an answer set of $P''$. However, there is an $M'' < M'$ such that $M''$ is a fixpoint of $\Pi_{P''M''}$, which contradicts the fact that
5.5. AGGREGATORS

$M'$ is an answer set of $P''$. Indeed, for $M'' = \{a^{0.2}, b^{0.7}\} \cup \{r'_{1i}^{0.2}, r'_{2i}^{1}\} \cup \{\text{aggr}^{0.2}\}$ it can be seen that $M''$ is a model of $P''$ and a fixpoint of $\Pi_{p\rightarrow M'}$ as follows. For $a$ we obtain:

$$\Pi_{p\rightarrow M'}(M'')(a) = T_M(N_W(M'(b)), M''(r'_{1i})) = T_M(1 - 0.7, 0.2) = 0.2$$

Likewise we obtain that $\Pi_{p\rightarrow M'}(M'')(b) = 0.7$. Now for $r'_{1i}$ we obtain

$$\Pi_{p\rightarrow M'}(M'')(r'_{1i}) = M''((I_M(N_W(b), a))^{M'}) = M''(I_M(N_W(b), a)) = I_M(0.3, 0.2) = 0.2$$

Likewise we obtain that $\Pi_{p\rightarrow M'}(M'')(r'_{2i}) = 1$. Last, for $\text{aggr}$ we obtain

$$\Pi_{p\rightarrow M'}(M'')(\text{aggr}) = M''(\inf(r'_{1i}, r'_{2i})) = \inf(0.2, 1) = 0.2$$

Hence $M''$ is a fixpoint of $\Pi_{p\rightarrow M'}$, contradicting that $M'$ is an answer set of $P'$.

The problem the preceding example illustrates is that we must be able to “fix” the value of the literals $r'_{1i}$ and $r'_{2i}$ when we are taking the reduct relative to $M'$. The only way to ensure this is by eliminating all literals from the body of the $r'_{1\rho}$ and $r'_{2\rho}$ rules by means of the reduct procedure. Hence we must replace each positively occurring literal in the bodies of $r'_{1\rho}$ and $r'_{2\rho}$ by a negatively occurring literal. This is done by replacing a positively occurring literal $a$ with $N_W(\text{not}_a)$, which preserves the same value, but will be replaced by the reduct operation.

The following propositions show that our translation preserves the semantics.

**Proposition 5.20.** Let $P$ be an AFASP program with $R_P = \{r_1, \ldots, r_n\}$ and let $P'$ be its corresponding CFASP$^f$ program as defined in Definition 5.18. If $M$ is an $m$-answer set of $P$, then $M' = M \cup \{\text{aggr}^{A_P(r)} \cup \{\text{not}_a^{M(a)} \mid a \in B_P \} \cup \{r^{M(r)} \mid r \in R_P\}$ is an answer set of $P'$.

**Proof.** It should be clear from the definition of $M'$ and $P'$ that $M'$ is a model of $P'$. Hence, we only need to show that $M'$ is the least fixpoint of $\Pi_{p\rightarrow M'}$. First we show that it is a fixpoint of $\Pi_{p\rightarrow M'}$. We consider five cases:
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

1. For \( \text{aggr} \in B_{P'} \) we obtain by definition of \( P' \) and \( M' \) that
\[
\Pi_{P', M'}(M'(\text{aggr})) = M'(f(r'_{1_i}, \ldots, r'_{n_i})) = A_P(\rho_M) = M'\text{(aggr)}
\]

2. For \( a \in B_P \) and corresponding \( \text{not}_a \in B_{P'} \) we obtain using the definition of \( M' \) that
\[
\Pi_{P', M'}(M'(\text{not}_a)) = M'(N_i(M(a))) = M'(\text{not}_a)
\]

3. For \( (r : a \leftarrow a) \in R_P \) with \( a \in B_P \) and corresponding \( r'_i \in B_{P'} \) we obtain using the definition of \( M' \) and the definition of \( P' \) that
\[
\Pi_{P', M'}(M'(r'_i)) = M'(I_r(a', N_i(\text{not}_a))^{M'})
\]
\[
= I_r(M(a'), M'(N_i(M(a))))
\]
\[
= I_r(M(a), M(a))
\]
\[
= M(r)
\]
\[
= M'(r'_i)
\]

4. For \( (k : a \leftarrow a) \in R_P \) with \( k \in L \) we obtain that \( \Pi_{P', M'}(M'(r'_i)) = M'(r'_i) \) similar to the previous case.

5. For \( a \in B_P \) we obtain, using the definition of \( P' \), the fact that for any expression \( a \) and interpretation of \( a \) we have \( I(a^I) = I(a) \), the definition of the reduct of a program and the fact that \( M \) is a fixpoint of \( \Pi_{P, M} \) that
\[
\Pi_{P', M'}(M'(a)) = \sup\{T_r(M'(a)^{M'}), M'(r'_i)) \mid (r : a \leftarrow a) \in P]\}
\[
= \sup\{T_r(M(a^M), \rho_M(r)) \mid (r : a \leftarrow a) \in P\}
\]
\[
= \sup\{T_r(M(a^R), \rho_M(r)) \mid (r : a \leftarrow a) \in P\}
\]
\[
= \sup\{T_r(M(a), \rho_M(r)) \mid (r : a \leftarrow a) \in P^M\}
\]
\[
= \sup\{M_{s}(r, \rho_M(r)) \mid (r : a \leftarrow a) \in P^M\}
\]
\[
= M(a)
\]
\[
= M'(a)
\]
5.5. AGGREGATORS

Hence we can conclude that $M'$ is a fixpoint of $\Pi_{PM'}$. Now suppose there is an interpretation $M'' < M'$ of $P'$ such that $M''$ is also a fixpoint of $\Pi_{PM'}$. For $a \in B_P$ we then obtain by definition of $P'$ that

$$
\Pi_{PM'}(M'')(\neg a) = M''(\langle N_i(a) \rangle^{M'}) = M''(N_i(M' (a))) = N_i(M' (a))
$$

Hence $M''(\neg a) = M'(\neg a)$ for each $a \in B_P$. For $(r : a \leftarrow a) \in R_P$ with $a \in B_P$ we obtain by definition of $P'$, the fact that in $a'$ there are no naf-literals and the fact that implicators are increasing in their first and decreasing in their second argument that

$$
\Pi_{PM'}(M'')(r'_i) = M''(\langle \mathcal{I}_r(a',N_i(\neg a)) \rangle^{M'}) = M''(\mathcal{I}_r(M'(a'),N_i(M'(\neg a)))) = M''(\mathcal{I}_r(M(a),N_i(N_i(M(a)))) = M''(\mathcal{I}_r(M(a),M(a))) = M(r) = M'(r'_i)
$$

Similarly we obtain for any $(r : k \leftarrow a) \in P$ with $k \in \mathcal{L}_P$ that $\Pi_{PM'}(M'')(r'_i) = M'(r'_i)$. From this and the definition of $P'$ it then also easily follows that $M''(\text{aggr}) = M'(\text{aggr})$. Hence as $M'' < M'$, from the foregoing it follows that $M'' \cap B_P < M$. Now for each $a \in B_P$ we can show that

$$
\Pi_{PM,PM}(M'')(a) = \sup \{ \mathcal{T}_r(M''(a^M),\rho_M(r)) \mid (r : a \leftarrow a) \in R_P \} = \sup \{ \mathcal{T}_r(M''(a^M),\rho_M(r)) \mid (r : a \leftarrow a) \in R_P \} = \sup \{ \mathcal{T}_r(M''(a^M),M''(r'_i)) \mid (r : a \leftarrow a) \in R_P \} = \sup \{ M''_s(r'_i,\rho_M(r'_i)) \mid (r : a \leftarrow a) \in R_P \} = \Pi_{PM'}(M'')(r'_i) = M''(a)
$$

meaning $M'' \cap B_P$ is a fixpoint of $\Pi_{PM}$, contradicting the fact that $M$ is the least fixpoint of $\Pi_{PM}$. Hence such an $M''$ cannot exist and $M'$ is the least fixpoint of $\Pi_{PM'}$. 

\[\square\]
Proposition 5.21. Let $P$ be an AFASP program and let $P'$ be its corresponding CFASP$^f$ program as defined in Definition 5.18. If $M'$ is an answer set of $P'$, with $m = M'(aggr)$, then $M' \cap B_P$ is an $m$-answer set of $P$.

Proof. We have to show that for $M = M' \cap B_P$, we have $A_P(\rho_M) \geq m$ for any $m \leq M'(aggr)$ and that $M$ is the least fixpoint of $\Pi_{PM,PM}$. First, we show that $A_P(\rho_M) \geq m$ for any $m \leq M'(aggr)$. Suppose $m \in \mathcal{L}_P$ such that $m \leq M'(aggr)$. Since $M'$ is a fixpoint of $\Pi_{PM'}$ and from the definition of $P'$ we can easily see that for any $a \in B_P$ we must have $M'(not_a) = N_i(M'(a)) = N_i(M(a))$ as there is only one rule with $not_a$ in the head and likewise that for any $(r:a \leftarrow a) \in \mathcal{P}_P$ with $a \in B_P$ and corresponding $r'_i \in B_{P'}$ we must have $M'(r'_i) = M'(\mathcal{I}_P(a,N_i(not_a))) = \rho_M(r)$. Similarly we obtain for any $(r:k \leftarrow a) \in \mathcal{P}_P$ with $k \in \mathcal{L}_P$ and corresponding $r'_i \in B_{P'}$ that $M'(r'_i) = \rho_M(r)$. Furthermore it must follow that $M'(aggr) = M'(f(r'_1,\ldots,r'_{ni}))$ as again there is only one rule with $\rho_{aggr}$ in the head and thus as $M'(f(r'_1,\ldots,r'_{ni})) = A_P(\rho_M)$ by construction of $P'$ that $A_P(\rho_M) \geq m$ as $M'(\rho_{aggr}) \geq m$.

Second we show that $M$ is the least fixpoint of $\Pi_{PM,PM}$. First, we show that $M$ is a fixpoint of $\Pi_{PM,PM}$. Suppose $a \in B_P$, then from the fact that $M'$ is a fixpoint of $\Pi_{PM'}$, the definition of $M$, the fact that for any expression $a$ we have $M(a^M) = M(a)$ and the foregoing part of the proof we know that:

$$M'(a) = \Pi_{PM'}(M')(a)$$

$$= \sup\{M'((\mathcal{T}_r(a,r'_i))^M) \mid (r:a \leftarrow a) \in \mathcal{R}_P\}$$

$$= \sup\{\mathcal{T}_r(M'(a^M'), M'(r'_i)) \mid (r:a \leftarrow a) \in \mathcal{R}_P\}$$

$$= \sup\{\mathcal{T}_r(M(a^M), \rho_M(r)) \mid (r:a \leftarrow a) \in \mathcal{R}_P\}$$

$$= \sup\{\mathcal{T}_r(M(a^M), \rho_M(r^M)) \mid (r:a \leftarrow a) \in \mathcal{R}_P\}$$

$$= \sup\{M(r, \rho_M(r)) \mid (r:a \leftarrow a) \in (\mathcal{R}_P)^M\}$$

$$= \Pi_{PM,PM}(M)(a)$$

Hence $M$ is a fixpoint of $\Pi_{PM,PM}$. Now, suppose there is some $M'' < M$ such that $M''$ is also a fixpoint of $\Pi_{PM,PM}$. Consider then $M'' = M'' \cup \{aggr^{M'(aggr)}\} \cup \{r_i^{PM}(r) \mid r_i \in \mathcal{R}_P\}$. Obviously $M'' < M'$ by construction. We show that $M''$ is a fixpoint of $\Pi_{PM'}$ contradicting the assumption that $M'$ is an answer set of $P'$. 

138
5.5. AGGREGATORS

1. For \( a \in B_P \) and the corresponding \( \text{not} a \in B_{P'} \) we obtain

\[
\Pi_{pM'}(M'''(\text{not} a)) = M'''((N_{i}(a))^{M'})
\]
\[
= M'''(N_{i}(M'(a)))
\]
\[
= N_{i}(M'(a))
\]
\[
= M'''(\text{not} a)
\]

2. For \((r : a \leftarrow a) \in R_P \) and the corresponding \( r_i' \) we obtain

\[
\Pi_{pM'}(M'''(r_i')) = M'''((I_r(\alpha', N_i(\text{not} a)))^{M'})
\]
\[
= M'''(I_r(M'(a'), N_i(N_i((M'(a))))))
\]
\[
= I_r(M'(a'), M'(a))
\]
\[
= I_r(M(a), M(a))
\]
\[
= I_r(M(r)
\]
\[
= M'''(r_i')
\]

3. For \((r : k \leftarrow a) \in R_P \) and the corresponding \( r_i' \) we obtain similar to the above that \( \Pi_{pM'}(M'''(r_i')) = M'''(r_i'). \)

4. For \( \text{aggr} \) we obtain

\[
\Pi_{pM'}(M'''(\text{aggr})) = M'''(f(r_1', \ldots, r_n'))
\]
\[
= f(\rho_M(r_1), \ldots, \rho_M(r_n))
\]
\[
= \lambda_p(\rho_M)
\]
\[
= M'(\text{aggr})
\]
\[
= M'''(\text{aggr})
\]

5. Suppose \( a \in B_P \). Since \( M'' \) is a fixpoint of \( \Pi_{pM',\rho_M} \) and \( M'' = M''' \cap B_P \) it
follows that $M''' \cap B_P$ is a fixpoint of $\Pi_{PM,\rho_M}$. From this we obtain:

$$\Pi_{pM'}(M''') (a) = \sup \{ M''' ((T_r(a, r_i'))^M') \mid (r : a \leftarrow \alpha) \in R_P \}$$

$$= \sup \{ T_r(M'''(a^M'), M'''(r_i')) \mid (r : a \leftarrow \alpha) \in R_P \}$$

$$= \sup \{ T_r(M'''(a^M), \rho_M(r)) \mid (r : a \leftarrow \alpha) \in R_P \}$$

$$= \sup \{ T_r(M', \rho_M(r)) \mid (r : a \leftarrow \alpha) \in R_P \}$$

$$= \Pi_{pM,\rho_M} (M''') (a)$$

$$= M''' (a)$$

Hence $M'''$ is a fixpoint of $\Pi_{pM'}$. This is however impossible as $M'$ is an answer set of $P'$ and thus the least fixpoint of $\Pi_{pM'}$. Thus no such $M''$ can exist and $M$ is the least fixpoint of $\Pi_{pM}$.

5.6 S-implicators

Occasionally, when using aggregators, the ability to use other types of implicators in rules could be useful. If the Kleene-Dienes implicator were used, for example, then $r : a \leftarrow f(b_1, \ldots, b_n)$ would only evaluate to 1 if either the body evaluates to 0, or the head evaluates to 1. This means that as soon as $I(f(b_1, \ldots, b_n)) > 0$, the rule is triggered and the head is taken to be completely true.

Example 5.22. Consider the following program $P_{bbq}$, encoding that we want to have a barbecue, unless the weather is bad:

$$r_1 : \text{bad\_weather} \leftarrow_{kd} \text{rain}$$

$$r_2 : \text{bad\_weather} \leftarrow_{l} \mathcal{N}(\text{sunshine})$$

$$r_3 : \text{bbq} \leftarrow_{l} \mathcal{N}(\text{bad\_weather})$$

where $\text{rain}$ is the degree to which it is raining and $\text{sunshine}$ is the expected amount of sunshine. Because the Kleene-Dienes implicator is used in the first rule, a barbecue is out of the question even if it rains only a little bit (e.g. drizzle). If it is not raining, our motivation for having a barbecue depends linearly on the amount of sunshine, hence a Łukasiewicz implicator is used.
5.6. S-IMPLICATORS

However, as the rules in AFASP are restricted to residual implicators, we are not directly able to write the program above. In this section we explain how the semantics of AFASP can be extended to support S-implicators and moreover show that any AFASP program with S-implicators can be simulated by a normal AFASP program. Similar to the previous sections, we first define an extension of AFASP with S-implicator rules, called AFASP$^S$ and then show that any program in AFASP$^S$ can be reduced to an equivalent AFASP program.

**Definition 5.23.** An AFASP$^S$ rule is a FASP rule that is associated with either a residual implicator or an S-implicator constructed from a t-norm and an involutive negator. An AFASP$^S$ program is a tuple $\langle R, A \rangle$ where $R$ is a FASP program with AFASP$^S$ rules and $A$ is an aggregator function over $R$.

For an AFASP$^S$ program $P$ we define the Herbrand Base, the set $P_a$ for a given atom $a$, the rule base $R_P$, the aggregator $A_P$, the immediate consequence operator, the reduct and interpretations similar to AFASP. For a given AFASP$^S$ program $P$, $w \in L_P$ and rule $r \in P$ that is associated with an S-implicator, i.e. for any AFASP$^S$ program $P$, $w \in L_P$ and rule $r \in P$ that is associated with an S-implicator we define $I_s(r, w) = \inf\{k \in L_P \mid I_r(I(a), k) \geq w\}$. The following lemma and proposition shows how this support can be computed in an easy manner.

**Lemma 5.24.** Let $L$ be a lattice, $N_i$ an involutive negator on $L$ and $T$ a t-norm on $L$. Then for any $x, y, w \in L$ it holds that$^3$:

$$I_{T, N_i}(x, y) \geq w \equiv y \geq N_i(I_T(x, N_i(w))) \geq 1$$

$^2$See Definition 4.8 on page 78.
$^3$Recall that $I_{T, N_i}$ is the S-implicator constructed using the t-norm $T$ and the negator $N_i$, as defined in Definition 2.48 on page 45.
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

Proof. Using the definition of $I_{T,N_i}$ and the residuation principle we obtain:

\[
I_{T,N_i}(x,y) \geq w \equiv N_i(T(x,N_i(y))) \geq w \\
\equiv T(x,N_i(y)) \leq N_i(w) \\
\equiv N_i(y) \leq I_T(x,N_i(w)) \\
\equiv y \geq N_i(I_T(x,N_i(w)))
\]

Proposition 5.25. Let $L$ be a lattice and consider an AFASP$^S$ rule $r:a \leftarrow \alpha$ over $L$ to which an $S$-implicator $I_{T,N_i}$ is attached, where $N_i$ is an involutive negator on $L$. For any $w \in L$ and interpretation $I$ of an AFASP$^S$ program $P$ over $L$ that contains $r$ it holds that:

\[
I_s(r,w) = N_i(I_T(I(\alpha),N_i(w)))
\]

where we recall that $I_T$ is the residual implicator of $T_r$.

Proof. Using the definition of $I_s(r,w)$ and Lemma 5.24 we obtain:

\[
I_s(r,w) = \inf\{k \in L \mid I_{T,N_i}(I(\alpha),k) \geq w\} \\
= \inf\{k \in L \mid y \geq N_i(I_T(I(\alpha),N_i(w)))\} \\
= N_i(I_T(I(\alpha),N_i(w)))
\]

Using the characterization of the support above, the semantics of an AFASP$^S$ program are defined as in Chapter 4.

Now we show that, under certain conditions on the underlying lattice, we can transform an AFASP$^S$ program to an equivalent AFASP program.

Definition 5.26. Let $P$ be an AFASP$^S$ program with rule base $R_P = R_r \cup R_s$ such that $R_r$ is the set of rules associated with residual implicators and $R_s$ is the set of rules associated with $S$-implicators. Furthermore assume that $L_P$ is such that there is a residual implicator $I_i$ that induces an involutive negator.
5.6. S-IMPLICATORS

Then the rule base of the AFASP program $P'$ corresponding to $P$ is defined as:

$$
\mathcal{R}_{P'} = \mathcal{R}_P^s \\
\cup \{ r': a \leftarrow N_i(\mathcal{I}_r(a, not_{w_r})) \mid (r: a \leftarrow_s a) \in \mathcal{R}_P^s, a \in \mathcal{B}_P \} \\
\cup \{ r'_w: w_r \leftarrow \mathcal{I}_r(a', N_i(not_a)) \mid (r: a \leftarrow_s a) \in \mathcal{R}_P^s, a \in \mathcal{B}_P \} \\
\cup \{ r'_w: w_r \leftarrow \mathcal{I}_r(a') \mid (r: k \leftarrow_s a) \in \mathcal{R}_P^s, k \in \mathcal{L}_P \} \\
\cup \{ r'_w: not_a \leftarrow N_i(a) \mid a \in \mathcal{B}_P \} \\
\cup \{ r'_w: not_{w_r} \leftarrow N_i(w_r) \mid (r: a \leftarrow_s a) \in \mathcal{R}_P^s, a \in \mathcal{B}_P \} \\
\cup \{ r'_w: 0 \leftarrow N_i(w_r) \mid (r: a \leftarrow_s a) \in \mathcal{R}_P^s \}
$$

where the literals $w_r$, $not_a$ and $not_{w_r}$ are fresh literals not contained in $\mathcal{B}_P$ and a rule of the form $r: a \leftarrow_i \alpha$ is associated with the implicator $\mathcal{I}_r$. Furthermore for $\alpha = f(b_1, \ldots, b_n; c_1, \ldots, c_m)$ we define $\alpha' = f'(b_1, \ldots, b_n, not_{c_1}, \ldots, not_{c_m})$, with $f'(b_1, \ldots, b_n, not_{c_1}, \ldots, not_{c_m}) = f(b_1, \ldots, b_n; N_i(not_{c_1}), \ldots, N_i(not_{c_m}))$. The aggregator of $P'$ is defined as:

$$
\mathcal{A}_{P'}(\rho) = \begin{cases} 
(\mathcal{A}_P)(\rho') & \text{if } \mathcal{T}_M(\rho(r'), \rho(r'_w), \rho(r'_{not_a}), \rho(r'_{not_{w_r}})) \geq 1 \\
0 & \text{otherwise}
\end{cases}
$$

where $\rho'(r) = \rho(r'_c)$ for any $r \in \mathcal{R}_P^s$ and $\rho'(r) = \rho(r)$ for any $r \in \mathcal{R}_P \setminus \mathcal{R}_P^s$.

Recall that if the subscript is missing from the $\leftarrow$ symbol in an AFASP rule, the implicator that is associated with this rule can be arbitrarily chosen. Hence in the above definition rules $r'$, $r'_w$, $r'_{not_a}$ and $r'_{not_{w_r}}$ are associated with an arbitrary (residual) implicator. The constraint on the lattice $\mathcal{L}$ states that there must exist at least one negator $N_i$ on $\mathcal{L}$ such that for all $x \in \mathcal{L}$ we have that $N_i(x) = \mathcal{I}_r(x, 0)$ and $\mathcal{I}_r(\mathcal{I}_r(x, 0), 0) = x$. This constraint is needed to properly inject the value of the interpretation of the rule in the aggregator. It is clear that in $[0,1]$ the Łukasiewicz implication satisfies this criterion. In general every MV-algebra\footnote{An MV-algebra is an algebraic structure with a binary operation, a unary operation and the constant 0, satisfying certain axioms. They are models of Łukasiewicz logic. See [72] for more information.} satisfies this by definition [72]. Also note that the program defined above still contains S-implicators (viz. the $\mathcal{I}_r$ functions in the bodies). However, they appear as functions in the rule
bodies, and not as the implication associated to a rule. Hence, we have reduced the semantics of a program with mixed S-implication and residual implication rules to a program solely consisting of the latter. Furthermore note that for any interpretation \( I \) of \( P \) and corresponding interpretation \( I' \) of \( P' \) we have \( p_1(r'_c) = I(w_r) \). In this way the aggregator expression obtains the same value for any interpretation of \( P' \) as it does for \( P \). Last, the construction with not\(_a\) is necessary to be able to fix the value of not\(_a\) w.r.t. a certain reduct as in Section 5.5.

As an example, consider the AFASP\(_S\) program \( P_{bbq} \) from Example 5.22. The corresponding AFASP program contains the following rule base:

\[
\begin{align*}
  r_2 & : \quad bad\_weather & \leftarrow & \quad N_W(sunshine) \\
  r_3 & : \quad bbq & \leftarrow & \quad N_W(bad\_weather) \\
  r'_1 & : \quad bad\_weather & \leftarrow & \quad N_W(I_M(rain, not\_w'_{r_1})) \\
  r'_{1w_1} & : \quad w'_{r_1} & \leftarrow & \quad I_{TM}.N_W(rain, N_W(not\_bad\_weather)) \\
  r'_{1not\_bad\_weather} & : \quad not\_bad\_weather & \leftarrow & \quad (N_W(bad\_weather)) \\
  r'_{1nw'_{r_1}} & : \quad not\_w'_{r_1} & \leftarrow & \quad N_W(w'_{r_1}) \\
  r'_1 & : \quad 0 & \leftarrow & \quad N_W(w'_{r_1})
\end{align*}
\]

Suppose we add some facts that tell us that it is raining to a degree of 0.2 and it is sunny to a degree of 0.7. The 1-answer set we obtain for the program from Example 5.22 is \( A = \{\text{rain}^{0.2}, \text{sunny}^{0.7}, \text{bad\_weather}^1, \text{bbq}^0\} \). One can verify that the 1-answer set of the S-implicator free version is \( A' = A \cup \{w'_{r_1}^1\} \cup \{not_{not\_bad\_weather}^0\} \). The following propositions show that the answer sets of the AFASP\(_S\) program coincide with those of the corresponding AFASP program. Hence AFASP\(_S\) programs can be translated to equivalent CFASP programs using the results from Section 5.5. We summarized this in Figure 5.1 on page 156.

**Proposition 5.27.** Let \( P \) be an AFASP\(_S\) program and let \( P' \) be the corresponding AFASP program in Definition 5.26. If \( M \) is an \( m \)-answer set (\( m \in P_P \) and \( m > 0 \)) of \( P \), it holds that \( M' = M \cup \{w_{\rho(M)}^r \mid r \in \mathcal{R}_P^s \} \cup \{not_{\delta_{t}(M'(a))}^a \mid a \in B_P \} \cup \{not_{\rho_{w'}_{r_1}}^r \mid r \in \mathcal{R}_P^s, r_h \in B_P \} \) is an \( m \)-answer set of \( P' \), where \( N_t \) and \( N_r \) are as in Definition 5.26.

*Proof.* See Section 5.A on page 149.
Proposition 5.28. Let $P$ be an AFASP\textsuperscript{s} program and let $P'$ be the corresponding AFASP program as defined in Definition 5.26. If $M'$ is an $m$-answer set ($m \in P'$ and $m > 0$) of $P$, it holds that $M = M' \cap B_P$ is an $m$-answer set of $P$.

Proof. See Section 5.A on page 149.

5.7 Strong Negation

As mentioned in Section 2.2.2, in ASP, there is a second form of negation besides negation-as-failure, called classical negation (also known as strong negation). This form of negation is used when explicit derivation of negative information is needed. The resulting semantic difference can be very important. For example, if we wish to state that it is safe to cross the train tracks when no train is coming we write the following program when using negation-as-failure (from [5]):

\[
\text{cross} \leftarrow \neg \text{train}
\]

This however means that when the information about a train coming is absent, we cross the tracks, which is not the safest thing to do. With strong negation, the problem is written as:

\[
\text{cross} \leftarrow \neg \text{train}
\]

where $\neg \text{train}$ is a special literal that can appear in the head of rules. As the value of $\neg \text{train}$ is derived using the rules of the program and not derived by the absence of information about $\text{train}$, we only cross the tracks when we can explicitly derive that no train is coming. Of course, when both $a$ and $\neg a$ occur in the head of rules there is the possibility of inconsistency. The usual semantics for ASP determine that whenever the standard definition would lead to an answer set of a program $P$ in which both $a$ and $\neg a$ occur, by definition, the only answer set of $P$ is given by $\text{Lit}(P) = B_P \cup \{\neg a \mid a \in B_P\}$ (see [5]). A program in which this occurs is inconsistent and, as in classical propositional logic, anything can be derived from an inconsistent program.

In [175] a fuzzy version of classical negation is introduced. The inconsistency problem in this fuzzy case is solved by attaching to each interpretation $I$ of a program $P$ and literal $a \in B_P$ a score of consistency $I_c(a) = N_c(T_c(1(a), 1(\neg a)))$, where $N_c$ is a negator and $T_c$ a t-norm. The interpretation $I$ is then called $x$-consistent, with $x \in L_P$, iff $A_c(I_c) \geq x$, where $A_c$ is the consistency aggregator that maps
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

$I_c$ to a global consistency score for $I$. This consistency aggregator, which differs from the regular aggregator, is required to be increasing when the consistencies for literals increase.

It is well-known that ASP programs with strong negation can be translated to equivalent programs without strong negation by substituting a new literal $a'$ and adding the constraint $c : \leftarrow a, a'$ for each $\neg a \in \text{Lit}(P)$. The resulting program has no consistent answer sets iff program $P$ has the unique answer set $\text{Lit}(P)$.

We can generalize the procedure for eliminating strong negation in classical ASP to fuzzy programs and embed strong negation in AFASP. In particular, to implement the strong negation approach of [175] into AFASP, we proceed as follows:

**Definition 5.29.** Let $P$ be an AFASP program with strong negation and let $N_c$, $T_c$ and $A_c$ be the negator, t-norm and aggregator expression determining the consistency score of $P$ w.r.t. some interpretation. Furthermore assume that we can define an implicator $I_i$ in $L_P$ that induces an involutive negator $N_i$. Then $P'$, the strong-negation free version of $P$ is defined as follows:

1. $B_{P'} = (B_P \setminus \{\neg a \mid a \in B_P\}) \cup \{a' \mid \neg a \in B_P\}$
2. $R_{P'} = \{r : a \leftarrow a' \mid (r : a \leftarrow \alpha) \in R_P\} \cup \{c_a : 0 \leftarrow_i N_i(N_c(T_c(a, a'))) \mid a \in B_P\}$
3. $A_{P'}(\rho) = (A_c(\{a^{p(c_a)} \mid a \in B_P\}), A_{P}(\rho))$

where for a rule $(r : a \leftarrow \alpha) \in R_P$ we define $\alpha'$ as the expression obtained by replacing each $\neg a$ by the corresponding $a'$. Furthermore a rule of the form $r : a \leftarrow \alpha$ is associated with the $I_i$ implicator. Last, for each $a \in B_P$ it must hold that $a' \not\in B_P$, i.e. $a'$ is a “fresh” literal.

Technically, we first replace all classically negated literals with a fresh variable. Then we “inject” the value of $I_c(a)$ for a literal $a$ into the aggregator of the new program using the $c_d$ rule. Since $I_i$ and $N_i$ are chosen such that $I_i(x, 0) = N_i(x)$ and $N_\alpha(N_i(x)) = x$ we obtain that the evaluation of rule $c_a$ will always be equal to $N_c(T_c(a, a'))$, thus ensuring that the aggregator has access to this value. The reason for this injection method is the fact that the aggregator only takes rule interpretations into account, not regular interpretations, and therefore cannot refer to the value of a single literal. Last, we create the new aggregator as a tuple of the consistency degree and the old aggregator, allowing us to order answer sets using both measures. Note that for an AFASP program $P$ this aggregator obtains a value.
in $\mathcal{L}^2_{P}$, which is a complete lattice.

**Example 5.30.** Consider an AFASP program $P$ with rule base $\mathcal{R}_{P}$:

\[
\begin{align*}
    r_1 &: \quad a \leftarrow_m -b \\
    r_2 &: \quad -b \leftarrow_m 0.2 \\
    r_3 &: \quad -a \leftarrow_m 0.4
\end{align*}
\]

The rule aggregator is defined as $A_{\mathcal{P}}(\rho) = \min(1, \rho(r_1) + \rho(r_2) + \rho(r_3))$ and the consistency aggregator as $A_{\mathcal{C}}(I_{c}) = \inf \{ I_{c}(a) \mid a \in B_{\mathcal{P}} \}$. For all literals the consistency negator $N_{c}$ is $N_{W}$ and the consistency t-norm $T_{c}$ is $T_{M}$. According to Definition 5.29, the strong-negation free version of $P$ is a program $P'$ with rule base $\mathcal{R}_{P'}$:

\[
\begin{align*}
    r_1 &: \quad a \leftarrow_m b' \\
    r_2 &: \quad b' \leftarrow_m 0.2 \\
    r_3 &: \quad a' \leftarrow_m 0.4 \\
    c_{a} &: \quad 0 \leftarrow_l N_{W}(N_{W}(T_{M}(a, a'))) \\
    c_{b} &: \quad 0 \leftarrow_l N_{W}(N_{W}(T_{M}(b, b')))
\end{align*}
\]

The aggregator of $P'$ is $A_{\mathcal{P}'}(\rho) = (A_{\mathcal{C}}(\{a^{0.2}, b^{0.2}, a^{0.4}\}), A_{\mathcal{P}}(\rho))$. Now, consider an interpretation $I = \{a^{0.2}, b^{0.2}, a^{0.4}\}$ of $P$ and the corresponding interpretation $I' = \{a^{0.2}, b^{0.2}, a^{0.4}\}$ of $P'$. Computing $I_{c}$ we obtain that $I_{c}(a) = 1 - T_{M}(I(a), I(-a)) = 1 - T_{M}(0.2, 0.4) = 0.8$ and likewise $I_{c}(b) = 1 - T_{M}(I(b), I(-b)) = 1 - T_{M}(0,0.2) = 1$. Computing $\rho_{I'}$ we easily obtain $\rho_{I'}(r_1) = \rho_{I'}(r_2) = \rho_{I'}(r_3) = 1$; for $c_{a}$ and $c_{b}$ we obtain $\rho_{I'}(c_{a}) = N_{W}(N_{W}(T_{M}(a, a'), 0)) = 1 - (N_{W}(N_{W}(T_{M}(a, a'), 0)) = 1 - (1 - I_{c}(a)) = I_{c}(a)$ and likewise for $c_{b}$ we get $\rho_{I'}(c_{b}) = I_{c}(b)$. The evaluation of $A_{\mathcal{P}'}$ with $\rho_{I'}$ then gives $A_{\mathcal{P}'}(\rho_{I'}) = (A_{\mathcal{C}}(\{a^{0.2}, b^{0.2}, a^{0.4}\}), A_{\mathcal{P}}(\rho_{I'})) = (A_{\mathcal{C}}(I_{c}), A_{\mathcal{P}}(\rho_{I'}))$. Hence, the aggregator indeed maps an interpretation to a tuple containing the consistency degree and the rule aggregation score.

The following proposition links our definition to the strong negation approach of [175].

**Proposition 5.31.** Let $P$ be an AFASP program with strong negation and let $P'$ be its strong-negation free version. Then an interpretation $I'$ of $P'$ is an $(x, y)$-answer set of $P$ iff the corresponding answer set $I$ of $P$ is $x$-consistent in the sense of [175].
Proof. Immediate from the construction of \( P' \) and \( A_{P'} \).

5.8 Summary

To investigate the expressivity of certain FASP constructs we introduced a core language for FASP, called CFASP, that only contains non-constraint rules with monotonically increasing functions and negators in rule bodies. We then studied whether these constructs can be simulated in CFASP.

First, we extended CFASP with constraints, resulting in the language CFASP\(\bot\). We investigated whether CFASP\(\bot\) can be translated to CFASP, i.e. whether CFASP is capable of simulating constraints. We showed that by using specific FASP programs without answer sets, we are able to simulate constraints. This is similar to an existing constraint-elimination procedure for ASP and in fact our method generalizes this procedure to the fuzzy case. Furthermore we showed that constraints can also be used for locking the value of certain atoms in a specific interval.

Second, we extended CFASP with monotonically decreasing functions, resulting in the language CFASP\(\tilde{f}\). We showed that by using an involutive negator we can simulate the decreasing behavior of these functions using only monotonic functions and extended literals. Hence, we proved that any CFASP\(\tilde{f}\) program can be translated to an equivalent CFASP program.

Third, we investigated whether AFASP programs with an aggregator that ranges over the same lattice as the program rules can be simulated in CFASP. At first sight this seems straightforward since for each rule \( r_i \) (\( 1 \leq i \leq n \)) in an AFASP program \( P \) we can add rules of the form \( s_{r_i}: r_i \leftarrow \mathcal{I}_{r_i}(r_i)_b, (r_i)_h \) to its CFASP\(\tilde{f}\) translation \( P' \) and then add a rule \( \text{aggr}: k \leftarrow f_A(r_1, \ldots, r_n) \) that computes the aggregated value. We showed that this naive translation does not correctly preserve the semantics and that care needs to be taken when monotonically decreasing functions, such as \( \mathcal{I}_{r_i} \), occur. We defined a translation that includes the necessary bookkeeping rules for arguments of decreasing functions and proved that it correctly translates an AFASP program to a CFASP\(\tilde{f}\) program, which can be translated to a CFASP program.

Fourth, we extended AFASP with rules that are associated with S-implicators instead of residual implicators, resulting in the language AFASP\(S\). To define the semantics of this language we extended the support concept of AFASP to AFASP\(S\) and showed that the support of rules associated with specific S-implicators can be computed using only a residual implicator and negator. We then used this result to show that AFASP\(S\) can be simulated in AFASP using a simulation that is similar to
the AFASP to CFASP\textsuperscript{f} translation.

Last, we investigated whether a previously proposed extension of AFASP with classical negation can be simulated in AFASP. This extension has a separate negation aggregator that combines the scores of the conjunction of the classically negated literal $\neg l$ and its corresponding atom $l$. We showed that we can inject the value of literals in the rule aggregator function of AFASP by using an involutive negator and constraints. This allows us to simulate the negation aggregator using the rule aggregator function of AFASP.

Our results are important to create a bridge between FASP users who want a rich and diverse modeling language and FASP theoreticians and implementers who want a small core language that is easy to reason about and implement. We summarized our translations in Figure 5.1 on page 156. In the next chapter we show how a subclass of CFASP can be implemented using fuzzy SAT solving techniques. Due to the results in this chapter we know that our implementation method can also be used to solve programs with constraints, monotonically decreasing functions, aggregators, S-implicators and classical negation by including an appropriate solver front-end that translates these constructs to the corresponding CFASP program.

### 5.A Proofs

To show Proposition 5.27, we introduce a few technical lemmas.

**Lemma 5.32.** Let $P$ be an AFASP\textsuperscript{S} program and let $P'$ be its corresponding AFASP program as defined in Definition 5.26. Then if $M$ is a fixpoint of $\Pi_{P,M}^{\rho_M}$ of $P$, it holds for the interpretation $M' = M \cup \{\nu_{r,M}^P(r) \mid r \in R^S_P\} \cup \{\not a^N_M(M(a)) \mid a \in B_P\} \cup \{\not w^N_M(r_M) \mid r \in R^S_P\}$ of $P'$ that

1. $\forall r \in R^T_P \cdot \rho_{M'}(r) = \rho_M(r)$
2. $\forall r \in R^S_P \cdot \rho_{M'}(r') = 1$
3. $\forall r \in R^S_P \setminus C_P \cdot \rho_{M'}(r'_w) = 1$
4. $\forall r \in R^S_P \cap C_P \cdot \rho_{M'}(r'_w) = 1$
5. $\forall r \in R^S_P \cdot \rho_{M'}(r'_\not a) = 1$
6. $\forall r \in R^S_P \cdot \rho_{M'}(r'_\not w) = 1$
7. \( \forall r \in \mathcal{R}_P^s \cdot \rho_{M'}(r') = \rho_M(r) \)

where \( r', r'_w, r'_{\text{not}_a}, \) and \( r'_c \) are defined as in Definition 5.26 and \( \mathcal{C}_P \) is the set of constraints in \( P \).

Proof. We consider these cases separately:

1. Trivial from the definition of \( P' \) and \( M' \).

2. When \( r : a \leftarrow_s a \in \mathcal{R}_P^s \), from Proposition 5.25 we have for the corresponding \( r' \in \mathcal{R}_{P'} \) by definition of \( M' \) that

\[
\rho_{M'}(r') = \mathcal{I}_{r'_{\text{not}_a}}(\mathcal{I}_{r}(\mathcal{I}_{r'}(M'(a), \text{not}_w)), M'(a))
\]

where \( r'_{\text{not}_a} \) and \( r'_{\text{not}_a} \) are defined as in Definition 5.26 and \( \mathcal{C}_P \) is the set of constraints in \( P \).

3. When \( (r : a \leftarrow s a) \in \mathcal{R}_P^s \) and \( a \in \mathcal{B}_P \), using the definition of \( M' \) we have for the corresponding \( r'_w \in \mathcal{R}_{P'} \) that

\[
\rho_{M'}(r'_w) = \mathcal{I}_{r'_w}(\mathcal{I}_{r}(M'(a), \text{not}_w), M'(w))
\]

4. When \( (r : k \leftarrow s a) \in \mathcal{R}_P^s \) and \( k \in \mathcal{L}_P \), we obtain that \( \rho_{M'}(r'_w) = 1 \) similar to the above case.

5. When \( r \in \mathcal{R}_P^s \) with \( r_h = a \), we have for the corresponding \( r'_{\text{not}_a} \in \mathcal{R}_{P'} \) that

\[
\rho_{M'}(r'_{\text{not}_a}) = \mathcal{I}_{r'_{\text{not}_a}}(\mathcal{N}_i(M'(a)), M'(\text{not}_a))
\]

where \( r'_{\text{not}_a} \) and \( r'_{\text{not}_a} \) are defined as in Definition 5.26 and \( \mathcal{C}_P \) is the set of constraints in \( P \).
5.A. PROOFS

6. When \((r:a \leftarrow s a) \in R_p^s\), we obtain for the corresponding \(r'_{nw, r} \in R_{p'}\) that

\[
\rho_{M'}(r'_{nw, r}) = I_{r'_{nw, r}}(N_i(\rho_M(r)) (N_i(\rho_M(r))) = 1
\]

7. When \(r \in R_p\), we obtain for the corresponding \(r'_c \in R_{p'}\) that

\[
\rho_{M'}(r'_c) = I_i(N_i(M'(w'_r)), 0) = N_i((N_i(M'(w'_r)))) = \rho_M(r)
\]

\[\square\]

**Lemma 5.33.** Let \(P\) be an AFASP\(^{\delta}\) program with \(M\) a fixpoint of \(\Pi_{pM', \rho_M}\) and let \(M' = M \cup \{w_{r'}^{\rho_M(r)} \mid r \in R_p^s\} \cup \{\text{not}_a^{N_i(M(a))} \mid a \in B_P\} \cup \{\text{not}_{w_{r'}}^{\rho_M(r)}(r) \mid r \in R_p\}\) be an interpretation of its corresponding AFASP program \(P'\), as defined by Definition 5.26. Then any interpretation \(I\) of \(P\) and \(I'\) of \(P'\) such that \(I(a) = I'(a)\) for all \(a \in B_P\) satisfies

\[
\Pi_{p'M', \rho_{M'}}(I')(I) = \Pi_{pM, \rho_M}(I) \cup \{w_{r'}^{\rho_M(r)} \mid r \in R_p^s\} \cup \{\text{not}_a^{N_i(M(a))} \mid a \in B_P\} \cup \{\text{not}_{w_{r'}}^{\rho_M(r)}(r) \mid r \in R_p\}
\]

with \(N_i\) and \(N_r\) as in Definition 5.26.

**Proof.** Note that by Definition 5.26 one can immediately see that \(B_{p'}\) consists of four partitions, i.e. \(B_{p'} = B_P \cup \{w_{r'} \mid r \in R_p^s \setminus C_P\} \cup \{w_{r'} \mid r \in R_p \cap C_P\} \cup \{\text{not}_a \mid a \in B_P\} \cup \{\text{not}_{w_{r'}} \mid r \in R_p\}\), where \(C_P\) is the set of constraints in \(P\). We consider the elements of these partitions separately.

1. Suppose \((r:a \leftarrow s a) \in R_p^s\) with \(a \in B_P\) and consider the corresponding \(w_{r'} \in B_{p'}\), then by definition of \(\Pi_{p'M', \rho_{M'}}\) and Proposition 4.9 we have

\[
\Pi_{p'M', \rho_{M'}}(I')(w_{r'}) = \sup\{I_r'(r'_b), \rho_{M'}(r') \mid r' \in p'_{w_{r'}}\}
\]

By Definition 5.26 we know that \(p'_{w_{r'}} = \{(r'_{w})^{M'}\}\). From Lemma 5.32 we know \(\rho_{M'}(r'_{w}) = 1\), which, together with Definition 5.26, leads to

\[
\Pi_{p'M', \rho_{M'}}(I')(w_{r'}) = I'((I_r(a), N_i(\text{not}_a)))^{M'}
\]

151
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

Now by definition of the reduct and $M'$, combined with the fact that $N_i$ is involutive, this leads to

$$\Pi_{p_{M'} \phi_{M'}}(I')(w_r) = I'(\mathcal{I}_r(M'(a), M'(a))) = \rho_{M'}(r)$$

Now by the definition of $\rho_{M}$ we obtain

$$\Pi_{p_{M'} \phi_{M'}}(I')(w_r) = \rho_{M'}(r) = \rho_{M}(r)$$

2. Suppose $(r:k \leftarrow s) \in R_{p}^{s}$ with $k \in L_{p}$ and consider the corresponding $w_r \in B_{p'}$. This case follows similar to the above case.

3. Suppose $r:a \leftarrow s \alpha \in R_{p}^{s}$ and consider the corresponding not$a \in B_{p'}$. As $p_{M'}^{M'}_{not a} = \{(r'_{not a})^{M'}\}$, we obtain by the definition of $\Pi_{p_{M'} \phi_{M'}}$ and Proposition 4.9 that

$$\Pi_{p_{M'} \phi_{M'}}(I')(not a) = \mathcal{T}_{\mathcal{I}_{not a}}(I'(\mathcal{I}^{M'}_{not a} b), \rho_{M'}(r'_{not a}))$$

By Lemma 5.32 we know that $\rho_{M'}(r'_{not a}) = 1$, hence by definition of $r'_{not a}$ we get

$$\Pi_{p_{M'} \phi_{M'}}(I')(not a) = I'(\mathcal{N}_i(M(a)))^{M'}$$

By the definition of the reduct and the fact that $M(a) = M'(a)$ for $a \in B_{p}$ this means

$$\Pi_{p_{M'} \phi_{M'}}(I')(not a) = \mathcal{N}_i(M(a)) = M'(not a)$$

4. Suppose $r:a \leftarrow s \alpha \in R_{p}^{s}$ and consider the corresponding not$w_r \in B_{p'}$. This case is entirely analogous to the case for not$a$.

5. Finally, for $a \in B_{p}$ we consider two cases:

(a) Suppose $(r:a \leftarrow \alpha) \in R_{p}^{r}$, then it is easy to see by definition of $P'$, Proposition 4.9 and Lemma 5.32 that

$$I_s(r_{M}, \rho_{M}(r)) = I'_s(r_{M'}, \rho_{M'}(r)) \quad (5.9)$$
5.A. PROOFS

(b) Suppose $r : a ← a \in \mathcal{R}_P$ with corresponding $r' : a ← N_r(\mathcal{I}_r(\alpha, not_w_r))$ in $\mathcal{R}_{P'}$. By definition of reduct, we obtain

$$r^{M'} : a ← N_r(\mathcal{I}_r(\alpha^{M'}, M'(not_w_r)))$$

Hence by definition of $M'$ we obtain

$$r^{M'} : a ← N_r(\mathcal{I}_r(\alpha^{M'}, N_r(\rho_M(r))))$$

Combining this with Proposition 4.9, Proposition 5.25, Lemma 5.32 and the fact that for $a \in B_P \cap B_{P'}$ it holds that $I(a) = I'(a)$, we obtain:

$$I'_s(r^{M'}, \rho_{M'}(r')) = \mathcal{I}_r(\rho_M(r'), I'(N_r(\mathcal{I}_r(\alpha^{M'}, N_r(\rho_M(r))))))$$

$$= N_r(\mathcal{I}_r(\rho_M(r'), N_r(\rho_M(r))))$$

Combining Equation 5a and item 5b we easily obtain that

$$\Pi_{P^M, \rho_M}(I) = \Pi_{P^{M'}, \rho_{M'}}(I') \cap B_P$$

Hence, by combining these cases the stated follows. □

Proof of Proposition 5.27. First, by Lemma 5.32 and Definition 5.26 it is easy to see that $A_{P'}(\rho_{M'}) \geq m$.

Second, we show that $M'$ is an answer set of $P'$, i.e. that it is the least fixpoint of $\Pi_{P^{M'}, \rho_{M'}}$. From Lemma 5.33 we can readily see that $M'$ is a fixpoint of $\Pi_{P^{M'}, \rho_{M'}}$. Suppose now there is an $M'' < M'$ such that $M''$ is also a fixpoint of $\Pi_{P^{M'}, \rho_{M'}}$. We show by contradiction that such an $M''$ cannot exist. First, note that due to Lemma 5.33 and the fact that both $M'$ and $M''$ are fixpoints of $\Pi_{P^{M'}, \rho_{M'}}$, it must hold that for all $l \in B_{P'} \setminus B_P$ we have $M''(l) = M'(l)$. Hence by Lemma 5.33 this means $M'' \cap B_P < M' \cap B_P$ and thus $M'' \cap B_P < M$. However, by Lemma 5.33 and the fact that $M''$ is a fixpoint of $\Pi_{P^{M'}, \rho_{M'}}$, we have that $M'' \cap B_P$ must be a fixpoint of $\Pi_{P^M, \rho_M}$, contradicting the fact that $M$ is the least fixpoint $\Pi_{P^M, \rho_M}$ due to $M$ being an answer set of $P$. □
To show Proposition 5.28, we introduce the following Lemma.

**Lemma 5.34.** Suppose $P$ is an AFASP program and let $P'$ be its corresponding AFASP program as defined in Definition 5.26. Then for any $m$-answer set $M'$ of $P'$, with $0 < m, m \in \mathbb{L}_P$ and $M = M' \cap B_P$, it holds for all $r : a \leftarrow s \in R_P^s$ that

1. $M'(\text{not}_a) = N_i(M'(a))$
2. $M'(\text{not}_{w_r}) = N_r(M'(w_r))$
3. $M'(w_r) = \rho_M(r)$

**Proof.** Since $M'$ is a fixpoint of $\Pi_{P,\not_a}^{M',\not_a}$, these cases follow easily from the definition of $\Pi_{P,\not_a}^{M',\not_a}$, Proposition 4.9, the definition of $P'$ and the fact that $P_{\not_a}^{M'} = \{ (r_{\not_a})^{M'} \}, P_{\not_{w_r}}^{M'} = \{ (r_{\not_{w_r}})^{M'} \}$ and $P_{w_r}^{M'} = \{ (r_{w_r})^{M'} \}$. 

**Proof of Proposition 5.28.** First we show that $A_P(\rho_M) \geq m$. For a rule $r \in R_P^r$, by definition of $P'$ it holds trivially that $\rho_M(r) = \rho_M'(r)$. Further, for each rule $r \in R_P^s$ there is a corresponding rule $r'_r : 0 \leftarrow I_i(w_r)$ in $P'$. From Lemma 5.34 we know that $M'(w_r) = \rho_M(r)$ and thus, as $N_i(x) = I_i(x, 0)$ and $N_i(N_i(x)) = x$ that $\rho_{M'}(r'_r) = \rho_M(r)$. Hence, as $A_{P'}(\rho_M') \geq m$ and $m > 0$, this means $A_P(\rho_M) \geq m$ by definition of $A_P$.

Second we show that $M$ is the least fixpoint of $\Pi_{P,\not_a}^{M',\not_a}$. First we show that it is a fixpoint. From the definition of $\Pi_{P,\not_a}^{M,\not_a}$ and Proposition 4.9 we obtain for $a \in B_P$ that

$$\Pi_{P,\not_a}^{M,\not_a}(M)(a) = \sup \{ M_s(r^M, \rho_M) \mid (r : a \leftarrow s) \in R_P \}$$

We consider two cases: $(r : a \leftarrow s) \in R_P^r$ and $(r : a \leftarrow s) \in R_P^s$.

1. If $(r : a \leftarrow s) \in R_P^r$, then in $P'$ we have an equivalent rule and thus combining this with the former remark that $\rho_M(r) = \rho_M'(r)$ for such rules we obtain

$$M_s(r^M, \rho_M(r)) = M'_s(r^M, \rho_M'(r))$$

2. If $(r : a \leftarrow s) \in R_P^s$, then there is a rule $r' : a \leftarrow N_r(I_T(r, a, w_r))$ in $R_P$. We can show that $M_s(r^M, \rho_M(r)) = M'_s(r'^M, \rho_M'(r))$ for this rule using Proposition 4.9, the fact that $\rho_M'(r') \geq 1$ since $A_{P'}(\rho_M') > 0$, the fact that
5.A. PROOFS

\[ I(\beta^I) = I(\beta) \] for any interpretation \( I \), Lemma 5.34, Proposition 5.25 and the definition of the reduct:

\[
M'_s(r'^{M'}, \rho_{M'}(r)) = T_r'((M'(N_r(\mathcal{I}_{\mathcal{T}_r}(a, \text{not}_{w_r}^{M'}))), \rho_{M'}(r')) \\
= T_r'(M'(N_r(\mathcal{I}_{\mathcal{T}_r}(a, \text{not}_{w_r}))), 1) \\
= N_r(\mathcal{I}_{\mathcal{T}_r}(M'(a), N_r(M'(w_r)))) \\
= N_r(\mathcal{I}_{\mathcal{T}_r}(M'(a), N_r(\rho_M(r)))) \\
= N_r(\mathcal{I}_{\mathcal{T}_r}(M(a), N_r(\rho_M(r)))) \\
= N_r(\mathcal{I}_{\mathcal{T}_r}(M(a)^M, N_r(\rho_M(r)))) \\
= M_s(r^{M}, \rho_M(r))
\]

From item 1 and item 2 we thus obtain for \( a \in B_P \) that

\[
\Pi_{\mu^M, \rho_M}(M)(a) = \sup\{M_s(r^M, \rho_M(r)) \mid r \in P_a\} \\
= \sup\{M'_s(r'^{M'}, \rho_{M'}(r)) \mid r \in P'_a\} \\
= \Pi_{\mu^M', \rho_{M'}}(M')(a) \\
= M'(a) \\
= M(a)
\]

Hence, \( M \) is a fixpoint of \( \Pi_{\mu^M, \rho_M} \). Now, suppose that \( M \) is not the least fixpoint of \( \Pi_{\mu^M, \rho_M} \), then some \( M'' = \Pi_{\mu^M, \rho_M}^{*} < M \) must exist. Consider then

\[
M''' = M'' \cup \{w_r^{\mu_M(r)} \mid r \in \mathcal{R}_P\} \cup \{\text{not}^N_{\alpha}(\mu_M(a)) \mid a \in B_P\} \\
\cup \{\text{not}^N_{\alpha}(\rho_M(r)) \mid r \in \mathcal{R}_P\}
\]

It is clear from the construction of \( M''' \) that \( M''' < M' \). Now, using Lemma 5.34 we know from the construction of \( M'' \) and \( M''' \) that

\[
\Pi_{\mu^M', \rho_{M'}}(M''') = M'''
\]

Hence, \( M''' \) is a fixpoint of \( \Pi_{\mu^M', \rho_{M'}} \), which contradicts the fact that \( M' \) is the least fixpoint of \( \Pi_{\mu^M', \rho_{M'}} \).
CHAPTER 5. CORE FUZZY ANSWER SET PROGRAMMING

5.B Diagram

Figure 5.1: Diagram of the relationships between the different CFASP and AFASP languages.
Reducing FASP to Fuzzy SAT

6.1 Introduction

In recent years, efficient solvers for classical ASP have been developed. Some of these are based on the DPLL algorithm [34] such as Smodels [156] and DLV [53], others use ideas from SAT solving such as clasp [64], while still others directly use SAT solvers to find answer sets, e.g. ASSAT [101], cmodels [68], and pbmodels [102]. The SAT based approaches have been shown to be fast, and have the advantage that they can use the high number of efficient SAT solvers that have been released in recent years. The DPLL based solvers have the advantage that they allow a flexible modeling language, since they are not restricted to what can directly and efficiently be translated to SAT, and that they can be optimized for specific types of programs.

Probabilistic ASP can be reduced to classical SAT [148], allowing implementations using regular SAT solvers. Likewise, possibilistic ASP can be reduced to classical ASP [133], which means ASP solvers can be used for solving possibilistic ASP programs.

In the case of FASP programs with a finite number of truth values, it has been
CHAPTER 6. REDUCING FASP TO FUZZY SAT

shown in [174] that FASP can be solved using regular ASP solvers. However, to date, no FASP solvers or solving methods have been developed for programs with infinitely many truth values. Our goal in this chapter is to take a first step towards creating such efficient solvers by showing how the idea of translating ASP programs to SAT instances can be generalized to FASP. In this way we can create FASP solvers that use existing techniques for solving fuzzy satisfiability problems, such as mixed-integer programming and other forms of mathematical programming (see for example [15, 71]) or, in the case of certain fuzzy satisfiability problems with Łukasiewicz connectives, constraint solvers (see for example [154]). Specifically, we focus on the ASSAT approach introduced in [101]. While translating ASP to SAT is straightforward when ASP programs do not contain cyclic dependencies, called loops, careful attention is needed to correctly cover the important case of programs with loops. The solution presented by ASSAT is based on constructing particular propositional formulas for any loop in the program. In this chapter, we pursue a similar strategy where fuzzy loop formulas are used to correctly deal with loops. Our main contributions can be summarized as follows:

1. We define the completion of the subclass of CFASP⊥ programs on \(([0, 1], \leq)\) with only t-norms and negators in their bodies and show that the answer sets of programs without loops are exactly the models of its completion.

2. By generalizing the loop formulas from [101], we then show how the answer sets of arbitrary CFASP⊥ programs in this subclass can be found. We furthermore show how the ASSAT procedure, which attempts to overcome the problem that the number of loops may be exponential, can be generalized to the fuzzy case.

The structure of the chapter is as follows. We begin by defining the completion of CFASP⊥ programs on \(([0, 1], \leq)\) in Section 6.2 and by discussing the problems that occur with programs containing loops. Section 6.3 then shows how these problems can be solved by adding loop formulas to the completion. We illustrate our approach on the problem of placing a set of ATM machines on the roads connecting a set of cities such that each city has an ATM machine nearby in Section 6.4. Finally, we discuss the issues that arise when generalizing our approach to arbitrary (A)FASP programs in Section 6.5.
6.2 Completion of FASP Programs

In this section we show how a subclass of FASP programs can be translated to fuzzy theories such that the models of these theories correspond to answer sets of the program and vice versa. The class of FASP programs we consider is the subclass of CFASP$^+$ on the lattice $([0, 1], \leq)$ containing programs with only t-norms as monotonic functions. Hence, in the remainder of this chapter the term FASP program refers to a program in this subclass. Note that in Section 4.4.3 on page 110 we have shown that for this subclass the fixpoint semantics and unfounded-based semantics coincide. Since the development of our fuzzy SAT translation can more easily be done using the unfounded-based semantics, whereas our generalization of the ASSAT procedure in Section 6.3 is based on the fixpoint semantics, this is the main reason for only considering this specific subclass of FASP.

Definition 6.1. Let $P$ be a FASP program. The completion of $P$, denoted as $\text{comp}(P)$, is defined as the following set of fuzzy formulas:

$$\{ a \approx (\max\{r_b \mid r \in P_a\}) \mid a \in B_P \} \cup \{ I(r_b, r_h) \mid r \in P, r_h \in [0, 1] \}$$

where $I$ is an arbitrary residual implicator and $\approx$ is the biresiduum of $I$ and an arbitrary t-norm $T$, i.e. for all $x, y \in [0, 1]$ we have that $x \approx y = T(I(x, y), I(y, x))$.

The completion of a program consists of two parts, viz. a part for the literals $\{ a \approx (\max\{r_b \mid r \in P_a\}) \mid a \in B_P \}$, and a part for constraints $\{ I(r_b, r_h) \mid r \in P, r_h \in [0, 1] \}$. The constraints part simply ensures that all constraints are satisfied. The literal part ensures two things. By definition of the biresiduum and the fact that $I(x, y) = 1$ iff $x \leq y$ for any residual implicator$^1$ $I$ and $x, y \in [0, 1]$, we have that $x \approx y = 1$ iff $x \leq y$ and $y \leq x$. Hence, the literal part of the completion establishes first that rules are satisfied and second that the value of the literal is not higher than what is supported by the rule bodies.

---

$^1$See Proposition 2.54 on page 47.
Example 6.2. Consider the following FASP program $P_{\text{ex6.2}}$:

$$
\begin{align*}
    r_1 : & a \leftarrow T_M(b, c) \\
    r_2 : & b \leftarrow 0.8 \\
    r_3 : & c \leftarrow T_M(a, N_W(b)) \\
    r_4 : & 0 \leftarrow T_W(a, b)
\end{align*}
$$

The completion of the above program is the following set of fuzzy propositions

$$
\begin{align*}
    a \approx T_M(b, c) \\
    b \approx 0.8 \\
    c \approx T_M(a, N_W(b)) \\
    I_W(T_W(a, b), 0)
\end{align*}
$$

Note that when applying Definition 6.1 for a literal $l$ that does not appear in the head of any rule, we get $a \approx \max \emptyset$, where we define $\max \emptyset = 0$.

We can now show that any answer set of a program $P$ is a model of its completion $\text{comp}(P)$.

**Proposition 6.3.** Let $P$ be a FASP program and let $\text{comp}(P)$ be its completion. Then any answer set of $P$ is a model of $\text{comp}(P)$.

**Proof.** Suppose $A$ is an answer set of $P$. Since $A$ is a model of $P$ it follows from Lemma 4.49 on page 112 that $A$ is a fixpoint of $\Pi_{P, \varphi}$. Hence for each $a \in B_P$ we have that $A(a) = \sup \{ T_r(A(r_b), 1) \mid r \in P_a \} = \sup \{ A(r_b) \mid r \in P_a \}$. By construction of $\text{comp}(P)$, it then easily follows that $A \models \text{comp}(P)$.

Example 6.4. Consider program $P_{\text{ex6.2}}$ from Example 6.2 and its interpretation $I_1 = \{ a^0, b^{0.8}, c^0 \}$. It is easy to see that $I_1$ is an answer set of $P_{\text{ex6.2}}$ and a model of $\text{comp}(P_{\text{ex6.2}})$.

The reverse of Proposition 6.3 is not true in general, which is unsurprising because it is already invalid for classical answer set programming. The problem occurs for programs with “loops”, as shown in the following example.
6.2. COMPLETION OF FASP PROGRAMS

Example 6.5. Consider program $P_{ex6.2}$ from Example 6.2 and its interpretation $I_2 = \{a^{0.2}, b^{0.8}, c^{0.2}\}$. We can easily see that $I_2$ is a model of $\text{comp}(P_{ex6.2})$, but it is not an answer set of $P_{ex6.2}$.

One might wonder whether taking the minimal models of the completion would solve the above problem. The following example shows that the answer is negative.

Example 6.6. Consider the following program $P_{min}$

$$
\begin{align*}
  r_1 & : \ a \leftarrow \ a \\
  r_2 & : \ p \leftarrow T_W(N_W(p), N_W(a))
\end{align*}
$$

The completion $\text{comp}(P_{min})$ is

$$
\begin{align*}
  a & \approx a \\
  p & \approx T_W(N_W(p), N_W(a))
\end{align*}
$$

Consider now the interpretation $I = \{a^{0.2}, p^{0.4}\}$. Since $I(a) = I(a)$ and $T_W(N_W(I(p)), N_W(I(a))) = \max(0, 1 - I(p) + 1 - I(a) - 1) = 0.4$ we can see that $I$ is a model of $\text{comp}(P_{min})$. We show that it is a minimal model as follows. Suppose some $I' \subset I$ exists. Then we can consider three cases: (i) $I'(a) < I(a)$ and $I'(p) = I(p)$; (ii) $I'(a) = I(a)$ and $I'(p) < I(p)$; (iii) $I'(a) < I(a)$ or $I'(p) < I(p)$. In all three cases we obtain that $T_W(N_W(I'(p)), N_W(I'(a))) > 0.4 > I'(p)$, since $N_W(I'(a)) = 1 - I'(a) > 0.8$ or $N_W(I'(p)) > 0.6$. Hence $I'$ is not a model of $\text{comp}(P_{min})$ and $I$ is thus a minimal model of $\text{comp}(P_{min})$. However, $I'$ is not an answer set of $P_{min}$ since $\Pi_{I^{*}_{P_{min}}} = \{a^{0.1}, p^{0.4}\}$.

However, similar to the crisp case, when a program has no loops in its positive dependency graph, the models of the completion and the answer sets coincide. First we define exactly what a loop of a FASP program is, and then we show that this property still holds for FASP.

Definition 6.7. Let $P$ be a FASP program. The positive dependency graph of $P$ is a directed graph $G_P = \langle B_P, D \rangle$ where $(a, b) \in D$ iff $\exists r \in P_n \cdot b \in r^+_a$. 

161
where $r_\mathbb{P}^+_{b}$ is defined as in Definition 4.47 on page 111. For ease of notation we also denote this relation with $(a, b) \in G_{\mathbb{P}}$ for atoms $a$ and $b$ in the Herbrand base of $\mathbb{P}$. We call a non-empty set $L \subseteq B_{\mathbb{P}}$ a loop of $\mathbb{P}$ iff for all literals $a$ and $b$ in $L$ there is a path (with length $> 0$) from $a$ to $b$ in $G_{\mathbb{P}}$ such that all vertices of this path are elements of $L$.

Note that in the remainder of this chapter the term dependency graph is used for the positive dependency graph.

**Example 6.8.** Consider program $P_{\text{min}}$ from Example 6.6. The dependency graph of $P_{\text{min}}$ is pictured in Figure 6.1. We can see that $\{a\}$ is a loop. If this loop was not in the program, its completion would become

\[
\begin{align*}
a &\approx 0 \\
p &\approx T_{W}(N_{W}(p), N_{W}(a))
\end{align*}
\]

This fuzzy theory has the single model $M = \{p^{0.5}\}$. One can easily verify that $M$ coincides with the answer set of $P_{\text{min}} \setminus \{r_1\}$, i.e. the answer set of the program without the loop $\{a\}$.

**Example 6.9.** Consider program $P_{\text{ex6.2}}$ from Example 6.2. The dependency graph of $P_{\text{ex6.2}}$ is pictured in Figure 6.2. We can clearly see that there is a loop between nodes $a$ and $c$. Due to this loop, the values of $a$ and $c$ are not sufficiently constrained in the completion.

From the preceding examples one might think that removing the loops from the program would be sufficient to make the models of the completion and the answer sets coincide. However, this is not the case, as the semantics of the program then changes.
6.2. COMPLETION OF FASP PROGRAMS

Example 6.10. Consider program $P_{\text{change}}$ consisting of the following rules

\begin{align*}
    r_1 &: a \leftarrow 0.3 \\
    r_2 &: a \leftarrow b \\
    r_3 &: b \leftarrow a
\end{align*}

Its single answer set is $\{a^{0.3}, b^{0.3}\}$. If we remove rule $r_2$ or $r_3$, the answer set of the resulting program is $\{a^{0.3}\}$.

We can now show that for programs without loops the answer sets coincide with the models of their completion. We first introduce two lemmas. Note that in the proofs of these lemmas and other propositions and lemmas we will use the unfounded-based semantics for FASP that are described in Definition 4.47 on page 111.

Lemma 6.11. Let $G = \langle V, E \rangle$ be a directed graph with a finite set of vertices and $X \subseteq V$ with $X \neq \emptyset$. If every node in $X$ has at least one outgoing edge to another node in $X$, there must be a loop in $X$.

Proof. From the assumptions it holds that each $x \in X$ has an outgoing edge to another node in $X$. This means that there is an infinite sequence of nodes $x_1, x_2, \ldots$ such that $(x_i, x_{i+1}) \in E$ for $i \geq 1$. Since $X$ is finite, it follows that some vertex occurs twice in this sequence, and hence that there is a loop in $X$. \qed
Lemma 6.12. Let $P$ be a FASP program, $I$ an interpretation of $P$ and $U \subseteq B_P$. Then if $I \models \text{comp}(P)$ and $U$ is unfounded w.r.t. $I$ it holds that for each $u$ in $U \cap \text{supp}(I)$ there is some $r$ in $P_u$ for which $r_b^+ \cap U \cap \text{supp}(I) \neq \emptyset$ holds².

Proof. Assume that $u \in U \cap \text{supp}(I)$, in other words $u \in U$ and $I(u) > 0$. As $U$ is unfounded w.r.t. $I$, for each $r \in P_u$ it holds that either

1. $r_b^+ \cap U \neq \emptyset$; or
2. $I(r_b) < I(u)$; or
3. $I(r_b) = 0$

We can now show that there is at least one rule $r \in P_u$ that violates the second and third of these conditions, meaning it must satisfy the first.

From $I \models \text{comp}(P)$ we know by construction of $\text{comp}(P)$ that for each $u \in U$, $I(u) = \sup \{ I(r_b) \mid r \in P_u \}$. Hence for each $u \in U$ there is a rule $r \in P_u$ such that $I(u) = I(r_b)$, thus the second condition is violated. Since $I(u) > 0$, it then also follows that the third condition is violated.

In other words there must be some $r \in P_u$ such that $I(r_b) = I(u)$ and $I(r_b) \neq 0$. Since $U$ is unfounded w.r.t. $I$, this means that $r_b^+ \cap U \neq \emptyset$. Since $I(r_b) \neq 0$ implies that $r_b^+ \subseteq \text{supp}(I)$ due to the fact that $T(0, x) = 0$ for any t-norm $T$, we can conclude that there is some $r \in P_u$ such that $r_b^+ \cap U \cap \text{supp}(I) \neq \emptyset$. □

Using these lemmas we can now show that the answer sets of any program without loops in its dependency graph coincide with the models of its completion. This resembles Fages’ theorem on tight programs in classical ASP [54].

Proposition 6.13. Let $P$ be a FASP program. If $P$ has no loops in its positive dependency graph it holds that an interpretation $I$ of $P$ is an answer set of $P$ iff $I \models \text{comp}(P)$.

Proof. We already know from Proposition 6.3 that any answer set of $P$ is necessarily a model of $\text{comp}(P)$, hence we only need to show that every model of $\text{comp}(P)$ is an answer set of $P$ under the conditions of this proposition. As $I \models \text{comp}(P)$, it holds that $I$ is a model of $P$. We show by contradiction that $I$ is unfounded-free. Assume that there is a set $U \subseteq B_P$ such that $U$ is unfounded w.r.t. $I$ and $U \cap \text{supp}(I) \neq \emptyset$. From Lemma 6.12 we know that for each $u \in U \cap \text{supp}(I)$ it holds that there is

²We recall that for an interpretation $I$ of a program $P$ the set $\text{supp}(I)$ contains all atoms $a \in B_P$ for which $I(a) > 0$. See Definition 2.37 on page 41.
some rule $r \in P_u$ such that $r^+_b \cap U \cap \text{supp}(I) \neq \emptyset$. Using the definition of $G_P$, this means that for each such $u$ there is some $u' \in U \cap \text{supp}(I)$ such that $G_P(u, u')$. This however means that there is a loop in $G_P$ by Lemma 6.11, contradicting the assumption. Hence, finding the answer sets of a program with no loops in its positive dependency graph can be done by finding models of its completion.

### 6.3 Loop Formulas

As mentioned in the previous section, sometimes the models of the completion are not answer sets. For boolean answer set programming two solutions have been proposed for this problem. The first solution [13] consists of assigning indices to atoms and requiring for any atom $a$ in a loop $L$ that if the derivation of $a$ depends on some $a' \in L$, this $a'$ must have a lower index than $a$. Interestingly, this requirement can be encoded in SAT, leading to a translation of an ASP program $P$ to a SAT problem $S$ such that the models of $S$ coincide with the answer sets of $P$. The second solution consists of adding loop formulas to the completion, which ensure that models of the completion that are not answer sets cannot occur [101]. Since in the case of FASP a generalization of the former would need an infinite number of indices, one for each combination of atom and truth value, we focus on a generalization of the latter.

To define loop formulas, we start from a partition of the rules whose heads are in some particular loop $L$. Based upon this partition, for every loop $L$ we define a formula in fuzzy logic, such that any model of the completion satisfying these formulas is an answer set.

For any program $P$ and loop $L$ we consider the following partition of the rules in $P$ whose head belongs to the set $L$ (due to [101])

\[
R^+_P(L) = \{ r : a \leftarrow a \mid ((r : a \leftarrow a) \in P) \land (a \in L) \land (r^+_b \cap L \neq \emptyset) \}\]  \hspace{1cm} (6.1)

\[
R^-_P(L) = \{ r : a \leftarrow a \mid ((r : a \leftarrow a) \in P) \land (a \in L) \land (r^+_b \cap L = \emptyset) \}\]  \hspace{1cm} (6.2)

Note that this partition only takes the positive occurrences of atoms in the loop into account. Intuitively, the set $R^+_P(L)$ contains the rules that are “in” the loop $L$, i.e. the rules that are jointly responsible for the creation of the loop in the positive dependency graph, whereas the rules in $R^-_P(L)$ are the rules that are outside of this loop. We will refer to them as “loop rules”, resp. “non-loop rules.”
CHAPTER 6. REDUCING FASP TO FUZZY SAT

Example 6.14. Consider program $P$ from Example 6.2. It is clear that for the loop $L = \{a, c\}$ the set of loop rules is $R^+_P(L) = \{r_1, r_3\}$ and the set of non-loop rules is $R^-_P(L) = \emptyset$.

Example 6.15. Consider program $P$ from Example 6.2 with interpretations $I_1 = \{a^{0}, b^{0.8}, c^{0}\}$ and $I_2 = \{a^{0.2}, b^{0.8}, c^{0.2}\}$ once again. It is clear that in $I_1$ no loop rules were used to derive the values of $a$ and $c$, whereas in $I_2$ only loop rules are used.

Hence there is a problem when the value of literals in a loop are only derived from rules in the loop. To solve this problem, we should require that at least one non-loop rule motivates the value of these loop literals. As illustrated in the next example, one non-loop rule is sufficient as the value provided by this rule can propagate through the loop by applying loop rules.

Example 6.16. Consider program $P_{\text{change}}$ from Example 6.10 again. Clearly this program has a loop $L = \{a, b\}$ with $R^+_P(L) = \{r_2, r_3\}$ and $R^-_P(L) = \{r_1\}$. Consider then interpretations $I_1 = \{a^{0.3}, b^{0.3}\}$ and $I_2 = \{a^{1}, b^{1}\}$. We can easily see that $I_1$ is an answer set of $P$, whereas $I_2$ is not, although they are both models of $\text{comp}(P)$. The problem is that in $I_2$ the values of $a$ and $b$ are higher than what can be derived from the non-loop rule $r_1$, whereas in $I_1$ their values are exactly what can be justified from applying rule $r_1$. The latter is allowed, as values are properly supported from outside the loop, while the former is not, as in this case the loop is “self-motivating”.

To remove the non-answer set models of the completion, we add loop formulas to the completion, defined as follows.

Definition 6.17 (Loop Formula). Let $P$ be a FASP program and $L = \{l_1, \ldots, l_m\}$ a loop of $P$. Suppose that $R^-_P(L) = \{r_1, \ldots, r_n\}$. Then the loop formula induced by loop $L$, denoted by $\mathbb{L}F(L, P)$, is the following fuzzy logic formula:

$$\mathbb{I} = \max(l_1, \ldots, l_m), \max((r_1)_b, \ldots, (r_n)_b)$$ (6.3)
6.3. LOOP FORMULAS

where \( I \) is an arbitrary residual implicator. If \( R_P^{-}(L) = \emptyset \), the loop formula becomes

\[
I(\max(l_1, \ldots, l_m), 0)
\]

The loop formula proposed for boolean answer set programs in [101] is of the form

\[
\neg(\bigwedge(r_1)_b \lor \ldots \lor \bigwedge(r_n)_b) \Rightarrow (\neg l_1 \land \ldots \land \neg l_m)
\]

(6.4) It can easily be seen that (6.3) is a straightforward generalization of (6.4) as the latter is equivalent to

\[
(l_1 \lor \ldots \lor l_m) \Rightarrow (\bigwedge(r_1)_b \lor \ldots \lor \bigwedge(r_n)_b)
\]

Note that this equivalence is preserved in Łukasiewicz logic, but not in Gödel or product logic. Furthermore, since \( I | = \mathcal{I}(\max(l_1, \ldots, l_m), 0) \) only when \( \max(I(l_1), \ldots, I(l_m)) \leq 0 \), it is easy to see that in the case where \( R_P^{-}(L) = \emptyset \), the truth value of all atoms in the loop \( L \) is 0.

**Example 6.18.** Consider program \( P_{\text{ex6.2}} \) from Example 6.2 on page 160 and its interpretations \( I_1 = \{a^{0.0}, b^{0.8}, c^{0.0}\} \) and \( I_2 = \{a^{0.2}, b^{0.8}, c^{0.2}\} \). The loop formula for its loop \( L = \{a, c\} \) is the fuzzy formula \( I_M(\max(a, c), 0) \), since \( R_P^{-}(L) = \emptyset \). It is easy to see that \( I_2 \) – which is not an answer set – does not satisfy this formula, while interpretation \( I_1 \) – which is an answer set – does.

**Example 6.19.** Consider program \( P_{\text{change}} \) from Example 6.10. The loop formula for its loop \( L = \{a, b\} \) is the propositional formula \( I_M(\max(a, b), 0.3) \), since \( R_P^{-}(L) = \{r_1\} \). Again we see that interpretation \( I_1 \) from Example 6.16 satisfies this loop formula, whereas interpretation \( I_2 \) from the same example does not.

We now show that by adding loop formulas to the completion of a program, we get a fuzzy propositional theory that is both sound and complete with respect to the answer set semantics. First we show that this procedure is complete.

**Proposition 6.20.** Let \( P \) be a FASP program, let \( \mathcal{L} \) be the set of all loops of \( P \), and define \( \mathcal{LF}(P) = \{\mathcal{LF}(L, P) \mid L \in \mathcal{L}\} \). For any answer set \( I \) of \( P \), it holds that \( I | = \mathcal{LF}(P) \cup \text{comp}(P) \).
Proof. Suppose \( I \) is an answer set of \( P \) and \( I \not\models \mathbf{LF}(P) \cup \mathbf{comp}(P) \). Since any answer set is a model of \( \mathbf{comp}(P) \) according to Proposition 6.3 on page 160, this means that \( I \not\models \mathbf{LF}(P) \). Hence, the loop formula of some loop \( L \) in \( P \) is not fulfilled; this means:

\[
\sup_{u \in L} I(u) > \sup_{r \in R^-(P, L)} I(r_b)
\]

Consider then the set \( U = \{ u \in L \mid I(u) > \sup_{r \in R^-(P, L)} I(r_b) \} \). We show that \( U \) is unfounded w.r.t. \( I \), i.e. we show that for each \( u \in U \) and rule \( r \in P_u \), at least one of the conditions of Definition 4.47 on page 111 applies.

Since \( P_u = R^+(P, L) \cup R^-(P, L) \), each rule \( r \in P_u \) must either be in \( R^+(P, L) \) or in \( R^-(P, L) \). We consider the following cases:

1. If \( r \in R^-(P, L) \) then by construction of \( U \) it holds that \( I(r_b) < I(u) \).
2. If \( r \in R^+(P, L) \) and \( I(r_b) \leq \sup_{r' \in R^-(P, L)} I(r'_b) \), by construction of \( U \) we have that \( I(r_b) < I(u) \).
3. Suppose \( r \in R^+(P, L) \) and \( I(r_b) > \sup_{r' \in R^-(P, L)} I(r'_b) \). Since \( T(x, y) \leq \min(x, y) \) for each t-norm \( T \), we know that \( I(l) \leq I(l') \) for each \( l \in r^+_b \). Hence for each \( l \in r^+_b \) we have \( I(l) > \sup_{r' \in R^-(P, L)} I(r'_b) \). This means that, since \( r \in R^+(P, L) \) and thus \( r^+_b \cap L \neq \emptyset \), we know from the definition of \( U \) that \( r^+_b \cap U \neq \emptyset \).

Now remark that \( U \cap \text{sup}(I) \neq \emptyset \) as \( U \subseteq \text{sup}(I) \) due to \( I(u) > 0 \) for each \( u \in U \). From the above we can thus conclude that \( U \) is unfounded w.r.t. \( I \), and since \( U \cap \text{sup}(I) \neq \emptyset \), that \( I \) is not unfounded-free: a contradiction. 

Second we show that adding the loop formulas to the completion of a program is a sound procedure.

**Lemma 6.21.** Let \( G = \langle V, E \rangle \) be a directed graph and \( X \subseteq V \), with \( V \) finite, such that each node of \( X \) has at least one outgoing edge to another node in \( X \). Then there is a set \( L \subseteq X \) such that \( L \) is a maximal loop in \( X \) and for each \( l \in L \) we have that there is no \( x \in X \setminus L \) for which \( (l, x) \in E \).

**Proof.** From Lemma 6.11 on page 163 we already know that there must be a loop in \( X \). Hence, there must also be a maximal loop in \( X \). First, remark that maximal loops must of course be disjoint as otherwise their union would form a bigger loop. Consider then the set \( X \), which is a collection of disjoint maximal loops \( L \) and remaining nodes \( S \) (single nodes that are not in any loop). There is an induced graph \( G' \) of \( G \) with nodes \( S \cup L \) (i.e. each maximal loop is a single node in the
6.3. LOOP FORMULAS

induced graph) and edges $E$ induced as usual (i.e. $(L_1, L_2) \in E$ if for some node $l_1$ in $L_1$ there is a node $l_2$ in $L_2$ such that $(l_1, l_2) \in E$ and likewise for the nodes in $S$). Clearly, $G'$ is acyclic as otherwise the nodes in $G'$ on the cycle would create a bigger loop in $X$. Hence, $G'$ has leafs without outgoing edges. However, a leaf cannot be in $S$ since that would imply a node in $X$ without an outgoing edge. Thus we can conclude that all leafs in $G'$ are maximal loops in $X$.  

**Proposition 6.22.** Let $P$ be a FASP program and let $\mathit{LF}(P)$ be the set of all loop formulas of $P$. Then for any interpretation $I$ of $P$ it holds that if $I \models \mathit{LF}(P) \cup \mathit{comp}(P)$, then $I$ must be an answer set of $P$.

**Proof.** Suppose $I \models \mathit{LF}(P) \cup \mathit{comp}(P)$ and $I$ is not an answer set of $P$. Since any model of $\mathit{comp}(P)$ must be a model of $P$, this must mean that $I$ is not unfounded-free, i.e. that there exists a set $U \subseteq B_P$ such that $U$ is unfounded w.r.t. $I$. From Lemma 6.12 on page 164 we know that for each $u \in U \cap \mathit{supp}(I)$ there must be some $r \in P_a$ such that $r^+_b \cap U \cap \mathit{supp}(I) \neq \emptyset$. Hence, by definition of $G_P$ this means that for each $u \in U \cap \mathit{supp}(I)$ there is some $u' \in U \cap \mathit{supp}(I)$ such that $(u, u') \in G_P$. Using Lemma 6.21 this means that there is a set $L \subseteq U \cap \mathit{supp}(I)$ such that $L$ is a loop in $P$ and for each $l \in L$ there is no $u \in (U \cap \mathit{supp}(I)) \setminus L$ such that $(l, u) \in G_P$. In other words, for each $l \in L$ and rule $r \in P_l$ we have that

$$
(U \cap \mathit{supp}(I) \cap r^+_b \neq \emptyset) \Rightarrow (L \cap r^+_b \neq \emptyset) \quad (6.5)
$$

Now, consider $l \in L$. Since $L \subseteq U \cap \mathit{supp}(I)$, we know that $I(l) > 0$. Hence, if $I(r_b) = I(l)$ for some rule $r \in P_l$, we know that $I(r_b) > 0$. As $U$ is unfounded w.r.t. $I$, it follows from Definition 4.47 that $L \cap r^+_b \neq \emptyset$.

Using contraposition, this means that for each $l \in L$ and $r \in P_l$ we have that

$$
(L \cap r^+_b = \emptyset) \Rightarrow (I(r_b) \neq I(l)) \quad (6.6)
$$

By the definition of $\mathit{comp}(P)$, however, we know that for each model of $\mathit{comp}(P)$ and for each $a \in B_P$ and $r \in P_a$ we have $I(a) \geq I(r_b)$. Hence for each $l \in L$ and $r \in P_l$ from (6.6) we have that

$$
(L \cap r^+_b = \emptyset) \Rightarrow (I(r_b) < I(l)) \quad (6.7)
$$

Now, for each $l \in L$ and $r \in R^+_P(L) \cap P_l$ by definition of $R^+_P(L)$ it holds that $L \cap r^+_b = \emptyset$, meaning $I(r_b) < I(l)$. Thus, $\sup \{I(r_b) \mid r \in R^+_P(L)\} < \sup \{I(l) \mid l \in L\}$, meaning $I \not\models \mathit{LF}(L, P)$, a contradiction.  

---

169
A straightforward procedure for finding answer sets would now be to extend the completion of a program with all possible loop formulas and let a fuzzy SAT solver generate models of the resulting fuzzy propositional theory. The models of this theory are the answer sets of the program, as ensured by Propositions 6.20 and 6.22. As there may be an exponential number of loops, however, this translation is not polynomial in general. A similar situation arises for classical ASP. The solution proposed in [101] overcomes this limitation by iteratively adding loop formulas. In particular, a SAT solver is first used to find a model of the completion of a classical ASP program. Then it is checked in polynomial time whether this model is an answer set. If this is not the case, a loop formula, which is not satisfied by the model that was found, is added to the completion. The whole process is then repeated until an answer set is found. We will show that a similar procedure can be used to find answer sets of a FASP program.

Starting from the fixpoint characterization of answer sets of FASP programs, we show in Proposition 6.24 that for any given model of the completion of a program that is not an answer set, we can construct a loop that is violated. Before stating this proposition we define the specific fuzzy set intersection on which it is based.

**Definition 6.23.** Consider a FASP program $P$ and two interpretations $I$ and $J$ of $P$. The **intersection** of $I$ and $J$ is the interpretation $I \odot J$ defined for any $a \in B_P$ as $(I \odot J)(a) = \max(0, I(a) - J(a))$.

**Proposition 6.24.** Let $P$ be a FASP program. If an interpretation $I$ of $P$ is a model of $\text{comp}(P)$ and $I \neq \Pi^*_P$, then some $L \subseteq \text{supp}(I \odot \Pi^*_P)$ must exist such that $I \neq \text{LF}(P, L)$.

**Proof.** Suppose $I$ is an interpretation of $P$ and $I \models \text{comp}(P)$, then from the definition of $\text{comp}(P)$ and Lemma 4.48 on page 111, we can easily see that $I$ is a fixpoint of $\Pi_P$. Since $I \neq \Pi^*_P$, some $I' \subset I$ must exist such that $I' = \Pi^*_P$.

Consider then the set $U = \{u \in B_P \mid I(u) > I'(u)\}$. It holds that $U = \text{supp}(I \odot I')$ since $I' \subset I$ and thus $U = \text{supp}(I \odot \Pi^*_P)$ by definition of $I'$. From the proof of Proposition 4.50 on page 112 we then also know that for this set $U$ the following property holds

$$\forall u \in U \cdot \forall r \in P_u \cdot (r^+_b \cap U = \emptyset) \Rightarrow (I(b) < I(u))$$

(6.8)

We can then show that there is a loop in $U$ whose loop formula is violated. Since $I = \Pi_P(I)$ we know from Lemma 4.48 that $I = \Pi_P(I)$. From the definition of
6.3. LOOP FORMULAS

$\Pi_P$ this means

$$\forall l \in B_P \cdot I(l) = \sup\{I(r_b) \mid r \in P_l\}$$

Since the supremum is attained because $P$ is finite we obtain

$$\forall l \in B_P \cdot \exists r \in P_l \cdot I(l) = I(r_b)$$

As $U \subseteq B_P$ this means

$$\forall u \in U \cdot \exists r \in P_u \cdot I(l) = I(r_b)$$

Using (6.8) it then holds that

$$\forall u \in U \cdot \exists r \in P_u \cdot r_b^+ \cap U \neq \emptyset$$

From the definition of $G_P$ we thus get

$$\forall u \in U \cdot \exists u' \in U \cdot (u,u') \in G_P$$

Using Lemma 6.21 it follows that there is a set $L \subseteq U$ that is a loop in $P$ such that for each $l \in L$ there is no $l' \in U \setminus L$ such that $(l,l') \in E$. In other words, for each $l \in L$ there is no $l' \in U \setminus L$ such that there is a rule $r \in P_l$ for which $l' \in r_b^+$. Hence for each $l \in L$ and rule $r \in P_l$ such that $U \cap r_b^+ \neq \emptyset$, it follows that $L \cap r_b^+ \neq \emptyset$.

From (6.8) and using contraposition this means there is some $L \subseteq U$ that is a loop in $P$ and for each $l \in L$ and $r \in P_l$ if $L \cap r_b^+ = \emptyset$ it must hold that $I(r_b) < I(l)$. Now, for each $l \in L$ and $r \in R^-_P(L) \cap P_l$ by definition it holds that $L \cap \text{supp}(l) = \emptyset$, meaning $I(r_b) < I(l)$. Thus, $\sup\{I(r_b) \mid r \in R^-_P(L)\} < \sup\{I(l) \mid l \in L\}$, meaning $I \nmid LF(L,P)$. \hfill \Box

Now, we can extend the ASSAT-procedure from [101] to fuzzy answer set programs $P$. The main idea of this method is to use fuzzy SAT solving techniques to find models of the fuzzy propositional theory which consists of the completion of $P$, together with the loop formulas of particular maximal loops of $P$. If a model is found which is not an answer set, then we determine a loop that is violated by the model and add its loop formula to the fuzzy propositional theory, after which the fuzzy SAT solver is invoked again. The algorithm thus becomes:

1. Initialize $Loops = \emptyset$

2. Generate a model $M$ of $\text{comp}(I) \cup LF(P, Loops)$, where $LF(P, Loops)$ is the set of loop formulas of all loops in $Loops$. 

171
3. If $M = \Pi^*_p M$, return $M$ as it is an answer set. Else, find the loops occurring in $\text{supp}(I \odot \Pi^*_p M)$, add their loop formulas to $\text{Loops}$ and return to step 2.

The reason that we can expect this process to be efficient is articulated by Proposition 6.24. Indeed, when searching for violated loops, we can restrict our attention to subsets of $\text{supp}(I \odot \Pi^*_p I)$. Although the worst-case complexity of this algorithm is still exponential, in most practical applications, we can expect $\text{supp}(I \odot \Pi^*_p I)$ to be small, as well as the number of iterations of the process that is needed before an answer set is found. In [101] experimental evidence for this claim is provided in the case of classical ASP. Last, note that the fuzzy SAT solving technique depends on the t-norms used in the program. If only the Łukasiewicz t-norm is used, we can use (bounded) mixed integer programming (bMIP) [71] or constraint solvers [154]. Since Fuzzy Description Logic Solvers are based on the same techniques as fuzzy SAT solvers, we also know that for the product t-norm we need to resort to bounded mixed integer quadratically constrained programming (bMICQP) [15].

6.4 Example: the ATM location selection problem

In this section we illustrate our algorithm on a FASP program modeling a real-life problem. Suppose we are tasked with placing $k$ ATM machines $\text{ATM} = \{a_1, \ldots, a_k\}$ on roads connecting $n$ towns $\text{Towns} = \{t_1, \ldots, t_n\}$ such that the distance between each town and some ATM machine is minimized, i.e. we aim to find a configuration in which each town has an ATM that is as close as practically possible. To obtain this we optimize the sum of closeness degrees for each town and ATM. Note that this problem closely resembles the well-known $k$-center selection problem (see e.g. [3]). The difference is that in the $k$-center problem the ATMs need to be placed in towns, where we allow them to be placed on the roads connecting towns. We can model this problem as an undirected weighted graph $G = \langle V, E \rangle$ where $V = \text{Towns}$ is the set of vertices and the edge set $E$ connects two towns if they are directly connected by a road. Given a distance function $d : \text{Towns} \times \text{Towns} \to \mathbb{R}$ that models the distance between two towns, the weight of the edge $(a, b) \in E$ is given by the normalized distance $d(a, b)/d_{\text{sum}}$, where $d_{\text{sum}} = \sum\{d(t_1, t_2) \mid t_1, t_2 \in \text{Towns}\}$.

Since our FASP programs can only have t-norms in rule bodies, we also need to find a way to sum up the distances between towns and ATM machines. By using the

---

3 For cities that are not connected the function $d$ models the distance of the shortest path between them.
nearness degree, or closeness degree, which for a normalized distance \( d \) is defined as \( 1 - d \), we can perform summations of distances in our program. To see this, consider the following derivation:

\[
T_W(1 - dist_1, 1 - dist_2) = \max(1 - dist_1 + 1 - dist_2 - 1, 0) \\
= \max(1 - (dist_1 + dist_2), 0) \\
= 1 - \min(dist_1 + dist_2, 1)
\]

Hence, by applying the Łukasiewicz t-norm on the nearness degrees, we are summing the distances. The program \( P_{ATM} \) solving the ATM selection problem is given as follows:

\[
gloc : \quad \text{loc}(A, T1, T2) \leftarrow T_W(\text{conn}(T1, T2), \beta) \\
gnear : \quad \text{locNear}(A, T1) \leftarrow N_W(\text{locNear}'(A, T1)) \\
gnear' : \quad \text{locNear}'(A, T1) \leftarrow T_W(\text{loc}(A, T1, T2), N_W(\text{near}(T1, T2)), \text{locNear}(A, T2)), T1 \neq T2 \\
nearr : \quad \text{near}(T1, T2) \leftarrow T_W(\text{conn}(T1, T3), \text{near}(T1, T3), \text{near}(T3, T2)) \\
lcor : \quad \text{loc}(A, T1, T2) \leftarrow \text{loc}(A, T2, T1) \\
atmr : \quad \text{ATMNear}(A, T) \leftarrow T_W(\text{loc}(A, T1, T2), \text{locNear}(A, T1), \text{near}(T, T1)) \\
tDist : \quad \text{totNear} \leftarrow T_W(\{\text{ATMNear}(a, t) \mid a \in \text{ATM}, t \in \text{Towns}\})
\]

where

\[
\beta = T_W(\{N_M(\text{loc}(A, T1', T2')) \mid \{T'_1, T'_2\} \neq \{T_1, T_2\}\})
\]

Note that, after grounding, a rule such as lcor actually corresponds to a set of variable-free rules \( \{\text{loc}_{a,t_1,t_2} \mid a \in \text{ATM}, t_1, t_2 \in \text{Towns}\} \). We will keep referring to the specific grounded instance of a rule by the subscript.

Program \( P_{ATM} \) consists of a generate and define part, which for a specific configuration is augmented with an input part consisting of facts. The generate part consists of the three rules gloc, gnear, and gnear', which generate a specific configuration of ATMs. The gloc rule chooses an edge on which the ATM machine \( A \) is placed by guessing a location for an ATM that has not yet been assigned a location, as ensured by the \( \beta \) part of this rule. The gnear and gnear' rules generate a location on this edge where \( A \) is placed. Rules gnear and gnear' originate from the constraint \( d(a, t_1) = d(t_1, t_2) - d(a, t_2) \), where \( d(x, y) \) is the distance between \( x \) and \( y \), if ATM \( a \) is placed on the edge between \( t_1 \) and \( t_2 \). Defining \( n(x, y) \) as the nearness degree between \( x \) and \( y \) and noting that \( n(a, t_1) = 1 - d(a, t_1) = 1 - (d(t_1, t_2) - d(a, t_2)) \).
we can rewrite this constraint in terms of t-norms and nearness degrees:

\[
n(a, t_1) = 1 - (d(t_1, t_2) - d(a, t_2)) \\
= 1 - (d(t_1, t_2) + (1 - d(a, t_2)) - 1) \\
= 1 - T_W(d(t_1, t_2), 1 - d(a, t_2)) \\
= N_W(T_W(1 - n(t_1, t_2), n(a, t_2)))
\]

Hence, the bodies of rules \texttt{gnear} and \texttt{gnear’} ensure that this constraint is satisfied. The reason we need two rules and cannot directly write a rule with body \(N_W(T_W(loc(A, T_1, T_2), N_W(near(T_1, T_2)), locNear(A, T_2)))\) is that the syntax does not allow negation in front of arbitrary expressions.

Rule \texttt{nearr} recursively defines the degree of closeness between two towns based on the known distances for connected towns. Additionally, since the bodies of rules with the same head are combined using the maximum, the nearness degree obtained by \texttt{nearr} is always one minus the distance of the shortest path. The \texttt{locr} rule makes sure that if an ATM is located on the edge between town \(T_1\) and \(T_2\), it is also recognized as being on the edge between \(T_2\) and \(T_1\), as we are working with an undirected graph. The \texttt{atmr} rule defines the location between a particular ATM machine and a town. Note that due to rule \texttt{locr} this rule also covers the case when \(near(T, T_2)\) is higher than \(near(T, T_1)\). The \texttt{tDist} rule aggregates the total distances such that different answer sets of this program can be compared and ordered. In this way we could for example search for the answer set that has a maximal total degree of nearness, i.e. in which the distance from the towns to the ATMs is lowest.

Consider the specific configuration \(G_P = \langle V, E \rangle\) of towns \(Towns = \{t_1, t_2, t_3\}\) depicted in Figure 6.3 and suppose \(ATM = \{a_1, a_2\}\). In Figure 6.4 we depicted a subset of the dependency graph of the grounded version of \(P'_{ATM} = P_{ATM} \cup F\), where \(F\) is the input part of the problem, given by the following rules

\[
F = \{\text{conn}(t, t') \leftarrow 1 \mid t, t' \in Towns, (t, t') \in E\} \\
\cup \{\text{near}(t, t') \leftarrow k \mid t, t' \in Towns, (t, t') \in E, k = 1 - (d(t, t')/d_{sum})\}
\]
6.4. EXAMPLE: THE ATM LOCATION SELECTION PROBLEM

For the configuration depicted in Figure 6.3 the input part $F$ is

$$F = \{\text{conn}(t_1, t_1) \leftarrow 1, \text{conn}(t_1, t_2) \leftarrow 1, \text{conn}(t_1, t_3) \leftarrow 1\}$$
$$\cup \{\text{conn}(t_2, t_1) \leftarrow 1, \text{conn}(t_2, t_2) \leftarrow 1, \text{conn}(t_2, t_3) \leftarrow 1\}$$
$$\cup \{\text{conn}(t_3, t_1) \leftarrow 1, \text{conn}(t_3, t_2) \leftarrow 1, \text{conn}(t_3, t_3) \leftarrow 1\}$$
$$\cup \{\text{near}(t_1, t_1) \leftarrow 1, \text{near}(t_1, t_2) \leftarrow 0.8, \text{near}(t_1, t_3) \leftarrow 0.7\}$$
$$\cup \{\text{near}(t_2, t_1) \leftarrow 0.8, \text{near}(t_2, t_2) \leftarrow 1, \text{near}(t_2, t_3) \leftarrow 0.5\}$$
$$\cup \{\text{near}(t_3, t_1) \leftarrow 0.7, \text{near}(t_3, t_2) \leftarrow 0.5, \text{near}(t_3, t_3) \leftarrow 1\}$$

Note that $\text{conn}$ and $\text{near}$ need to be reflexive since an ATM $a$ can be placed directly in a town $t$. It is clear that $P'_{\text{ATM}}$ contains a number of loops. The completion of $P'_{\text{ATM}}$ is the following fuzzy propositional theory:

$$\text{conn}(t_1, t_1) \approx 1, \quad \text{conn}(t_1, t_2) \approx 1, \quad \text{conn}(t_1, t_3) \approx 1$$
$$\text{conn}(t_2, t_1) \approx 1, \quad \text{conn}(t_2, t_2) \approx 1, \quad \text{conn}(t_2, t_3) \approx 1$$
$$\text{conn}(t_3, t_1) \approx 1, \quad \text{conn}(t_3, t_2) \approx 1, \quad \text{conn}(t_3, t_3) \approx 1$$
$$\text{near}(t_1, t_1) \approx 1, \quad \text{near}(t_1, t_2) \approx 0.8, \quad \text{near}(t_1, t_3) \approx 0.7$$
$$\text{near}(t_2, t_1) \approx 0.8, \quad \text{near}(t_2, t_2) \approx 1, \quad \text{near}(t_2, t_3) \approx 0.5$$
$$\text{near}(t_3, t_1) \approx 0.7, \quad \text{near}(t_3, t_2) \approx 0.5, \quad \text{near}(t_3, t_3) \approx 1$$
$$\text{loc}(a_1, t_1, t_1) \approx \max(\mathcal{T}_W(\text{conn}(t_1, t_1), \beta_{1,1,1}), \text{loc}(a_1, t_1, t_1))$$
$$\text{loc}(a_1, t_1, t_2) \approx \max(\mathcal{T}_W(\text{conn}(t_1, t_2), \beta_{1,1,2}), \text{loc}(a_1, t_2, t_1))$$
$$\text{loc}(a_1, t_1, t_3) \approx \max(\mathcal{T}_W(\text{conn}(t_1, t_3), \beta_{1,1,3}), \text{loc}(a_1, t_3, t_1))$$
$$\ldots$$
$$\text{loc}(a_2, t_3, t_1) \approx \max(\mathcal{T}_W(\text{conn}(t_3, t_1), \beta_{2,3,1}), \text{loc}(a_2, t_1, t_3))$$
$$\text{loc}(a_2, t_3, t_2) \approx \max(\mathcal{T}_W(\text{conn}(t_3, t_2), \beta_{2,3,2}), \text{loc}(a_2, t_2, t_3))$$
$$\text{loc}(a_2, t_3, t_3) \approx \max(\mathcal{T}_W(\text{conn}(t_3, t_3), \beta_{2,3,3}), \text{loc}(a_2, t_3, t_3))$$
$$\text{locNear}(a_1, t_1) \approx N_W(\text{locNear}'(a_1, t_1))$$
$$\ldots$$
$$\text{locNear}(a_2, t_3) \approx N_W(\text{locNear}'(a_2, t_3))$$
$$\text{locNear}'(a_1, t_1) \approx \max(\mathcal{T}_W(\text{loc}(a_1, t_1, t_2), \text{locNear}(a_1, t_2), N_W(\text{near}(t_1, t_2))),$$
$$\mathcal{T}_W(\text{loc}(a_1, t_1, t_3), \text{locNear}(a_1, t_3), N_W(\text{near}(t_1, t_3))))$$
$$\ldots$$
CHAPTER 6. REDUCING FASP TO FUZZY SAT

\[ \text{locNear}'(a_2, t_3) \approx \max (T_W(\text{loc}(a_2, t_3, t_1), \text{locNear}(a_2, t_1), N_W(\text{near}(t_3, t_1))), \\
T_W(\text{loc}(a_2, t_3, t_2), \text{locNear}(a_2, t_2), N_W(\text{near}(t_3, t_2)))) \]

\[ \text{near}(t_1, t_1) \approx \max (T_W(\text{conn}(t_1, t_1), \text{near}(t_1, t_1), \text{near}(t_1, t_1)), \\
T_W(\text{conn}(t_1, t_2), \text{near}(t_1, t_2), \text{near}(t_1, t_2)), \\
T_W(\text{conn}(t_1, t_3), \text{near}(t_1, t_3), \text{near}(t_1, t_3), 1) \]

\[ \text{near}(t_1, t_2) \approx \max (T_W(\text{conn}(t_1, t_1), \text{near}(t_1, t_1), \text{near}(t_1, t_2)), \\
T_W(\text{conn}(t_1, t_2), \text{near}(t_1, t_2), \text{near}(t_1, t_2)), \\
T_W(\text{conn}(t_1, t_3), \text{near}(t_1, t_3), \text{near}(t_1, t_3), 0.8) \]

\[ \ldots \]

\[ \text{near}(t_3, t_3) \approx \max (T_W(\text{conn}(t_3, t_3), \text{near}(t_3, t_3), \text{near}(t_3, t_3)), \\
T_W(\text{conn}(t_3, t_2), \text{near}(t_3, t_2), \text{near}(t_3, t_2)), \\
T_W(\text{conn}(t_3, t_1), \text{near}(t_3, t_1), \text{near}(t_3, t_1), 1) \]

\[ \text{ATMNear}(a_1, t_1) \approx \max (T_W(\text{loc}(a_1, t_1, t_1), \text{locNear}(a_1, t_1), \text{near}(t_1, t_1))), \\
T_W(\text{loc}(a_1, t_1, t_2), \text{locNear}(a_1, t_1), \text{near}(t_1, t_1)), \\
\ldots \]

\[ T_W(\text{loc}(a_1, t_3, t_2), \text{locNear}(a_1, t_3), \text{near}(t_1, t_3)) \]

\[ T_W(\text{loc}(a_1, t_3, t_3), \text{locNear}(a_1, t_3), \text{near}(t_1, t_3)) \]

\[ \ldots \]

\[ \text{ATMNear}(a_2, t_3) \approx \max (T_W(\text{loc}(a_2, t_1, t_1), \text{locNear}(a_2, t_1), \text{near}(t_3, t_1))), \\
T_W(\text{loc}(a_2, t_1, t_2), \text{locNear}(a_2, t_1), \text{near}(t_3, t_1)), \\
\ldots \]

\[ T_W(\text{loc}(a_2, t_3, t_2), \text{locNear}(a_2, t_3), \text{near}(t_3, t_3)) \]

\[ T_W(\text{loc}(a_2, t_3, t_3), \text{locNear}(a_2, t_3), \text{near}(t_3, t_3)) \]

\[ \text{totNear} \approx T_W\{\text{ATMNear}(a, t) \mid a \in \text{ATM}, t \in \text{Towns}\} \]

where

\[ \beta_{i,j,k} = T_W(\{N_M(\text{loc}(a_i, t'_j, t'_k)) \mid \{t'_j, t'_k\} \neq \{t_j, t_k\}\}) \]

Note that e.g. the 1 in the right-hand side of the fuzzy proposition with \(\text{near}(t_1, t_1)\) on the right-hand side stems from the inputs \(F\) we added to \(P_{ATM}'\). From the completion \(\text{comp}(P'_{ATM})\) we can see that an interpretation \(M\) satisfying \(M(\text{near}(t_1, t_2)) = 1\) can be a model of \(\text{comp}(P'_{ATM})\), which is clearly unwanted as this would overes-
timate the nearness degrees between towns (i.e. underestimate the distances). For example, consider

\[ M = \{ \text{loc}(a_1,t_1,t_2)^1, \text{loc}(a_1,t_2,t_1)^1, \text{loc}(a_2,t_1,t_3)^1, \text{loc}(a_2,t_3,t_1)^1, \]
\[ \text{locNear}(a_1,t_1)^1, \text{locNear}(a_1,t_2)^1, \text{locNear}(a_2,t_1)^{0.75}, \text{locNear}(a_2,t_3)^{0.75}, \]
\[ \text{locNear}'(a_2,t_1)^{0.25}, \text{locNear}'(a_2,t_3)^{0.25}, \text{near}(t_1,t_1)^1, \text{near}(t_1,t_2)^1, \]
\[ \text{near}(t_2,t_1)^1, \text{near}(t_1,t_3)^{0.7}, \text{near}(t_3,t_1)^{0.7}, \text{near}(t_2,t_3)^{0.5}, \text{near}(t_3,t_2)^{0.5}, \]
\[ \text{near}(t_2,t_2)^1, \text{near}(t_3,t_3)^1, \text{ATMNear}(a_1,t_1)^1, \text{ATMNear}(a_1,t_2)^1, \]
\[ \text{ATMNear}(a_1,t_3)^{0.7}, \text{ATMNear}(a_2,t_1)^{0.75}, \text{ATMNear}(a_2,t_2)^{0.75}, \]
\[ \text{ATMNear}(a_2,t_3)^{0.75} \} \]

Note that atoms \( a \) for which \( M(a) = 0 \) are not included in the set notation, which is e.g. the case for \( \text{totNear} \). One can easily verify that \( M \) is a model of \( \text{comp}(P'_{\text{ATM}}) \).

To check whether \( M \) is an answer set we compute \( \Pi^*_M(P'_{\text{ATM}}) \) by repeatedly applying \( \Pi^*_M(P'_{\text{ATM}}) \), starting from the empty set, until we obtain a fixpoint, and check whether \( M = \Pi^*_M(P'_{\text{ATM}}) \). Performing this procedure, we obtain

\[ \Pi^*_M(P'_{\text{ATM}}) = \{ \text{loc}(a_1,t_1,t_2)^1, \text{loc}(a_1,t_2,t_1)^1, \text{loc}(a_2,t_1,t_3)^1, \text{loc}(a_2,t_3,t_1)^1, \]
\[ \text{locNear}(a_1,t_1)^1, \text{locNear}(a_1,t_2)^1, \text{locNear}(a_2,t_1)^{0.75}, \text{locNear}(a_2,t_3)^{0.75}, \]
\[ \text{locNear}'(a_2,t_1)^{0.25}, \text{locNear}'(a_2,t_3)^{0.25}, \text{near}(t_1,t_1)^1, \text{near}(t_1,t_2)^{0.8}, \]
\[ \text{near}(t_2,t_1)^{0.8}, \text{near}(t_1,t_3)^{0.5}, \text{near}(t_3,t_1)^{0.5}, \text{near}(t_2,t_3)^{0.7}, \text{near}(t_3,t_2)^{0.7}, \]
\[ \text{near}(t_2,t_2)^1, \text{near}(t_3,t_3)^1, \text{ATMNear}(a_1,t_1)^1, \text{ATMNear}(a_1,t_2)^1, \]
\[ \text{ATMNear}(a_1,t_3)^{0.7}, \text{ATMNear}(a_2,t_1)^{0.75}, \text{ATMNear}(a_2,t_2)^{0.75}, \]
\[ \text{ATMNear}(a_2,t_3)^{0.75} \} \]

We can see that \( \Pi^*_M(\text{near}(t_1,t_2)) = 0.8 \neq M(\text{near}(t_1,t_2)) \), hence \( M \) is not an answer set of \( P'_{\text{ATM}} \). From Proposition 6.24 we then know that there must be a loop in \( \text{supp}(M \odot \Pi^*_M(P'_{\text{ATM}})) = \{ \text{near}(t_1,t_2), \text{near}(t_2,t_1) \} \) whose loop formula is violated. Looking at the dependency graph, we can see that \( L = \text{supp}(M \odot \Pi^*_M(P'_{\text{ATM}})) = \{ \text{near}(t_1,t_2), \text{near}(t_2,t_1) \} \) contains three loops: \( L_1 = L_2 = L_3 \).
CHAPTER 6. REDUCING FASP TO FUZZY SAT

$L_2 = \{\text{near}(t_1, t_2)\}$ and $L_3 = \{\text{near}(t_2, t_1)\}$. Their loop formulas are

$$\text{LF}(L_1, P'_{\text{ATM}}) = \mathcal{I}\left(\max\left(\text{near}(t_1, t_2), \text{near}(t_2, t_1)\right), \max\left(\mathcal{T}_W(\text{conn}(t_1, t_3), \text{near}(t_1, t_3), \text{near}(t_2, t_3), \text{near}(t_3, t_1))\right)\right)$$

$$\text{LF}(L_2, P'_{\text{ATM}}) = \mathcal{I}\left(\max\left(\text{near}(t_1, t_2)\right), \max\left(\mathcal{T}_W(\text{conn}(t_1, t_3), \text{near}(t_1, t_3), \text{near}(t_3, t_2)), 0.8\right)\right)$$

$$\text{LF}(L_3, P'_{\text{ATM}}) = \mathcal{I}\left(\max\left(\text{near}(t_2, t_1)\right), \max\left(\mathcal{T}_W(\text{conn}(t_2, t_3), \text{near}(t_2, t_3), \text{near}(t_3, t_1)), 0.8\right)\right)$$

Clearly, these loop formulas are violated by $M$, hence following the algorithm introduced in Section 6.3, we create a new fuzzy propositional theory $\text{comp} (P'_{\text{ATM}}) \cup \{\text{LF}(L_1, P'_{\text{ATM}}), \text{LF}(L_2, P'_{\text{ATM}}), \text{LF}(L_3, P'_{\text{ATM}})\}$, and try to find a model of this new theory. Consider then the following model of this new theory:

$$M = \{\text{loc}(a_1, t_1, t_2)^1, \text{loc}(a_1, t_2, t_1)^1, \text{loc}(a_2, t_1, t_3)^1, \text{loc}(a_2, t_3, t_1)^1, \text{locNear}(a_1, t_1)^0.15, \text{locNear}(a_1, t_2)^0.05, \text{locNear}'(a_1, t_1)^0.85, \text{locNear}'(a_1, t_2)^0.95, \text{locNear}(a_2, t_1)^0.75, \text{locNear}(a_2, t_3)^0.75, \text{locNear}'(a_2, t_1)^0.25, \text{locNear}'(a_2, t_3)^0.25, \text{near}(t_1, t_1)^1, \text{near}(t_1, t_2)^0.8, \text{near}(t_2, t_1)^0.8, \text{near}(t_1, t_3)^0.7, \text{near}(t_3, t_1)^0.7, \text{near}(t_2, t_3)^0.5, \text{near}(t_3, t_2)^0.5, \text{near}(t_2, t_2)^1, \text{near}(t_3, t_3)^1, \text{ATMNear}(a_1, t_1)^0.85, \text{ATMNear}(a_1, t_2)^0.95, \text{ATMNear}(a_1, t_3)^0.55, \text{ATMNear}(a_2, t_1)^0.75, \text{ATMNear}(a_2, t_2)^0.55, \text{ATMNear}(a_2, t_3)^0.75\}$$

One can readily verify that this model is an answer set of $P'_{\text{ATM}}$, hence the algorithm stops and returns $M$.

We could have solved this problem using Mixed Integer Programming (MIP). However, the exact encoding of this problem would be less clear and straightforward to write. The reason for this is that in the MIP translation the loop formulas would

$^4$Though in general the Gödel negation $N_M$ cannot be implemented in MIP, in the ATM example we can implement the $gloc$ rules using integer variables.
6.4. EXAMPLE: THE ATM LOCATION SELECTION PROBLEM

Figure 6.3: Town configuration for $P_{ATM}$. The weights on the edges denote the nearness degrees between towns $t_1$, $t_2$ and $t_3$.

Figure 6.4: Dependency graph of $P_{ATM}$.
need to be explicitly represented in the program, while in FASP this is handled implicitly. Hence, only the implementer of a FASP system needs to handle these loop formulas, not the developer who writes the FASP programs. This is exactly the power of FASP: providing an elegant and concise modeling language for representing continuous problems, which, thanks to the results in this chapter, can be automatically translated to lower-level languages for solving continuous problems such as MIP. Of course, to make FASP really practical it needs to be augmented with high-level constructs such as choice rules. This is similar to ASP, where solvers support numerous high-level language extensions that are translated to regular rules before solving the program.

6.5 Discussion

The reader might wonder why we limit our approach to CFASP⊥ programs with t-norms in their body, because at first sight it seems the presented approach is easily extendable to arbitrary functions. It turns out that this is not the case, however. Consider CFASP⊥ with the Łukasiewicz t-norm in rule bodies. As mentioned before, the completion of such a program, and its loop formulas, are formulas in Łukasiewicz logic and are implementable using MIP. Now let us consider CFASP⊥ where both the Łukasiewicz t-norm and the Łukasiewicz t-conorm may occur in rule bodies. At first, one would suspect that the loop formulas of such a program would again be formulas in Łukasiewicz logic. This turns out to be wrong however. To see this, consider the following rules:

\[ b \leftarrow N_M(a) \]
\[ b \leftarrow S_W(b, b) \]

One can readily verify that in the answer sets of a program containing these rules, literal \( b \) will be equal to \( N_M(a) \) (provided that \( b \) does not occur in the head of any other rule). However, the negation \( N_M \) cannot be implemented in MIP, as the solution space of a MIP problem is always a topologically closed set (viz. the union of a finite number of polyhedra), whereas the solution space of a constraint \( b \approx N_M(a) \) cannot be represented as a closed set due to the strict negation in the definition of \( N_M \). This means that as soon as the Łukasiewicz t-conorm is allowed, in general, there will not exist a Łukasiewicz logic theory such that the models of that theory coincide with the answer sets of a given program. Hence, it is clear that the case where other operators than t-norms are used requires a different strategy.
Finding generalized loop formulas that cover e.g. both the Łukasiewicz t-norm and t-conorm is not a trivial problem. To illustrate some of the issues, let us examine two intuitive candidates. First, remark that the loop formulas introduced in Section 6.3 eliminate certain answer sets (i.e. they are too strict). Consider the following program $P$:

\begin{align*}
  a & \leftarrow S_W(a, b) \\
  b & \leftarrow S_W(a, b) \\
  b & \leftarrow k
\end{align*}

where $k \in [0, 1]$. This program has a loop $\{a, b\}$ with corresponding loop formula $\max(a, b) \leq k$. Now note that for $k > 0$ the value of $a$ in any answer set is equal to 1. Hence, the loop formula incorrectly eliminates all answer sets in this case. One might think this can be solved by including a condition in the loop formula: $(\max(a, b) \leq 1) \lor (b > 0)$. This formula however fails to eliminate models that are not answer sets (i.e. it is not strict enough) on the following program:

\begin{align*}
  a & \leftarrow \mathcal{T}_M(S_W(a, b), 0.8) \\
  b & \leftarrow S_W(a, b) \\
  b & \leftarrow k
\end{align*}

If $k > 0$ the unique answer set of this program is $\{a^{0.8}, b^1\}$. However, $\{a^1, b^1\}$ is also a model of the completion of this program and satisfies the above loop formula.

Although again more refined loop formulas can be thought of that handle the latter program correctly, we are pessimistic about the possibility of finding loop formulas that cover all cases. It appears that such a general solution should be able to capture some underlying idea of recursion: one loop may justify the truth value of some atom $a$, up to a certain level, which may then trigger other rules that justify the truth value of $a$, up to some higher level, etc.

Note that this problem does not occur in classical ASP (or when using the maximum t-conorm), since e.g. $a \leftarrow b \lor c$ is equivalent to $a \leftarrow b$ and $a \leftarrow c$, which is indeed why disjunctions in the body of rules are not considered in classical ASP.

Finally, we would like to remark that due to the translations defined in the preceding chapter our solving method can also be used for (A)FASP programs on $([0, 1], \leq)$ with only t-norms and negators in rule bodies.
CHAPTER 6. REDUCING FASP TO FUZZY SAT

6.6 Summary

Motivated by the existence of ASP solvers that translate ASP to a SAT problem we investigated in this chapter whether fuzzy SAT solvers can likewise be used for solving FASP. First we defined the completion of a FASP program and demonstrated that answer sets of a program are models, but that the reverse does not hold. Since the reverse property is important to be able to use fuzzy SAT solvers for solving FASP, we investigated whether there are conditions under which it holds. We showed that for programs that do not have loops in their positive dependency graph, the models of the completion of a program coincide with the answer sets of the program. For programs with loops in their positive dependency graph we demonstrated that we can create a fuzzy logic theory whose models coincide with the answer sets of a program $P$ by adding formulas corresponding to the loops in the positive dependency graph of $P$ to the completion of $P$. In the worst-case scenario the number of loop formulas that need to be added can grow exponentially in the number of atoms in the Herbrand Base, however. To cope with this problem we generalized the ASSAT procedure that exists for ASP to FASP. This procedure solves a FASP program $P$ by first using a fuzzy SAT solver to find a model of the completion of $P$. If the returned model $M$ is an answer set, the procedure ends; otherwise the loop formula of the loops that are violated by $M$ are added to the completion and the resulting fuzzy logic theory is solved by the fuzzy SAT solver. This is repeated until an answer set is found. We illustrated this procedure on a variant of the $k$-center problem where ATMs need to be placed along roads connecting towns such that each town is close to an ATM.

The results we obtained in this chapter are only valid for a subset of CFASP$^\bot$ where the monotonic functions are t-norms. We discussed the reason for this limitation and argued that a generalization of the loop formulas to arbitrary (C)FASP programs is very difficult due to the need of capturing some underlying idea of recursion. Nevertheless, the class of CFASP$^\bot$ programs with t-norms as monotonic functions is already quite large and due to the results in the previous chapter includes, among others, CFASP$^f$ and AFASP programs with t-norms as monotonic functions and, in the latter case, also an aggregator that can be written using only t-norms and negators. Hence, our results provide an important first step towards the development of fast FASP solvers that are based on fuzzy SAT solvers.
Conclusions

With the ever-growing importance of computers in our modern-day society, the need for languages that allow to quickly write non-defective programs instructing these machines is pressing. While programming languages have come a long way since the days of punched cards, the large number of errors in current software shows that we have not yet attained this goal. A growing trend in computer science is to solve problems using a language-oriented approach. The idea is that instead of solving a problem in a general-purpose programming language such as Java or Haskell, we first create a domain-specific language that is very close to the problem domain and then model the problem in this high-level language. The advantage of this approach is that programs can be written much faster and that it becomes easier to spot differences between the implementation and the intentions of the programmer. One such domain-specific language is answer set programming (ASP). ASP is a declarative programming language with roots in logic programming that allows to model combinatorial optimization problems in a concise and elegant manner. Among others it has been used to implement planning problems, configuration optimizations and decision support systems. Furthermore it has also been used as an intermediate language for other domain-specific languages, such as those used for modeling biological networks.

However, while ASP provides a rich language for modeling combinatorial problems, it is not directly suitable for modeling problems with continuous domains. Such problems occur naturally in diverse fields such as the design of gas and electricity networks, computer vision, business process management and investment portfolios. To overcome this problem, we studied the combination of ASP with fuzzy logic – a class of multi-valued logics that can handle continuity. The resulting formalism is called fuzzy answer set programming (FASP). After a short chapter that recalled some preliminary notions on ASP and fuzzy logic we described FASP in Chapter 3.

An important issue when modeling continuous optimization problems is how we
should handle overconstrained problems, i.e. problems that have no solutions. In many cases we can opt to accept an imperfect solution, i.e. a solution that does not satisfy all the stated rules (constraints). However, this leads to the question: what imperfect solutions should we choose? Current approaches to FASP implement approximate solutions in a rather limited way, where users are required to state to what extent each rule is satisfied. However, this leads to the problem of guessing the right weights and does not allow to rank candidate solutions. In Chapter 4 we proposed a method that is more flexible, called aggregated fuzzy answer set programming (AFASP). The basic idea is not to state the extent to which each rule should be satisfied beforehand, but to make this dynamic and let an aggregator function determine an overall score of suitability of a solution. The AFASP approach we proposed extends a previous proposal of FASP with aggregators in two ways. First, the aggregator expression is decoupled from the lattice underlying the truth values. This results in a more flexible language. Second, the semantics are based on fixpoint semantics for FASP instead of unfounded-based semantics. This ensures that we are no longer restricted to t-norms in rule bodies and furthermore reveals more closely the underlying links between different FASP formalisms. Moreover it solves a problem with the previous proposal when generalizing from $([0, 1], \leq)$ to arbitrary (complete) lattices. Finally, we illustrated AFASP on two examples: continuous graph coloring and the reviewer assignment problem. By means of these examples we illustrated the practical usefulness and the advantages of the flexibility of our framework.

While the addition of aggregators and other language constructs to FASP is very useful for programmers, it makes it harder to implement and reason about FASP. Therefore, in Chapter 5, we investigated whether it was possible to create a core language for FASP that was still able to express many FASP constructs. To this end, we proposed a language, called CFASP, that only consists of non-constraint rules with monotonically increasing functions and negators in rule bodies. We showed that CFASP is capable of simulating constraints, monotonically decreasing functions, AFASP with restricted aggregators, S-implicators and a generalization of classical negation. We also showed that the simulations of constraints and classical negation bear a great resemblance with their simulations in ASP, which provides additional insight into the connections between ASP and FASP.

The analysis in this chapter is important both from a theoretical and practical point of view. From the theoretical perspective, CFASP makes reasoning over FASP simpler, while our simulation results show that the theoretical results are still strong enough to cover a whole range of FASP programs. From the practical perspective
our results show that solvers only need to implement the core language. Note that the fact that many extensions of FASP can be compiled to CFASP does not mean that these extensions are useless: many of the simulations provided in this chapter are cumbersome. To make FASP an intuitive language that is easy to model in, these extensions are thus of crucial importance.

Motivated by the need for an implementation method for FASP we focused on the translation of FASP programs to particular satisfiability problems in Chapter 6. Such a translation provides us additional theoretical insights with respect to the links between FASP and fuzzy SAT on the one hand, and a practical implementation for FASP by reducing it to a fuzzy SAT problem on the other hand. We introduced the completion of a program and showed that in the case of programs without loops, the models of the completion are exactly the answer sets. Hence, the use of FASP has no real advantage over the use of fuzzy SAT when programs do not contain loops as writing such a FASP program requires the same amount of effort as writing the equivalent fuzzy SAT problem. This is similar to ASP, where a similar link between ASP programs without loops and boolean SAT exists.

For programs that do contain loops, we showed that we can reduce them to a fuzzy SAT problem by generalizing the notion of loop formulas in ASP to FASP. Since this translation is not as straightforward as for programs without loops, we can conclude that the real advantage of FASP over fuzzy SAT lies in problem domains that give rise to programs with loops. We illustrated this with a continuous version of the $k$-center problem. Moreover, this translation is important because it allows us to solve FASP programs using fuzzy SAT solvers. Under appropriate restrictions, for example, the satisfiability problems that are obtained can be solved using off-the-shelf mixed integer programming methods.

Although the results in this thesis provide the theoretical foundations for building an (A)FASP solver, some important issues still need to be tackled to build a real working system. One of the most important ones is optimizing the grounding of FASP programs. At first sight it seems that the grounding in FASP is more complex than in ASP, since atoms can be true to a certain degree, which leaves less room for removing rules. However, a promising technique is to combine reasoning over the upper and lower bounds of atoms with properties of the functions that are used. For example, if the upper bounds of $a$ and $b$ are 0.5, we know that the body of the rule $c \leftarrow T_W(a,b)$ is never greater than 0, meaning the rule can be removed. Investigating such optimizations thus seems to be an interesting and important path for future research in the implementation of FASP.

In conclusion, we have performed a thorough theoretical investigation of FASP
that is important from both a theoretical as a practical perspective. We contributed to the theory by introducing a new language called AFASP and by demonstrating that essential properties of ASP can be generalized to both FASP and AFASP. From a practical point of view we have shown that implementers of FASP can focus on a core language and can leverage the power of efficient fuzzy SAT solvers. Our results show that the combination of ASP with fuzzy logic results in a flexible domain-specific language that allows to solve continuous optimization problems in a concise and elegant manner without it becoming unwieldy to reason about.
Samenvatting

Door het alsmaar toenemende belang van de computer in onze maatschappij wordt de noodzaak voor talen die ons toelaten om snel foutloze programma’s te schrijven dringender dan ooit. Hoewel programmeertalen reeds ver geëvolueerd zijn sinds het tijdperk van de ponskaarten toont het grote aantal fouten in de huidige applicaties aan dat we dit doel nog niet bereikt hebben. Een nieuwe trend in de computerwetenschappen is om problemen op te lossen door middel van een taalgerichte aanpak. Het idee hier is dat in plaats van een programma te schrijven in een programmer-taal voor algemene doeleinden zoals Java of Haskell, we eerst een domeinspecifieke taal creëren die dicht aanleunt bij het probleem domein dat we willen modelleren en vervolgens het probleem beschrijven in deze nieuwe taal. Het voordeel van een dergelijke aanpak is dat programma’s veel sneller kunnen geschreven worden en dat het gemakkelijker wordt om verschillen te zien tussen de specificatie en de implementatie. Answer set programmeren (afk. ASP) is een dergelijke (declaratieve) domein-specifieke programmer-taal die zijn oorsprong kent in logisch programmeren en bijzonder geschikt is voor het beschrijven van combinatorische optimalisatieproblemen. Deze taal wordt ondermeer gebruikt om planningsproblemen op te lossen, configuraties te optimaliseren en om beslissingshulpmiddelen te modelleren. Verder doet ze ook dienst als een intermediaire taal voor andere domeinspecifieke talen, zoals deze gebruikt in de biologie.

Hoewel ASP een rijke taal is om combinatorische optimalisatieproblemen in te beschrijven, is ze niet geschikt om problemen met continue domeinen te beschrijven. Dergelijke problemen duiken echter op in heel diverse toepassingen, zoals bv. het ontwerp van gas- en elektriciteitsnetwerken, beeldherkenning en de optimalisatie van bedrijfsprocessen en investeringsportfolio’s. Om dit soort problemen toch te kunnen modelleren hebben we in deze thesis vaag answer set programmeren be-studeerd (Eng. “Fuzzy Answer Set Programming”, afk. FASP). Na een inleiding en een kort hoofdstuk dat enkele beginselen omtrent ASP en vaaglogica introduceert, beschrijven we FASP in Hoofdstuk 3.
Een belangrijk probleem wanneer we continue optimalisatieproblemen model-leren is hoe we problemen moeten behandelen die geen oplossingen hebben door een teveel aan opgelegde beperkingen. In veel gevallen kunnen we opteren om een imperfecte oplossing aan te nemen, m.a.w. een oplossing die niet voldoet aan alle gestelde regels (beperkingen). De vraag welk van deze oplossingen we dan moeten kiezen dringt zich echter op. De huidige aanpakken voor FASP implementeren deze benaderende oplossingen op een vrij beperkte manier, waarbij gebruikers geacht worden om expliciet te stellen in welke mate een regel op zijn minst moet voldaan zijn. Dit heeft echter als gevolg dat de gebruikers de juiste gewichten moeten gokken en laat niet toe om potentiële oplossingen te ordenen. In Hoofdstuk 4 stellen we geaggregeerd vaag answer set programmeren (Eng. “Aggregated Fuzzy Answer Set Programming”, afk. AFASP) voor om dit probleem op te lossen. Het basisidee is dat gebruikers de gewichten van elke regel niet zelf moeten vastleggen, maar we dynamische gewichten hebben en een aggregatorfunctie de totale geschiktheidswaarde laten bepalen. Onze aanpak breidt een eerder voorstel voor FASP met een aggrega- tor uit. In de eerste plaats koppelen we de aggregator los van de tralie die gebruikt wordt om de waarheidswaarden in het FASP programma te bepalen, wat resulteert in een taal die veel flexibeler is. Ten tweede baseren we onze semantiek op de fix- puntsemantiek voor FASP in plaats van de semantiek gebaseerd op ongefundeerde verzamelingen. Dit maakt dat we ons niet moeten beperken tot t-normen in de lichamen van regels en legt ook de relaties die er bestaan tussen verschillende FASP formalismen beter bloot. Verder lost deze semantiek ook een probleem op wanneer we niet-totaal geordende tralies gebruiken voor de waarheidswaarden. Ten laatste illustreren we het praktisch nut en de voordelen van de flexibiliteit van AFASP aan de hand van twee voorbeelden: een continue versie van het graafkleuringsprobleem en het probleem van het toekennen van “peer reviewers” aan artikels.

Hoewel de toevoeging van aggregatoren en andere taalconstructies voor FASP nuttig is voor programmeurs, hebben deze ook als gevolg dat de taal moeilijker wordt om te implementeren en om erover te redeneren. Daarom bestuderen we in Hoofdstuk 5 of het mogelijk is om een kerntaal te definiëren voor FASP die toelaat om veel FASP constructies te simuleren. We introduceren een taal, CFASP (Eng. “Core Fuzzy Answer Set Programming”) genaamd, die enkel bestaat uit niet-beperkende regels met monotoon stijgende functies en negatoren in de lichamen van regels. We tonen aan dat CFASP krachtig genoeg is om beperkingsregels, monotoon dalende functies, AFASP met bepaalde aggregatoren, S-implicatoren en een veralgemening van klassieke negatie in ASP te simuleren. Verder merken we op dat de simulaties van beperkende regels en klassieke negatie een grote overeenkomst
SAMENVATTING

hebben met hun simulaties in klassiek ASP, wat ons verder inzicht in de relaties tussen ASP en FASP verschaf.

Onze analyse is zowel vanuit theoretisch als vanuit praktisch standpunt belangrijk. Vanuit een theoretisch perspectief maakt CFASP redeneren over FASP eenvoudiger, maar behouden we door middel van onze simulaties het voordeel dat de resultaten krachtig genoeg zijn om ook te gelden voor FASP. Vanuit een praktisch perspectief tonen onze resultaten aan dat oplossingsproblemas voor FASP enkel CFASP hoeven te implementeren. Merk op dat het bestaan van deze simulaties niet betekent dat de gesimuleerde constructies nutteloos zijn: veel van onze simulaties zijn vrij groot en uitgebreid wanneer we ze vergelijken met de constructie die ze simuleren. Om van FASP een intuitive taal te maken die gemakkelijk modelleren toelaat zijn deze constructies dus van groot belang.

Vanuit de noodzaak om een implementatiemethode voor FASP te vinden gaan we in Hoofdstuk 6 op zoek naar een vertaling van FASP naar een vaag vervulbaarheidsprobleem. Een dergelijke vertaling levert ons enerzijds belangrijke theoretische inzichten op in de relaties die bestaan tussen FASP en vage vervulbaarheidsproblemen en anderzijds levert dit een praktische implementatiemethode voor FASP op. We introduceren het concept van de vervollediging van een programma en tonen aan dat voor programma’s zonder lussen de modellen van de vervollediging overeenkomen met de answer sets. Dit betekent tevens dat voor dergelijke programma’s FASP geen extra voordelen oplevert tegenover vage vervulbaarheidsproblemen, wat eveneens het geval is bij ASP en Booleaanse vervulbaarheidsproblemen.

Voor programma’s met lussen tonen we aan dat we deze kunnen reduceren naar een vaag vervulbaarheidsprobleem door het concept van lusformules van ASP te veralgemenen naar FASP. Aangezien deze vertaling niet zo voor de hand ligt is als voor programma’s zonder lussen, kunnen we concluderen dat het echte voordeel van FASP tegenover vage vervulbaarheidsproblemen dan ook ligt in probleemdomeinen die aanleiding geven tot programma’s met lussen. We illustreren dit aan de hand van een continue versie van het k-centrum probleem. Bovendien is deze vertaling belangrijk omdat ze ons toelaat om FASP programma’s op te lossen door middel van oplossingsproblemas voor vage vervulbaarheidsproblemen. Onder gepaste voorwaarden is het bijvoorbeeld mogelijk om de aldus bekomen vervulbaarheidsproblemen op te lossen door middel van kant-en-klare methodes voor gemengd-geheeltallige programmeringsproblemen.

Samenvattend kunnen we stellen dat we een diepgaand onderzoek verrichten naar FASP dat zowel vanuit een theoretisch als een praktisch standpunt belangrijk is. We verrijken de theorie van FASP door een nieuwe taal, AFASP genaamd, te
SAMENVATTING

introduceren en door aan te tonen dat essentiële kenmerken van ASP kunnen veralgemeend worden naar zowel FASP als AFASP. Vanuit een praktisch perspectief tonen we dat implementeerders van FASP zich kunnen concentreren op een kerntaal en de kracht van efficiënte oplossingsprogramma’s voor vage vervulbaarheidsproblemen kunnen gebruiken om FASP op te lossen. Onze resultaten tonen aan dat de combinatie van ASP met vaaglogica resulteert in een flexibele domein-specifieke taal die toelaat om continue optimalisatieproblemen op een compacte en elegante manier te beschrijven, zonder dat ze onhandelbaar wordt om over te redeneren.
List of Publications

1. A Core Language for Fuzzy Answer Set Programming. Jeroen Janssen, Steven Schockaert, Dirk Vermeir and Martine De Cock. Submitted to a journal listed in the Web of Science SCI - Science Citation Index.

2. Fuzzy Equilibrium Logic: Declarative Problem Solving in Continuous Domains. Steven Schockaert, Jeroen Janssen, Dirk Vermeir. Submitted to a journal listed in the Web of Science SCI - Science Citation Index.


8. Towards Possibilistic Fuzzy Answer Set Programming. Kim Bauters, Steven Schockaert, Jeroen Janssen, Dirk Vermeir and Martine De Cock. In the


8. Compiling Fuzzy Answer Set Programs to Fuzzy Propositional Theories. Jeroen Janssen, Stijn Heymans, Dirk Vermeir and Martine De Cock. In the Proceed-
Bibliography


[38] B. De Finetti. La logique de la probabilité. In Actes Congrès International de la Philosophy Scientifique, 1936.


BIBLIOGRAPHY


211
List of Symbols

**ASP**

$\Pi_P$ immediate consequence operator, page 30

$\text{comp}(P)$ The completion of $P$, page 38

$\text{gnd}_{U_P}(r)$ the grounding of rule $r$, page 28

$\text{gnd}(P)$ the grounding of program $P$, page 28

$B_P$ Herbrand base, page 28

$r_h$ head of rule $r$, page 27

$U_P$ Herbrand universe of $P$, page 28

$\Pi_P^+$ least fixpoint of $\Pi_P$, page 30

$\text{Lit}_A$ set of literals built from atoms in $A$, page 26

$I \models L$ literal set $L$ is true w.r.t. $I$, page 29

$I \models l$ literal $l$ is true w.r.t. $I$, page 29

$I \models r$ $r$ is satisfied by $I$, page 29

$P^I$ reduct of $P$ w.r.t. $I$, page 31

**Fuzzy Logic**

$(P, \leq)$ set $P$ with preorder $\leq$ on $P$, page 24

$\mathcal{T}_Z$ drastic t-norm, page 43
List of Symbols

$S_Z$ drastic t-conorm, page 43

gt$(x, y)$ body function: $gt(x, y) = 1$ if $x > y$ and 0 otherwise, page 122

$I$ implicator, page 44

$\inf A$ infimum of $A$, page 24

$\leq_P$ ordering of the preordered set $P = (P, \leq_P)$, page 24

$I_W$ Łukasiewicz implicator, page 47

$N_W$ standard negator, page 42

$max A$ maximum of $A$, page 24

$T_M$ minimum t-norm, page 42

$I_M$ Gödel implicator, page 47

$S_M$ maximum t-conorm, page 42

$\min A$ minimum of $A$, page 24

$N_M$ Gödel negator, page 42

$N$ negator, page 41

PC$(T)$ propositional calculus for $T$, page 48

$I_p$ Goguen implicator, page 47

$r^+_b$ atoms occurring as arguments in increasing positions of the expression $r_b$, page 111

$I_{T_W,N_W}$ Łukasiewicz implicator, page 45

$I_{T_M,N_W}$ Kleene-Dienes implicator, page 45

$I_{T_p,N_W}$ Reichenbach implicator, page 45

$I_{S,N}$ S-implicator induced by $S$ and $N$, page 45

$I_{T,N}$ S-implicator induced $T$ and $N$, page 45
List of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sup A$</td>
<td>supremum of $A$, page 24</td>
<td></td>
</tr>
<tr>
<td>$\text{supp}(A)$</td>
<td>$\text{supp}(A) = {x \mid x \in X, A(x) &gt; 0}$, page 41</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>t-conorm, page 42</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>t-norm, page 42</td>
<td></td>
</tr>
<tr>
<td>$A \subseteq B$</td>
<td>Zadeh inclusion of $A$ and $B$, page 41</td>
<td></td>
</tr>
<tr>
<td>$x \sqcap y$</td>
<td>infimum of $x$ and $y$, page 25</td>
<td></td>
</tr>
<tr>
<td>$x \sqcup y$</td>
<td>supremum of $x$ and $y$, page 25</td>
<td></td>
</tr>
</tbody>
</table>

**FASP**

- $r_b$ body of rule $r$, page 53
- $\text{comp}(P)$ completion of program $P$, page 159
- $\Pi_P$ immediate consequence operator of $P$, page 55
- $\text{gnd}(P)$ grounding of $P$, page 54
- $B_p$ Herbrand Base of FASP program $P$, page 54
- $B_r$ Herbrand base of rule $r$, page 53
- $r_h$ head of rule $r$, page 53
- $L_p$ lattice on which $P$ is defined, page 54
- $T_M(x,y)$ minimum t-norm, page 44
- $S_M(x,y)$ maximum t-conorm, page 44
- $I \cap J$ intersection of interpretations $I$ and $J$, page 170
- $I \subset J$ strict inclusion of two interpretations, page 55
- $I \subseteq J$ inclusion of two interpretations, page 55
- $P^l$ reduct of $P$ w.r.t. $I$, page 59
- $P_a$ set of rules with $a$ in the head, page 54
List of Symbols

$r : a \leftarrow \alpha$  A general rule, page 53

$\mathcal{R}^I$ reduct of $r$ w.r.t. $I$, page 59

**AFASP**

\(\mathcal{A}_P\) aggregator of AFASP program $P$, page 75

\(\mathcal{P}_P\) preorder for AFASP program $P$, page 75

\(\Pi_{P,\rho}\) immediate consequence operator, page 79

\(\mathcal{L}_P\) lattice for (A)FASP program $P$, page 75

$r : a \leftarrow I \alpha$ rule that is associated with the $I_W$ implicator, page 69

$r : a \leftarrow m \alpha$ rule that is associated with the $I_M$ implicator, page 69

$I_s(r,c)$ support of rule $r$ w.r.t. $I$ and $c$, page 78

$r : a \leftarrow p \alpha$ rule that is associated with the $I_P$ implicator, page 69

\(\mathcal{R}_P\) rule base of AFASP program $P$, page 75

\(\mathcal{I}_r\) residual implicator associated with rule $r$, page 69

\(\rho_I\) rule interpretation of $I$, page 70

\(\mathcal{T}_r\) t-norm associated with rule $r$, page 69

\(P_a\) set of rules with $a$ in their head, page 75
Index

1-tautology, 48
\(L\)-fuzzy set, 40
\textbf{BL}, 49
\text{CFASP}[^{\perp}], 127
\text{CFASP}[^{\perp}], 121
\text{CFASP}[^{\perp}], 129
\L\ukasiewicz
  implicator, 45, 47
  logic, 50
  negator, 42
  t-conorm, 44
  t-norm, 44
adjoint property, 46
\text{AFASP}, 68
\text{AFASP}[^{S}], 141
aggregator, 70
answer set
  \text{AFASP}, 77
  \text{ASP}, 31, 32
  \text{FASP}, 58, 60
Answer Set Programming, 26
paradigm, 33
answer set solver, 15
\text{ASP}, see Answer Set Programming
atom
  \text{ASP}, 26
  \text{FASP}, 52
basic logic, 49
biresiduum, 47
body
  \text{ASP}, 27
  \text{FASP}, 53
bounded, 24
bounded sum, 44
\text{CFASP}, 119
Clark's completion, 37
classical negation, 15, 26
complete lattice, 25
completion, 159
  \text{ASP}, 37
  \text{FASP}, 159
constant
  \text{ASP}, 26
  \text{FASP}, 52
constraint
  \text{ASP}, 27
  \text{FASP}, 53
divisibility, 47
drastic negator, 42
drastic t-conorm, 43
drastic t-norm, 43
dual image, 43
evaluation
  fuzzy formula, 48
extended literal
  \text{ASP}, 26
INDEX

CFASP, 120

fact
ASP, 27
FASP, 53
FASP, 52

fixpoint
least
  iterated computation, 25, 56

formula, 48

forward chaining, 55
fuzzy set, 41

Gödel
  implicator, 47
  logic, 50
  negator, 42

Galois connection, 46
Goguen
  implicator, 47

graph coloring
ASP, 33
FASP, 64

least element, 24

grounded, 26, 53

Herbrand base
AFASP, 75
ASP, 28
FASP
program, 54

Herbrand universe, 28

immediate consequence operator
AFASP, 79
ASP, 30
FASP, 55

implicator, 44
  Łukasiewicz, 45, 47
  Gödel, 47
  Goguen, 47
  Kleene-Dienes, 45

Reichenbach, 45
residual, 46
S-, 45

induced negator, 47
infimum, 24

interpretation
ASP, 29
total, 29
FASP, 54

intersection, 170

interval-locking rule, 127

involutive, 41

Kleene-Dienes implicator, 45

join, 25

Kleene-Dienes implicator, 45

label, 53

lattice, 25

least element, 24

literal
ASP, 26
consistent set of, 26
extended, 26

negation-as-failure, see naf-literal
CFASP
  extended, 120
lower bound, 24
maximum, 24
maximum t-conorm, 42, 44
meet, 25
minimal model
  ASP, 29
  FASP, 55
minimum, 24
minimum t-norm, 42, 44
model
  ASP, 29
  FASP, 55
fuzzy logic, 48
minimal
  ASP, 29
  FASP, 55
monotonic, 25

naf-literal, see negation-as-failure literal
negation
  as-failure, 26
  classical, 15, 26
  strong, 26
negation-as-failure literal
  ASP, 26
negator, 41
  induced, 47
normal rule, 27

order relation, 23

partially ordered set, 24
poset, 24
positive rule
  ASP, 27
predicate, 26, 52
preorder, 23
preordered set, 24
probabilistic set, 24
product
  logic, 50
product t-conorm, 44
product t-norm, 44
program
  AFASP, 75
  constraint-free, 76
  positive, 76
  simple, 76
  ASP, 27
  constraint-free, 28
  normal, 28
  positive, 28
  simple, 28
  CFASP, 120
  simple, 120
  FASP, 53
  constraint-free, 54
  positive, 54
  simple, 54
propositional calculus, 48

R-implicator, 46
Rational Pavelka Logic, 50
reduct
  ASP, 31
  FASP, 59
Reichenbach implicator, 45
residual implicator, 46
residuation principle, 46
RPL, 50
rule
  ASP, 27
INDEX

positive, 27
simple, 27
CFASP, 120
FASP, 53
    positive, 53
    simple, 53
interval-locking, 127
rule interpretation, 70
rule model, 76

S-implicator
    induced by t-conorm, 45
    induced by t-norm, 45
SAT, 36
satisfiable, 50
satisfied
    AFASP, 69
    ASP, 29
    FASP, 54
simple
    program
        AFASP, 76
simple rule
    ASP, 27
solver, 15
standard negator, 42
strong negation, 26
support, 41, 78
    s-implicator, 142
supremum, 24

t-conorm, 42
t-norm, 42
term
    ASP, 26
    FASP, 52
theory, 48

triangular conorm, 42
triangular norm, 42
unfounded set, 111
unfounded-free, 111
upper bound, 24
variable
    ASP, 26
    FASP, 52
Zadeh inclusion, 41