DIPLOMARBEIT

Disjunctive Datalog with Strong and Weak Constraints

Representational and Computational Issues

ausgeführt am

Institut für Informationssysteme E184/2

Abteilung für Datenbanken und Expertensysteme

der Technischen Universität Wien

unter Anleitung von

Univ.-Prof. Dott. Nicola Leone

und

Univ.-Ass. Dipl.-Ing. Gerald Pfeifer

als verantwortlich mitwirkendem Universitätsassistenten

durch

Wolfgang Faber
Hans-Kudlich Ring 21
2301 Groß-Enzersdorf

17. April 1998

"With the best will in the world,
we cannot always be completely
truthful or consistently rational"

Aldous Huxley

Dedicated to my parents

Acknowledgements

First of all, I would like to thank my supervisors, Nicola Leone and Gerald Pfeifer, for their genuine support and fruitful discussions. Special thanks to Gerald for his acribic (and time-consuming) proofreading.

A huge hug goes to Sonja Kovacic for her understanding and patience. In addition, I am grateful to all of the people who had to bear additional burdens because of my unavailibilities (cf. Section 3.2) during the creation of the thesis.

Gerald Kovacic has to be credited for the idea of including the example application of land use planning given in Section 3.3, thanks also to Andrea Weninger and Gernot Maierbrugger for subsequent discussions and suggestions on this topic.

Last but not least nothing of this work could have been accomplished without the support of my family, in particular of my parents.

Software Tools

The typesetting has been accomplished by LaTeX2$_\varepsilon$, using $\mathcal{AMS}$ extensions and BibTeX bibliography management. Diagrams and pictures have been drawn with `xfig`, text has been edited with `GNU emacs`. All work has been done on `Linux`, `FreeBSD`, and `Solaris` systems running `X11R6`.

Contents

Chapter 1

Introduction

Disjunctive Datalog (DATALOG$^{not,\vee}$) is considered as a very important tool for knowledge representation and common-sense reasoning tasks by researchers of the logic programming, database, and AI communities [EGM97, BG94, LMR92]. The necessity to extend the generic language DATALOG^{not} by disjunction is motivated by the ability to represent many important problems more naturally, and it has even been shown that DATALOG$^{not,\vee}$ can indeed express strictly more problems than its disjunction-free counterpart under reasonable semantics [EGM97] (unless the complexity classes P and NP coincide and thus the polynomial hierarchy collapses, which is generally believed not to be the case).

In addition to the inclusion of disjunction, the benefit of having a means to express *explicit* (or *true*) *negation* (as opposed to negation-as-failure) has been motivated [GL91]. Informally, explicit negation allows for specifying negative knowledge directly, whereas negation-as-failure can only derive negative information from the lack of contradictory positive knowledge.

Another extension of DATALOG$^{not,\vee}$ is adding *integrity constraints*, which are called *strong constraints* in our framework [BLR97b, BLR98]. While this concept is well-known in the domain of databases, it has only recently been proposed as an addition to DATALOG$^{not,\vee}$, albeit the representation is a natural generalisation of the syntactic structures of DATALOG$^{not,\vee}$.

The extensions described above allow for elegant representations of classical search and decision problems, but the representation of optimisation problems is usually still somewhat awkward. *Weak constraints* were introduced together with *strong constraints* in order to provide a means for specifying optimisation problems naturally and elegantly, while again just generalising the notion of a strong constraint in the sense that they do not have to be satisfied, but rather they should preferably be satisfied [BLR97b, BLR97a, BLR98]. If there is a choice between two different situations (possible worlds), where a weak constraint is satisfied in one of them but violated in the other, the former is preferred.

Apparently there are many situations, in which some weak constraints should be considered more important than others. For this reason, the notion of *priority levels* was introduced together with the notion of weak constraints [BLR97b, BLR97a, BLR98].

In this thesis, we unify the abovementioned extensions to DATALOG, and while doing this, we extend the notion of weak constraints by a novel means of

specifying different importance of constraints: *penalties* or *weights*, which allow for a more differentiated representation of precedence. We call the resulting language DATALOG$^{\neg,not,\vee,w}$, and we will give a formal definition of its syntax in Section 2.2.

We also introduce an *abstract syntax*, which is situated on a level between syntax and semantics. It is used to treat the different constructs in a unified way, thus defining equivalent programs on a syntactic level and easing the formal definition of semantics.

Concerning semantics, lots of different approaches have been published recently to capture the meaning of DATALOG$^{not,\vee}$ programs [Prz91, Ros90, Prz95, Dix95]. In spite of this babylonic situation, a generalisation of semantics can be given, which allows to speak about "an arbitrary semantics" of a DATALOG$^{not,\vee}$ program. We will define this general notion in Section 2.6, and after that we will focus on a particular semantics – the Stable Model Semantics. We will also define the meaning of programs containing true negation by a transforming them to programs containing only negation-as-failure. (Note that we need not define an extension to semantics in order to capture the meaning of strong constraints, since their meaning is already contained in the definition of abstract syntax.)

We will then give a definition for the meaning of weak constraints on top of an arbitrary semantics. We define the notion of *preferred models*, which are those models which violate the least number of constraints, taking into account the priority level and penalty information.

Comprising the second part of the thesis, we give several case studies of how the language DATALOG$^{\neg,not,\vee,w}$ can be used to represent problems of different domains, all of which have practical relevance, in a very natural way.

In particular, we show how several versions of abductive reasoning tasks can be accomplished using DATALOG$^{\neg,not,\vee,w}$ programs. It turns out that several ways of abductive logic programming correspond to the different kinds of weak constraints – *minimum cardinality abduction* corresponds to plain weak constraints, *priority minimal* abduction to weak constraints with priority layers, and *penalisation-based* abduction to weighted weak constraints.

Then we focus on how planning and scheduling problems can be represented. As an example we consider school timetabling and use a recent description ([CDM98]) of this problem, which makes heavy use of weak constraints involving both the priority layer and weight features.

As a conclusion to these case studies, we consider graph optimisation problems. We cover in depth the problem of finding a minimum spanning tree of undirected graphs, and also give representations for the problems of finding the minimum spanning tree of directed graphs and finding the minimum Steiner tree of an undirected graph, which can be obtained by simple extensions of the original representation of the minimum spanning tree problem.

The final part of the thesis deals with the implementation of the Preferred Stable Model Semantics. We extend an algorithm for computing stable models, which has originally been presented in [LRS97].

We first give an extension of the algorithm which efficiently treats strong constraints. Actually it enables us to abandon certain parts of the search space as soon as some constraint is violated by a partly computed model.

Then this approach is extended to weak constraints by defining a suitable objective function, which has to be minimised by all preferred models, and which is in turn minimal only for the preferred models. We use a greedy approach to

find exactly one preferred model. In many cases this algorithm works very efficiently and for most applications finding one model is sufficient. However, we give a method which computes all preferred models without raising the complexity considerably.

We argue about the complexity of the original algorithm which computes exactly one preferred model, conclude that it stays within the same complexity class as the original algorithm described in [LRS97] and on this basis we show that the extension to computing all preferred models is in this class, too.

Finally, we analyse how the changes to the algorithm can be integrated into the `dlv` system which has been developed and is currently enhanced at Institut für Informationssysteme of Technische Universität Wien [ELM+97a, CEF+97].

To summarise, the main contributions of this thesis are:

- We give a formal definition of the DATALOG$^{\neg,not,\vee,w}$ language, which contains different previously defined extensions, as well as the new notion of weighted weak constraints.
- We formally define the notion of *preferred models*, which assigns a meaning to DATALOG$^{\neg,not,\vee,w}$ programs.
- We analyse in depth how the DATALOG$^{\neg,not,\vee,w}$ language can be used to express various relevant problems.
- We give an algorithm which computes the preferred stable models of a given program efficiently.
- We describe how this algorithm can be effectively integrated into an existing system which is capable of computing stable models (and answer sets) for disjunctive datalog programs with strong constraints (and explicit negation).

Chapter 2

Language Definition

2.1 Roadmap

In this chapter, we will first define the syntax for the language DATALOG$^{\neg,not,\vee,w}$, i.e., disjunctive datalog with negation and strong and weak constraints and a notion of explicit negation as introduced by [GL91]. This will be done incrementally, and on the way to DATALOG$^{\neg,not,\vee,w}$, several other languages, which have already been described in the literature, will be presented (cf. Definition 2.2.19), namely:

- pure DATALOG
- DATALOG^{not}: DATALOG with negation-as-failure
- DATALOG$^{\vee}$: DATALOG with disjunction in the head
- DATALOG$^{not,\vee}$: DATALOG with negation-as-failure and disjunction
- DATALOG$^{not,\vee,s}$: DATALOG$^{not,\vee}$ with strong constraints
- DATALOG$^{not,\vee,c}$: DATALOG$^{not,\vee}$ with strong and weak constraints
- DATALOG$^{not,\vee,w}$: DATALOG$^{not,\vee}$ with strong and weighted weak constraints
- DATALOG$^{\neg}$: DATALOG enhanced by explicit negation
- DATALOG$^{\neg,not}$: DATALOG^{not} enhanced by explicit negation
- DATALOG$^{\neg,\vee}$: DATALOG$^{\vee}$ enhanced by explicit negation
- DATALOG$^{\neg,not,\vee}$: DATALOG$^{not,\vee}$ enhanced by explicit negation
- DATALOG$^{\neg,not,\vee,s}$: DATALOG$^{not,\vee,s}$ enhanced by explicit negation
- DATALOG$^{\neg,not,\vee,c}$: DATALOG$^{not,\vee,c}$ enhanced by explicit negation
- DATALOG$^{\neg,not,\vee,w}$: DATALOG$^{not,\vee,w}$ enhanced by explicit negation

Afterwards, a unifying "abstract syntax" will be defined, which will enable us to treat programs of all these languages in a unified framework and which will make certain low-level equivalences between programs explicit.

Last, a semantics of DATALOG$^{\neg,not,\vee,w}$ will be defined. We will first describe what semantics of DATALOG$^{\neg,not,\vee}$ look like and mention a few very important semantics. As DATALOG$^{\neg,not,\vee,w}$ is concerned, it will turn out that the language DATALOG$^{not,\vee,w}$ is best described using a "parametrised" semantics, which we will call *preferred model semantics*. By "parametrised" semantics we mean that the semantics will only be fixed for a certain part of the language, whereas for the remaining (well-researched) part any semantics can be chosen. In this thesis one particular choice for this parameter, the stable model semantics for DATALOG$^{not,\vee,s}$ and the computation of the preferred models under this semantics will be dealt with in detail.

Alternative definitions for syntax and semantics of DATALOG can be found in [CGT90] and [Ull89]; for DATALOG^{not}, DATALOG$^{\vee}$, and DATALOG$^{not,\vee}$ see [LMR92]; for DATALOG$^{not,\vee,s}$ and DATALOG$^{not,\vee,c}$, refer to [BLR97b, BLR98].

2.2 Syntax of DATALOG$^{not,\vee,w}$

2.2.1 Syntax Diversity

In the literature one can find lots of different syntactical definitions of DATALOG. Over the years, a Prolog-like syntax has been accepted by most of the researchers in this area. For instance, [CGT90] defines DATALOG syntax in a similar way as we will, while [LMR92] takes a more logic-based approach, similar to [Pfe96].

Since the definitions in this section are purely syntactical, we will give short informal hints about the meaning of the syntactical concepts before each definition to ease reading. After all, syntax definitions are made with semantics in mind. The formal semantical definitions will be given in section 2.6.

2.2.2 Notation

Strings and sets of strings will be the main notational elements in the syntax definition. We will use regular expressions, enhanced by the ellipsis (...) for the characterisation of strings. A brief description of these regular expressions follows:

Characters (i.e., string elements) are set in `typewriter font`, except for "$\leftarrow$", "$\Leftarrow$", "$\vee$", "$\neg$ ", ",", ".", and '(', ')' [1] , which are treated as characters, too, while variables representing strings are printed in *slanted font*.

A regular expression with ellipsis is one of the following forms (operator precedences correspond to the order of declaration below):

- A character `c` represents the string consisting only of the single character `c`
- [$\mathtt{c}_\mathtt{m} - \mathtt{c}_\mathtt{n}$] represents the strings consisting of one of the characters between $\mathtt{c}_\mathtt{m}$ and $\mathtt{c}_\mathtt{n}$, using the usual alphabetical or numerical ordering.

[1] To avoid ambiguity, '(' and ')', are always written between single quotes when used as characters in regular expressions.

- A string variable s represents the strings out of a specified set of strings
- $s_m\mathtt{c}\ldots\mathtt{c}s_n$ represents the concatenations of $abs(n-m)+1$ strings out of the specified set, separated by $\mathtt{c}$, where s_m to s_n are string variables defined over the same set of strings and $\mathtt{c}$ is some character.
- (r)[2], where r is a regular expression, represents the strings described by r.
- A regular expression followed by another regular expression: r_1r_2, represents the strings which are constructible by the concatenation of a string represented by r_1 and a string described by r_2.
- r^*, where r is a regular expression, represents an empty string, the string described by r, and the strings resulting of arbitrary many concatenations to itself
- $r_1|r_2$, where r_1 and r_2 are regular expressions, represent the strings described by r_1 or those represented by r_2.

$\mathbb{N}$ is the set of all non-negative integer numbers, $\mathbb{N}^+$the set of all positive integer numbers. $\mathbb{Z}$ is the set of all integer numbers.

2.2.3 Constants, Variables, Terms

The most basic part of our language is having some means to represent entities. An entity is anything which can be named. We call these names constants.

Definition 2.2.1 (Constants)
$\mathfrak{Constants}$ *is the set of all non-empty finite strings composed of letters, digits, and underscores, starting with a lower-case letter, plus the set of all non-empty, finite sequences of digits:*

$$\mathfrak{Constants} = \{([0-9][0-9]^*)|([\mathtt{a}-\mathtt{z}](\_|[\mathtt{a}-\mathtt{z}]|[\mathtt{A}-\mathtt{Z}]|[0-9])^*)\}$$

■

Example 2.2.1
x, *0815*, *planB*, *plan9fromOuterSpace*, *graph_1* $\in \mathfrak{Constants}$ ♦

We also need a way to make statements without naming concrete entities. That is, we need some names, which can represent any entity. These names are called variables.

Definition 2.2.2 (Variables)
$\mathfrak{Variables}$ *is the set of all non-empty finite strings composed of letters, digits, and underscores, starting with an upper-case letter:*

$$\mathfrak{Variables} = \{[\mathtt{A}-\mathtt{Z}](\_|[\mathtt{a}-\mathtt{z}]|[\mathtt{A}-\mathtt{Z}]|[0-9])^*\}$$

■

[2]Note that these brackets are not written between high commata.

Example 2.2.2
Placeholder, *X1*, *VAR*, *V_X_1* $\in \mathfrak{Variables}$ ♦

If we make statements about a particular entity, we can make the same statement about any entity. In other words, where a constant can appear, a variable may appear as well. The generalised concept of both is called term.

Definition 2.2.3 (Terms)
$\mathfrak{Terms}$ *is the set consisting of all constants and all variables:*

$$\mathfrak{Terms} = \mathfrak{Constants} \cup \mathfrak{Variables}$$

■

2.2.4 Predicates, Arities, Atoms, Literals

We do not just want to name entities, we want to make statements about them. Predicates represent these statements. In the context of database theory, these are also called relations.

Definition 2.2.4 (Predicates)
$\mathfrak{Predicates}$ *is the set of all non-empty finite strings composed of letters, digits, and underscores, starting with a lower-case letter:*[3]

$$\mathfrak{Predicates} = \{[\mathtt{a}-\mathtt{z}](\_|[\mathtt{a}-\mathtt{z}]|[\mathtt{A}-\mathtt{Z}]|[0-9])^*\}$$

■

Example 2.2.3
planB, *is_weird_film*, *predicate1*, *graph_1* $\in \mathfrak{Predicates}$ ♦

When we make statements about entities, the number of entities concerned in such a statement is called arity. When predicates are referred to, one usually specifies the predicate name followed by a slash and the corresponding arity.

Definition 2.2.5 (Arity)
`arity` *is a function, which associates a number to each predicate.*

$$\mathtt{arity} : \mathfrak{Predicates} \to \mathbb{N}$$

■

Atoms are the combination of predicates and terms. While predicates just give the means to make statements, atoms are used when we actually make these statements.

Definition 2.2.6 (Atoms)
$\mathfrak{Atoms}$ *is the set of all predicate symbols followed by a number of terms, which is determined by the predicate's arity. These terms are enclosed in parentheses and*

[3] Observe that syntactically $\mathfrak{Predicates} \subset \mathfrak{Constants}$; but as we shall see, the set-membership of a particular string is fully determined by the context.

separated by commata. The terms $t_1 \ldots t_n$ are called parameters (in database theory they are usually referred to as attributes).

$$\begin{aligned}\mathfrak{Atoms} =& \{p \mid p \in \mathfrak{Predicates}, \mathtt{arity}(p) = 0\} \cup \\ & \cup \{p'('t_1, \ldots, t_n')' \mid p \in \mathfrak{Predicates}, t_1, \ldots, t_n \in \mathfrak{Terms}, \\ & \qquad n \in \mathbb{N}^+, n = \mathtt{arity}(p)\}\end{aligned}$$

■

Example 2.2.4 (`is_weird_film/1`)
`is_weird_film(plan9fromOuterSpace)` $\in \mathfrak{Atoms}$
$\mathtt{arity}(\mathtt{is\_weird\_film}) = 1$ ♦

Sometimes we do not just want to state something positively, but we want to negate statements. We will present two flavours of negation: Explicit negation and negation-as-failure, this separation has been introduced in [GL91].

Intuitively, explicit negation (denoted as $\neg$) corresponds to stating that something is known not to be the case, whereas negation-as-failure (denoted as $\mathtt{not}$[4]) corresponds to stating that one does not have any contradicting knowledge of something.

We refer to an atom as literal when we know in which context – positive or negative – it occurs. We have several types of literals: explicitly negated literals, negation-as-failure literals, and a combination of these two. This combination is restricted to negation-as-failure of an explicitly negated literal. It makes sense to say that we do not have any contradicting knowledge that something is certainly not the case – but it does not make sense to say that we are sure about the falsity of the fact that we do not have any contradicting knowledge of *something*, because this is equivalent to being sure that *something* is true. Expressed differently, $\mathtt{not} \neg A \not\equiv A$, but $\neg\, \mathtt{not}\, A \equiv A$.

Definition 2.2.7 (Literals)
The explicitly negated literal set, $\mathfrak{Literals}^{\neg}$, is the set consisting of all atoms and all explicitly negated atoms.

$$\mathfrak{Literals}^{\neg} = \mathfrak{Atoms} \cup \{\neg\, a \mid a \in \mathfrak{Atoms}\}$$

The negation-as-failure literal set, $\mathfrak{Literals}^{\mathtt{not}}$ is the set consisting of all atoms and all negation-as-failure atoms.

$$\mathfrak{Literals}^{\mathtt{not}} = \mathfrak{Atoms} \cup \{\mathtt{not}\ a \mid a \in \mathfrak{Atoms}\}$$

The extended literal set, $\mathfrak{Literals}$, is the set consisting of all explicitly negated literals and all negation-as-failure explicitly negated literals.

$$\mathfrak{Literals} = \mathfrak{Literals}^{\neg} \cup \{\mathtt{not}\ l \mid l \in \mathfrak{Literals}^{\neg}\}$$

■

[4] in the literature also "$\sim$" or "`non`"

Definition 2.2.8
If $L \subseteq \mathfrak{Literals}^{\mathtt{not}}$, *let*

$$\begin{aligned} L^{+} &= L \cup \mathfrak{Atoms} \\ L^{-} &= L - \mathfrak{Atoms} \\ \mathtt{not}\ (L) &= \{\mathtt{not}\ a \mid a \in L^{+}\} \cup \{a \mid \mathtt{not}\ a \in L^{-}\} \end{aligned}$$

■

Example 2.2.5

$$\begin{aligned} \mathtt{naughty(john)} &\in \mathfrak{Literals}^{\neg} \cap \mathfrak{Literals}^{\mathtt{not}} \cap \mathfrak{Literals} \\ \mathtt{not\ naughty(john)} &\in \mathfrak{Literals}^{\mathtt{not}} \cap \mathfrak{Literals} \\ \neg\ \mathtt{naughty(Child)} &\in \mathfrak{Literals}^{\neg} \cap \mathfrak{Literals}, \\ \mathtt{not}\ \neg\ \mathtt{naughty(Child)} &\in \mathfrak{Literals} \end{aligned}$$

♦

2.2.5 Facts, Rules, Constraints

In the sequel several syntactic structures will be defined in two versions: one in which explicit negation is not permitted, as in most of the literature, and one in which explicit negation is allowed additionally. The latter version is referred to with the attribute "extended". Note that the extended structures are strict generalisations of the non-extended, since $\mathfrak{Atoms} \subset \mathfrak{Literals}^{\neg}$ and $\mathfrak{Literals}^{\mathtt{not}} \subset \mathfrak{Literals}$.

So far, we defined how to state something. Now we define how one says that some statement is true in its own right. Such statements are called facts.

Definition 2.2.9 (Facts)
$\mathfrak{Facts}$ *is the set of atoms followed by a full-stop.*

$$\mathfrak{Facts} = \{A. \mid A \in \mathfrak{Atoms}\}$$

The set of all extended facts is called $\mathfrak{Facts}^{\neg}$. $\mathfrak{Facts}^{\neg}$ *is the set comprised of explicitly negated literals followed by a full-stop.*

$$\mathfrak{Facts}^{\neg} = \{L. \mid L \in \mathfrak{Literals}^{\neg}\}$$

■

We might also want to state that at least one of several statements should be true.

Definition 2.2.10 (Disjunctive Facts)
$\mathfrak{Facts}^{\vee}$ *is the set of finite non-empty sequences of atoms, separated by* $\vee$[5], *followed by a full-stop.*

$$\mathfrak{Facts}^{\vee} = \{A_1 \vee \cdots \vee A_n. \mid A_1, \ldots, A_n \in \mathfrak{Atoms}, n \in \mathbb{N}^{+}\}$$

[5]written as "v"; in the literature, sometimes ";" is used (Prolog heritage)

The set of extended disjunctive facts, $\mathfrak{Facts}^{\neg,\vee}$ is comprised of finite sequences of explicitly negated literals, separated by $\vee$, followed by a full-stop.

$$\mathfrak{Facts}^{\neg,\vee} = \{L_1 \vee \cdots \vee L_n. \mid L_1, \ldots, L_n \in \mathfrak{Literals}^{\neg}, n \in \mathbb{N}^+\}$$

■

Example 2.2.6
Imagine Santa Claus[6]: He has to keep a database about all the children. For example if some child (here it is `john`*) has been naughty, he stores it in his database. Similarly, if he knew that* `john` *has not been naughty, he would store the second fact. If he is not sure yet, he stores the disjunctive facts in the third or fourth line, depending on whether his system supports explicit negation or not.*

$$\begin{aligned}
\texttt{naughty(john)}. &\in \mathfrak{Facts} \cap \mathfrak{Facts}^{\vee} \cap \mathfrak{Facts}^{\neg} \cap \mathfrak{Facts}^{\neg,\vee} \\
\neg\, \texttt{naughty(john)}. &\in \mathfrak{Facts}^{\neg} \cap \mathfrak{Facts}^{\neg,\vee} \\
\texttt{naughty(john)} \vee \texttt{nice(john)}. &\in \mathfrak{Facts}^{\vee} \cap \mathfrak{Facts}^{\neg,\vee} \\
\texttt{naughty(john)} \vee \neg\, \texttt{naughty(john)}. &\in \mathfrak{Facts}^{\neg,\vee}
\end{aligned}$$

⧫

We also want to be able to say that one statement is sure to hold, if we know that some other statements hold.

Definition 2.2.11 (Definite Rules)
$\mathfrak{Rules}$ is the set of structures composed of one atom followed by $\leftarrow$[7] and a sequence of atoms, separated by commata and terminated with a full-stop.

$$\mathfrak{Rules} = \{H \leftarrow B_1, \ldots, B_n. \mid H, B_1, \ldots, B_n \in \mathfrak{Atoms}, n \in \mathbb{N}\}$$

The set of extended rules, $\mathfrak{Rules}^{\neg}$, consists of elements composed of one explicitly negated literal followed by $\leftarrow$ and a sequence of explicitly negated literals, separated by commata and terminated with a full-stop.

$$\mathfrak{Rules}^{\neg} = \{H \leftarrow B_1, \ldots, B_n. \mid H, B_1, \ldots, B_n \in \mathfrak{Literals}^{\neg}, n \in \mathbb{N}\}$$

■

Example 2.2.7
Santa visits a child if it has been nice and also if it has not been naughty.

$$\begin{aligned}
\texttt{visit(C)} &\leftarrow \texttt{nice(C)}, \texttt{child(C)}. \in \mathfrak{Rules} \cap \mathfrak{Rules}^{\neg} \\
\texttt{visit(C)} &\leftarrow \neg\, \texttt{naughty(C)}, \texttt{child(C)}. \in \mathfrak{Rules}^{\neg}
\end{aligned}$$

⧫

Sometimes, one would like to say that a statement is sure to hold, if one knows that some statements hold and there is no evidence that some other statements hold.

[6] Do not take this too seriously; it should just illustrate how these syntactic forms look like.
[7] written as "`:-`"

Definition 2.2.12 (Normal Rules)
$\mathfrak{Rules}^{\mathfrak{not}}$ *is defined as* $\mathfrak{Rules}$*, but negation-as-failure literals rather than just atoms are allowed in the body.*

$$\mathfrak{Rules}^{\mathfrak{not}} = \{H \leftarrow B_1, \ldots, B_n. \mid H \in \mathfrak{Atoms}, B_1, \ldots, B_n \in \mathfrak{Literals}^{\mathfrak{not}}, n \in \mathbb{N}\}$$

$\mathfrak{Rules}^{\neg,\mathfrak{not}}$ *is defined as* $\mathfrak{Rules}^{\neg}$*, but general literals rather than just explicitly negated literals are allowed in the body.*

$$\mathfrak{Rules}^{\neg,\mathfrak{not}} = \{H \leftarrow B_1, \ldots, B_n. \mid H \in \mathfrak{Literals}^{\neg}, B_1, \ldots, B_n \in \mathfrak{Literals}, n \in \mathbb{N}\}$$

■

Example 2.2.8
Santa will visit a child if he has no information that it has been naughty. He will also visit a child if he has no reason to think that it has not been nice.

$$\texttt{visit(C)} \leftarrow \texttt{not naughty(C)}, \texttt{child(C)}. \in \mathfrak{Rules}^{\mathfrak{not}}$$
$$\texttt{visit(C)} \leftarrow \texttt{not} \neg \texttt{nice(C)}, \texttt{child(C)}. \in \mathfrak{Rules}^{\neg,\mathfrak{not}}$$

⧫

Now, one could want to say that at least one of several statements should hold, if some other statements hold.

Definition 2.2.13 (Positive Rules)
$\mathfrak{Rules}^{\vee}$ *is defined as* $\mathfrak{Rules}$*, but in the head a finite sequence of atoms, separated by* $\vee$*, called* **disjunction***, may occur.*

$$\mathfrak{Rules}^{\vee} = \{H_1 \vee \cdots \vee H_m \leftarrow B_1, \ldots, B_n. \mid H_1, \ldots, H_m, B_1, \ldots, B_n \in \mathfrak{Atoms}, m \in \mathbb{N}^+, n \in \mathbb{N}\}$$

$\mathfrak{Rules}^{\neg,\vee}$ *is defined as* $\mathfrak{Rules}^{\neg}$*, but in the head a finite sequence of explicitly negated literals, separated by* $\vee$*, may occur.*

$$\mathfrak{Rules}^{\neg,\vee} = \{H_1 \vee \cdots \vee H_m \leftarrow B_1, \ldots, B_n. \mid H_1, \ldots, H_m, B_1, \ldots, B_n \in \mathfrak{Literals}^{\neg}, m \in \mathbb{N}^+, n \in \mathbb{N}\}$$

■

Example 2.2.9
If a child has been naughty, Santa will bring her/him a tiny present if some is left (and probably only if the child was also nice sometimes), or he will skip the child's house (if he has to hurry or no parcels are left or the child has never been nice). Santa could use the second rule if he has a system supporting explicit negation.

$$\texttt{tiny\_present(C)} \vee \texttt{skip(C)} \leftarrow \texttt{naughty(C)}, \texttt{child(C)}. \in \mathfrak{Rules}^{\vee}$$
$$\texttt{tiny\_present(C)} \vee \neg\, \texttt{visit(C)} \leftarrow \neg\, \texttt{nice(C)}, \texttt{child(C)}. \in \mathfrak{Rules}^{\neg,\vee}$$

⧫

It is now straightforward to combine Definition 2.2.12 and Definition 2.2.13.

Definition 2.2.14 (Rules)
$\mathfrak{Rules}^{\mathtt{not},\vee}$ *is defined as* $\mathfrak{Rules}$*, but literals rather than just atoms are allowed in the body, and in the head a finite sequence of atoms, separated by* $\vee$*, may occur.*

$$\mathfrak{Rules}^{\mathtt{not},\vee} = \{H_1 \vee \cdots \vee H_m \leftarrow B_1, \ldots, B_n. \mid H_1, \ldots, H_m \in \mathfrak{Atoms}, \\ B_1, \ldots, B_n \in \mathfrak{Literals}^{\mathtt{not}}, \\ m \in \mathbb{N}^+, n \in \mathbb{N}\}$$

$\mathfrak{Rules}^{\neg,\mathtt{not},\vee}$ *is defined as* $\mathfrak{Rules}^{\neg}$*, but arbitrary literals rather than just explicitly negated literals are allowed in the body, and in the head a finite sequence of explicitly negated literals, separated by* $\vee$*, may occur.*

$$\mathfrak{Rules}^{\neg,\mathtt{not},\vee} = \{H_1 \vee \cdots \vee H_m \leftarrow B_1, \ldots, B_n. \mid H_1, \ldots, H_m \in \mathfrak{Literals}^{\neg}, \\ B_1, \ldots, B_n \in \mathfrak{Literals}, \\ m \in \mathbb{N}^+, n \in \mathbb{N}\}$$

■

Example 2.2.10
If there is no reason to assume that a child has been naughty, Santa will bring him or her a medium or big present, depending on his finances and how nice the child was. Also, if there is no reason to think that a child has not been nice, Santa will bring him or her a medium present or at least he will not skip the house.

$\mathtt{med\_present(C)} \vee \mathtt{big\_present(C)} \leftarrow \mathtt{not\ naughty(C), child(C).} \in \mathfrak{Rules}^{\mathtt{not},\vee}$

$\mathtt{med\_present(C)} \vee \neg\, \mathtt{skip(C)} \leftarrow \mathtt{not}\ \neg\, \mathtt{nice(C), child(C).} \in \mathfrak{Rules}^{\neg,\mathtt{not},\vee}$

⧫

Often, one wants to constrain (or restrict) the worlds as described by facts and rules, which means expressing that some statements should not be valid simultaneously. This concept is usually termed "integrity constraint". We will refer to "integrity constraints" as "constraints" or "strong constraints".

Definition 2.2.15 (Strong Constraints)
$\mathfrak{Constraints}_{\mathtt{strong}}$ *is the set of finite non-empty sequences of negation-as-failure literals, preceded by* $\leftarrow$*, separated by commata, and terminated by a full-stop.*

$$\mathfrak{Constraints}_{\mathtt{strong}} = \{\leftarrow B_1, \ldots, B_n. \mid B_1, \ldots, B_n \in \mathfrak{Literals}^{\mathtt{not}}, n \in \mathbb{N}^+\}$$

$\mathfrak{Constraints}^{\neg}_{\mathtt{strong}}$ *is the set of finite non-empty sequences of arbitrary literals, preceded by* $\leftarrow$*, separated by commata, and terminated by a full-stop.*

$$\mathfrak{Constraints}^{\neg}_{\mathtt{strong}} = \{\leftarrow B_1, \ldots, B_n. \mid B_1, \ldots, B_n \in \mathfrak{Literals}, n \in \mathbb{N}^+\}$$

■

Example 2.2.11
It can never be the case that Santa visits a child when he skips its house. It is not feasible that Santa has no reason to think that a child has been naughty but will skip its house. It can not occur that a child has been naughty and not nice and Santa will visit him or her. It is impossible that a child has been naughty and not nice and there is no evidence that Santa will not visit him or her, since it is sure that he will not come.

$$\leftarrow \texttt{skip}(\texttt{C}), \texttt{visit}(\texttt{C}). \in \mathfrak{Constraints}_{\mathfrak{strong}}$$
$$\leftarrow \texttt{not naughty}(\texttt{C}), \texttt{skip}(\texttt{C}). \in \mathfrak{Constraints}_{\mathfrak{strong}}$$
$$\leftarrow \texttt{naughty}(\texttt{C}), \neg\, \texttt{nice}(\texttt{C}), \texttt{visit}(\texttt{C}). \in \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}}$$
$$\leftarrow \texttt{naughty}(\texttt{C}), \neg\, \texttt{nice}(\texttt{C}), \texttt{not} \neg\, \texttt{visit}(\texttt{C}). \in \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}}$$

♦

The concept just defined can be generalised by stating that some statements should preferably not be valid simultaneously. These are called weak (sometimes also soft) constraints.

Definition 2.2.16 (Weak Constraints)
$\mathfrak{Constraints}_{\mathfrak{weak}}$ *is the set of finite non-empty sequences of negation-as-failure literals, preceded by* $\Leftarrow$[8]*, separated by commata, and terminated by a full-stop.*

$$\mathfrak{Constraints}_{\mathfrak{weak}} = \{\Leftarrow B_1, \ldots, B_n. \mid B_1, \ldots, B_n \in \mathfrak{Literals}^{\mathfrak{not}}, n \in \mathbb{N}^+\}$$

$\mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$ *is the set of finite sequences of general literals, preceded by* $\Leftarrow$*, separated by commata, and terminated by a full-stop.*

$$\mathfrak{Constraints}^{\neg}_{\mathfrak{weak}} = \{\Leftarrow B_1, \ldots, B_n. \mid B_1, \ldots, B_n \in \mathfrak{Literals}, n \in \mathbb{N}^+\}$$

■

Example 2.2.12
It is not desirable that a child gets a big present if there is no reason to think that it has been nice. It should possibly not be the case that a child has been nice and Santa has no reason to assume that he will not skip its house, since he should go there.

$$\Leftarrow \texttt{not nice}(\texttt{C}), \texttt{big\_present}(\texttt{C}). \in \mathfrak{Constraints}_{\mathfrak{weak}}$$
$$\Leftarrow \texttt{not} \neg\, \texttt{skip}(\texttt{C}), \texttt{nice}(\texttt{C}). \in \mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$$

♦

In real life, there are weak constraints which are more important to be met than others. To model this, we introduce layers of importance (or priorities), where any single weak constraint in some layer will be more important than all weak constraints in lower layers.

[8] denoted by "$:\sim$"

Definition 2.2.17 (Layered (Weak) Constraints)
$\mathfrak{Constraints}_{\mathfrak{layered}}$ *is the set of constraints found in* $\mathfrak{Constraints}_{\mathfrak{weak}}$*, possibly followed by a pair of square brackets, which enclose a positive integer followed by a colon:*

$$\mathfrak{Constraints}_{\mathfrak{layered}} = \{W[l:] \mid l \in \mathbb{N}^+, W \in \mathfrak{Constraints}_{\mathfrak{weak}}\} \cup \mathfrak{Constraints}_{\mathfrak{weak}}$$

$\mathfrak{Constraints}^{\neg}_{\mathfrak{layered}}$ *is the set of constraints found in* $\mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$*, possibly followed by a pair of square brackets, which enclose an integer followed by a colon:*

$$\mathfrak{Constraints}^{\neg}_{\mathfrak{layered}} = \{W[l:] \mid l \in \mathbb{N}^+, W \in \mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}\} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$$

■

Example 2.2.13
In Example 2.2.12, the second constraint is more important than the first, so Santa introduces two layers, such that the constraint which is more important resides in the higher priority layer.

$$\Leftarrow \texttt{not nice(C)}, \texttt{big\_present(C)}.[1:] \in \mathfrak{Constraints}_{\mathfrak{weak}}$$
$$\Leftarrow \texttt{not} \neg \texttt{skip(C)}, \texttt{nice(C)}.[2:] \in \mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$$

♦

In many problems, most notably problems defined over graphs, one wants to formulate weak constraints which are adorned by a weight, to be able to define precedences of weak constraints in a more differentiated way. The combination of weighted and layered weak constraints often leads to very elegant formulations of such problems.

Definition 2.2.18 (Weighted (Weak) Constraints)
$\mathfrak{Constraints}_{\mathfrak{weighted}}$ *is the set of constraints of* $\mathfrak{Constraints}_{\mathfrak{weak}}$*, possibly followed by a pair of square brackets, which enclose two integers separated by a colon, where either of the integers may be left out, but not both of them:*

$$\begin{aligned}\mathfrak{Constraints}_{\mathfrak{weighted}} =& \{W[l:w], W[:w] \mid l \in \mathbb{N}^+, w \in \mathbb{Z}, W \in \mathfrak{Constraints}_{\mathfrak{weak}}\} \\ & \cup \mathfrak{Constraints}_{\mathfrak{layered}}\end{aligned}$$

$\mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}}$ *is the set of constraints of* $\mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}$*, possibly followed by a pair of square brackets, which enclose two integers separated by a colon, where either of the integers may be left out, but not both of them:*

$$\begin{aligned}\mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}} =& \{W[l:w], W[:w] \mid l \in \mathbb{N}^+, w \in \mathbb{Z}, W \in \mathfrak{Constraints}^{\neg}_{\mathfrak{weak}}\} \\ & \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{layered}}\end{aligned}$$

■

Example 2.2.14
So, additionally, Santa thinks that it is good to assign a high weight to the important constraint, because he plans to have other constraints in the second layer, too. In contrast, the first one is not important to him at all, so he assigns a neutral weight of 0 to it.

$$\Leftarrow \texttt{not nice(C)}, \texttt{big\_present(C)}.[1:0] \in \mathfrak{Constraints}_{\mathfrak{weighted}}$$
$$\Leftarrow \texttt{not } \neg \texttt{ skip(C)}, \texttt{nice(C)}.[2:100] \in \mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}}$$

♦

Observation 1 (Relations between facts, rules, and constraints)
Note the following relations between the sets which have just been defined:

$$\mathfrak{Facts} \subset \mathfrak{Facts}^{\vee}$$
$$\mathfrak{Rules} \subset \mathfrak{Rules}^{\mathfrak{not}} \subset \mathfrak{Rules}^{\mathfrak{not},\vee}$$
$$\mathfrak{Rules} \subset \mathfrak{Rules}^{\vee} \subset \mathfrak{Rules}^{\mathfrak{not},\vee}$$
$$\mathfrak{Constraints}_{\mathfrak{weak}} \subset \mathfrak{Constraints}_{\mathfrak{layered}} \subset \mathfrak{Constraints}_{\mathfrak{weighted}}$$

The same holds for the extended versions:

$$\mathfrak{Facts}^{\neg} \subset \mathfrak{Facts}^{\neg,\vee}$$
$$\mathfrak{Rules}^{\neg} \subset \mathfrak{Rules}^{\neg,\mathfrak{not}} \subset \mathfrak{Rules}^{\neg,\mathfrak{not},\vee}$$
$$\mathfrak{Rules}^{\neg} \subset \mathfrak{Rules}^{\neg,\vee} \subset \mathfrak{Rules}^{\neg,\mathfrak{not},\vee}$$
$$\mathfrak{Constraints}^{\neg}_{\mathfrak{weak}} \subset \mathfrak{Constraints}^{\neg}_{\mathfrak{layered}} \subset \mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}}$$

We have already mentioned that the extended version of a syntactic form strictly contains the non-extended form. ★

2.2.6 Languages and Programs

At this point we are ready to define the languages already mentioned in section 2.1, and programs thereof.

Definition 2.2.19 (DATALOG languages)

$$\begin{aligned}
\text{DATALOG} &= \mathfrak{Facts} \cup \mathfrak{Rules} \\
\text{DATALOG}^{not} &= \mathfrak{Facts} \cup \mathfrak{Rules}^{\mathfrak{not}} \\
\text{DATALOG}^{\vee} &= \mathfrak{Facts}^{\vee} \cup \mathfrak{Rules}^{\vee} \\
\text{DATALOG}^{not,\vee} &= \mathfrak{Facts}^{\vee} \cup \mathfrak{Rules}^{\mathfrak{not},\vee} \\
\text{DATALOG}^{not,\vee,s} &= \mathfrak{Facts}^{\vee} \cup \mathfrak{Rules}^{\mathfrak{not},\vee} \cup \mathfrak{Constraints}_{\mathfrak{strong}} \\
\text{DATALOG}^{not,\vee,c} &= \mathfrak{Facts}^{\vee} \cup \mathfrak{Rules}^{\mathfrak{not},\vee} \cup \mathfrak{Constraints}_{\mathfrak{strong}} \cup \mathfrak{Constraints}_{\mathfrak{layered}} \\
\text{DATALOG}^{not,\vee,w} &= \mathfrak{Facts}^{\vee} \cup \mathfrak{Rules}^{\mathfrak{not},\vee} \cup \mathfrak{Constraints}_{\mathfrak{strong}} \cup \mathfrak{Constraints}_{\mathfrak{weighted}}
\end{aligned}$$

$$
\begin{aligned}
\text{DATALOG}^{\neg} &= \mathfrak{Facts}^{\neg} \cup \mathfrak{Rules}^{\neg} \\
\text{DATALOG}^{\neg,not} &= \mathfrak{Facts}^{\neg} \cup \mathfrak{Rules}^{\neg,\mathfrak{not}} \\
\text{DATALOG}^{\neg,\vee} &= \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\vee} \\
\text{DATALOG}^{\neg,not,\vee} &= \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\mathfrak{not},\vee} \\
\text{DATALOG}^{\neg,not,\vee,s} &= \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\mathfrak{not},\vee} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}} \\
\text{DATALOG}^{\neg,not,\vee,c} &= \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\mathfrak{not},\vee} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{layered}} \\
\text{DATALOG}^{\neg,not,\vee,w} &= \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\mathfrak{not},\vee} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}} \cup \mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}}
\end{aligned}
$$

■

We could also define (extended) DATALOG with negation, disjunction, strong and simple weak constraints, but since in the literature weak constraints appear only together with layers (with different syntax, though), we do not give any definition here, either.

While a DATALOG language without constraints describes all possible facts and rules, a corresponding program consists of some (i.e., finitely many) concrete facts and rules.

Definition 2.2.20 (DATALOG programs without constraints)

$$
\begin{aligned}
\Pi_{\text{DATALOG}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}\} \\
\Pi_{\text{DATALOG}^{not}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{not}\} \\
\Pi_{\text{DATALOG}^{\vee}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\vee}\} \\
\Pi_{\text{DATALOG}^{not,\vee}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{not,\vee}\}
\end{aligned}
$$

$$
\begin{aligned}
\Pi_{\text{DATALOG}^{\neg}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg}\} \\
\Pi_{\text{DATALOG}^{\neg,not}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,not}\} \\
\Pi_{\text{DATALOG}^{\neg,\vee}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,\vee}\} \\
\Pi_{\text{DATALOG}^{\neg,not,\vee}} &= \{\mathcal{P} \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,not,\vee}\}
\end{aligned}
$$

■

When speaking about programs with constraints, it is useful to distinguish between facts and rules, strong constraints, weak constraints (if they are in the corresponding language). Therefore they are usually represented as a pair or triple.

Definition 2.2.21 (DATALOG **programs with constraints**)

$$\begin{aligned}
\Pi_{\text{DATALOG}^{not,\vee,s}} &= \{\langle \mathcal{P}, \mathcal{S}\rangle \quad | \; \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}_{\mathfrak{strong}}\} \\
\Pi_{\text{DATALOG}^{not,\vee,c}} &= \{\langle \mathcal{P}, \mathcal{S}, \mathcal{W}\rangle \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}_{\mathfrak{strong}}, \\
&\qquad\qquad \mathcal{W} \underset{finite}{\subset} \mathfrak{Constraints}_{\mathfrak{layered}}\} \\
\Pi_{\text{DATALOG}^{not,\vee,w}} &= \{\langle \mathcal{P}, \mathcal{S}, \mathcal{W}\rangle \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}_{\mathfrak{strong}}, \\
&\qquad\qquad \mathcal{W} \underset{finite}{\subset} \mathfrak{Constraints}_{\mathfrak{weighted}}\}
\end{aligned}$$

$$\begin{aligned}
\Pi_{\text{DATALOG}^{\neg,not,\vee,s}} &= \{\langle \mathcal{P}, \mathcal{S}\rangle \quad | \; \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}}\} \\
\Pi_{\text{DATALOG}^{\neg,not,\vee,c}} &= \{\langle \mathcal{P}, \mathcal{S}, \mathcal{W}\rangle \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}}, \\
&\qquad\qquad \mathcal{W} \underset{finite}{\subset} \mathfrak{Constraints}^{\neg}_{\mathfrak{layered}}\} \\
\Pi_{\text{DATALOG}^{\neg,not,\vee,w}} &= \{\langle \mathcal{P}, \mathcal{S}, \mathcal{W}\rangle \mid \mathcal{P} \underset{finite}{\subset} \text{DATALOG}^{\neg,not,\vee}, \\
&\qquad\qquad \mathcal{S} \underset{finite}{\subset} \mathfrak{Constraints}^{\neg}_{\mathfrak{strong}}, \\
&\qquad\qquad \mathcal{W} \underset{finite}{\subset} \mathfrak{Constraints}^{\neg}_{\mathfrak{weighted}}\}
\end{aligned}$$

■

Observation 2 (Relations between languages and programs)
The following relations between languages, resp. sets of programs hold:

$$\text{DATALOG} \subset \text{DATALOG}^{not} \subset \text{DATALOG}^{not,\vee}$$
$$\text{DATALOG} \subset \text{DATALOG}^{\vee} \subset \text{DATALOG}^{not,\vee}$$
$$\text{DATALOG}^{not,\vee} \subset \text{DATALOG}^{not,\vee,s} \subset \text{DATALOG}^{not,\vee,c} \subset \text{DATALOG}^{not,\vee,w}$$
$$\Pi_{\text{DATALOG}} \subset \Pi_{\text{DATALOG}^{not}} \subset \Pi_{\text{DATALOG}^{not,\vee}}$$
$$\Pi_{\text{DATALOG}} \subset \Pi_{\text{DATALOG}^{\vee}} \subset \Pi_{\text{DATALOG}^{not,\vee}}$$
$$\Pi_{\text{DATALOG}^{not,\vee,c}} \subset \Pi_{\text{DATALOG}^{not,\vee,w}}$$

Analogously for the corresponding extended languages and programs:

$$\text{DATALOG}^{\neg} \subset \text{DATALOG}^{\neg,not} \subset \text{DATALOG}^{\neg,not,\vee}$$
$$\text{DATALOG}^{\neg} \subset \text{DATALOG}^{\neg,\vee} \subset \text{DATALOG}^{\neg,not,\vee}$$
$$\text{DATALOG}^{\neg,not,\vee} \subset \text{DATALOG}^{\neg,not,\vee,s} \subset \text{DATALOG}^{\neg,not,\vee,c} \subset \text{DATALOG}^{\neg,not,\vee,w}$$
$$\Pi_{\text{DATALOG}^{\neg}} \subset \Pi_{\text{DATALOG}^{\neg,not}} \subset \Pi_{\text{DATALOG}^{\neg,not,\vee}}$$
$$\Pi_{\text{DATALOG}^{\neg}} \subset \Pi_{\text{DATALOG}^{\neg,\vee}} \subset \Pi_{\text{DATALOG}^{\neg,not,\vee}}$$
$$\Pi_{\text{DATALOG}^{\neg,not,\vee,c}} \subset \Pi_{\text{DATALOG}^{\neg,not,\vee,w}}$$

Again, the extended languages and programs strictly contain their corresponding non-extended counterparts.

★

2.3 Abstract Syntax

In the preceding section we have defined the various DATALOG languages in a very basic way, without mangling with semantical aspects at all, except for the informal motivations we gave.

In this section we will describe abstractions over the syntactical constructs, which already capture some semantic equivalences (these are just in the scope of single rules, facts, etc., whereas a "real" semantics deals with the interaction of these concepts), and ease semantics descriptions in Section 2.6, since abstract syntax unifies the numerous types of programs.

2.3.1 Abstract Constructs

Although $\mathfrak{Facts}$, $\mathfrak{Facts}^{\vee}$, $\mathfrak{Rules}$, $\mathfrak{Rules}^{\vee}$, $\mathfrak{Rules}^{\mathfrak{not}}$, $\mathfrak{Rules}^{\mathfrak{not},\vee}$, and $\mathfrak{Constraints}_{\mathfrak{strong}}$ were introduced as distinct syntactical constructs in Section 2.2, they all correspond to one logical construct: a clause [Llo87]. This is reflected in Definition 2.3.1.

Definition 2.3.1 (Abstract Rule)
Abstract rules are pairs of sets: The first set of such a pair, called head, is a set of explicitly negated atoms, the second, called Body, is a set of general literals.

$$\mathit{Rules}_{\mathit{abstract}} = \{(H, B) \mid H \subseteq \mathfrak{Literals}^{\neg}, B \subseteq \mathfrak{Literals}\}$$

■

Weak constraints are different. They do not have a corresponding construct in classical logic, since in classical logic only absolute truth is considered, whereas weak constraints are a means to define statements which should **possibly** be true.

Definition 2.3.2 (Abstract (Weak) Constraint)
Abstract (weak) constraints consist of a set of literals, one natural number (representing the layer), and one integer number (the weight).

$$\mathit{Constraints}_{\mathit{abstract}} = \{(B, l, w) \mid B \subseteq \mathfrak{Literals}, l \in \mathbb{N}^{+}, w \in \mathbb{Z}\}$$

■

Of course, our aim is dealing with programs over the abstractions which we have just defined. We create a notion of the abstract program.

Definition 2.3.3 (Abstract Programs)
Abstract programs are finite sets consisting of abstract rules and abstract (weak) constraints.

$$\mathit{Programs}_{\mathit{abstract}} = \{\mathcal{P} \mid \mathcal{P} \underset{\mathit{finite}}{\subset} (\mathit{Rules}_{\mathit{abstract}} \cup \mathit{Constraints}_{\mathit{abstract}})\}$$

■

2.3.2 From Concrete to Abstract Syntax

Now we give a translation from the merely syntactical constructs into these unifying abstract concepts. Note that lots of equivalences between facts and rules, which were stated verbally in most of the literature, are made explicit by this translation. E.g. "The order of the literals (resp. atoms) in the body or head is not significant.", "A fact can be seen as a rule with an empty conjunction as its body.", "A strong constraint can be seen as a rule with an empty disjunction (which represents falsity) as head.". Also note that because of Observation 1, it is not necessary to define a translation for each type of facts and rules.

Definition 2.3.4 (Projection into Abstract Rules)

$$\mathtt{abstract}_{\mathit{rules}} : \mathfrak{Facts}^{\neg,\vee} \cup \mathfrak{Rules}^{\neg,\mathtt{not},\vee} \cup \mathfrak{Constraints}^{\neg}_{\mathtt{strong}} \to \mathit{Rules}_{\mathit{abstract}}$$

$$\mathtt{abstract}_{\mathit{rules}}(R) = \begin{cases} (\{H_1, \ldots, H_m\}, \emptyset) & \text{if } R = H_1 \vee \cdots \vee H_m. \\ (\{H_1, \ldots, H_m\}, \{B_1, \ldots, B_n\}) & \text{if } R = H_1, \ldots, H_m \leftarrow B_1, \ldots, B_n. \\ (\emptyset, \{B_1, \ldots, B_n\}) & \text{if } R = \leftarrow B_1, \ldots, B_n. \end{cases}$$

■

In the case of weak constraints, essentially we define the default value to be 1 if the layer information is missing and 0 if no weight is specified. Note that this is de facto semantics on a very low level. Also note that again we need not define a function for every type of weak constraints because of Observation 1.

Definition 2.3.5 (Projection into Abstract (Weak) Constraints)

$$\mathtt{abstract}_{\mathit{constraints}} : \mathfrak{Constraints}^{\neg}_{\mathtt{weighted}} \to \mathit{Constraints}_{\mathit{abstract}}$$

$$\mathtt{abstract}_{\mathit{constraints}}(\mathcal{W}) = \begin{cases} (\{L_1, \ldots, L_n\}, 1, 0) & \text{if } \mathcal{W} = \Leftarrow L_1, \ldots, L_n.) \\ (\{L_1, \ldots, L_n\}, l, 0) & \text{if } \mathcal{W} = \Leftarrow L_1, \ldots, L_n.[l :] \\ (\{L_1, \ldots, L_n\}, 1, w) & \text{if } \mathcal{W} = \Leftarrow L_1, \ldots, L_n.[: w] \\ (\{L_1, \ldots, L_n\}, l, w) & \text{if } \mathcal{W} = \Leftarrow L_1, \ldots, L_n.[l : w] \end{cases}$$

■

We now generalise Definitions 2.3.4 and 2.3.5 to programs. The definition covers all programs we have defined in Definition 2.2.20 and Definition 2.2.21, because of Observation 2.

Definition 2.3.6 (Projections into Abstract Programs)

$$\texttt{abstract} : \Pi_{\text{DATALOG}^{\neg,not,\vee}} \cup \Pi_{\text{DATALOG}^{\neg,not,\vee,s}} \cup \Pi_{\text{DATALOG}^{\neg,not,\vee,w}} \rightarrow \mathit{Programs}_{\mathit{abstract}}$$

$$\texttt{abstract}(\mathcal{P}) = \begin{cases} \{\texttt{abstract}_{\mathit{rules}}(R) \mid R \in \mathcal{P}\} & \text{if } \mathcal{P} \in \Pi_{\text{DATALOG}^{\neg,not,\vee}} \\ \{\texttt{abstract}_{\mathit{rules}}(R) \mid R \in \mathcal{P}_1 \cup \mathcal{S}\} & \text{if } \mathcal{P} = \langle \mathcal{P}_1, \mathcal{S} \rangle \\ \{\texttt{abstract}_{\mathit{rules}}(R) \mid R \in \mathcal{P}_1 \cup \mathcal{S}\} \cup & \\ \quad \cup \{\texttt{abstract}_{\mathit{constraints}}(W) \mid W \in \mathcal{W}\} & \text{if } \mathcal{P} = \langle \mathcal{P}_1, \mathcal{S}, \mathcal{W} \rangle \end{cases}$$

■

Example 2.3.1
We consider some tiny part of Santa's DATALOG$^{not,\vee,w}$ *program:*

$$\begin{aligned} \mathcal{P}^1_{Santa} = \{&\texttt{naughty(john)} \vee \neg\, \texttt{naughty(john)}., \\ &\texttt{med\_present(C)} \vee \neg\, \texttt{skip(C)} \leftarrow \texttt{not} \neg\, \texttt{nice(C)}, \texttt{child(C)}. \\ &\leftarrow \texttt{not naughty(C)}, \texttt{skip(C)}. \\ &\Leftarrow \texttt{not} \neg\, \texttt{skip(C)}, \texttt{nice(C)}.[2:100]\} \end{aligned}$$

The abstract program is:

$$\begin{aligned} \texttt{abstract}(\mathcal{P}^1_{Santa}) = \{&(\{\texttt{naughty(john)}, \neg\, \texttt{naughty(john)}\}, \emptyset), \\ &(\{\texttt{med\_present(C)}, \neg\, \texttt{skip(C)}\}, \{\texttt{not} \neg\, \texttt{nice(C)}, \texttt{child(C)}\}) \\ &(\emptyset, \{\texttt{not naughty(C)}, \texttt{skip(C)}\}) \\ &(\{\texttt{not} \neg\, \texttt{skip(C)}, \texttt{nice(C)}\}, 2, 100)\} \end{aligned}$$

♦

2.4 Properties of Abstract Programs

In this section we define several sets and functions over abstract programs for later use in Section 2.6.

A syntactical notice: Given a set S, $\mathbb{P}(S)$ denotes the power set (i.e., the set containing all subsets) of S; S^n $(n \geq 1)$ denotes the cartesian product of S with itself n times.

In Section 2.2 and Section 2.3 we have defined sets of various constructs. Now we define the sets of those constructs which actually occur in a given program.

Definition 2.4.1 (Literals of an Abstract Program)
The literals in a program are all literals which occur in the head or in the body of an abstract rule or in an abstract constraint.

$$\texttt{literals} : \mathit{Programs}_{\mathit{abstract}} \rightarrow \mathbb{P}(\mathfrak{Literals})$$

$$\texttt{literals}(\mathcal{P}) = \bigcup_{(H,B)\in\mathcal{P})} (H \cup B) \cup \bigcup_{(C,l,w)\in\mathcal{P}} C$$

■

Definition 2.4.2 (Atoms of an Abstract Program)
We begin by defining the explicitly negated literal in a general literal, denoted $\texttt{literals}_L^{\neg}$*:*

$$\texttt{literals}_L^{\neg} : \mathfrak{Literals} \to \mathfrak{Literals}^{\neg}$$
$$\texttt{literals}_L^{\neg}(L) = \begin{cases} M & \text{if } L = \texttt{not } M \\ L & \text{else} \end{cases}$$

The atom of an explicitly negated literal, denoted $\texttt{atoms}_L$*, is defined as follows:*

$$\texttt{atoms}_L : \mathfrak{Literals}^{\neg} \to \mathfrak{Atoms}$$
$$\texttt{atoms}_L(L) = \begin{cases} A & \text{if } L = \neg\, A \\ L & \text{else} \end{cases}$$

Since in every general literal there is at most one atom, the atoms of an abstract program can be defined by using `literals` *and the two helper functions defined above:*

$$\texttt{atoms} : \mathit{Programs}_{\mathit{abstract}} \to \mathbb{P}(\mathfrak{Atoms})$$
$$\texttt{atoms}(\mathcal{P}) = \{A \mid L \in \texttt{literals}(\mathcal{P}), A = \texttt{atoms}_L(\texttt{literals}_L^{\neg}(L))\}$$

We also define the set of explicitly negated literals of a program similarly:

$$\texttt{literals}^{\neg} : \mathit{Programs}_{\mathit{abstract}} \to \mathbb{P}(\mathfrak{Literals}^{\neg})$$
$$\texttt{literals}^{\neg}(\mathcal{P}) = \{C \mid L \in \texttt{literals}(\mathcal{P}), C = \texttt{literals}_L^{\neg}(L)\}$$

■

Definition 2.4.3 (Predicates of an Abstract Program)
Again we first define a simple function which maps an atom to the predicate which occurs in it:

$$\texttt{predicates}_A : \mathfrak{Atoms} \to \mathfrak{Predicates}$$
$$\texttt{predicates}_A(A) = \begin{cases} A & \text{if } A \in \mathfrak{Predicates} \\ P & \text{if } A = P'('t_1, \ldots, t_n')', P \in \mathfrak{Predicates}, \\ & \quad t_1, \ldots, t_n \in \mathfrak{Terms} \end{cases}$$

Now it is easy to generalise this to abstract programs:

$$\texttt{predicates} : \mathit{Programs}_{\mathit{abstract}} \to \mathbb{P}(\mathfrak{Predicates})$$
$$\texttt{predicates}(\mathcal{P}) = \{P \mid A \in \texttt{atoms}(\mathcal{P}), P = \texttt{predicates}_A(A)\}$$

■

Definition 2.4.4 (Constants of an Abstract Program)
Not all terms are constants, so in order to determine the constants of an atom we have to take the intersection between the set of terms in this atom and all constants.

$$\texttt{constants}_A : \mathfrak{Atoms} \to \mathbb{P}(\mathfrak{Constants})$$
$$\texttt{constants}_A(A) = \begin{cases} \emptyset & \text{if } A \in \mathfrak{Predicates} \\ \{t_1, \ldots, t_n\} \cap & \text{if } A = P'('t_1, \ldots, t_n')', P \in \mathfrak{Predicates}, \\ \quad \cap\, \mathfrak{Constants} & \quad t_1, \ldots, t_n \in \mathfrak{Terms} \end{cases}$$

We proceed analogously to Definition 2.4.3, but if there is no constant at all in the program, we take an arbitrary one (in our case it is `a`*).*

$$\texttt{constants} : \mathit{Programs}_{\mathit{abstract}} \to \mathbb{P}(\mathfrak{Constants})$$

$$\texttt{constants}(\mathcal{P}) = \begin{cases} \bigcup_{A \in \texttt{atoms}(\mathcal{P})} \texttt{constants}_A(A) & \text{if } \bigcup_{A \in \texttt{atoms}(\mathcal{P})} \texttt{constants}_A(A) \neq \emptyset \\ \{a\} & \text{else} \end{cases}$$

■

Since the logical equivalent of abstract rules are clauses (cf. e.g. [Llo87]), variables appearing in abstract rules have to be considered universally quantified. Thus they have no global meaning, but one which is limited to exactly one rule. This means that it does not really make sense to determine the set of all variables in a program since two occurrences of the same variable in different abstract rules or constraints do not have the same meaning. Thus we define the set of variables in an abstract rule or constraint.

Definition 2.4.5 (Variables in an Abstract Rule or Constraint)
The function which maps an atom to the set of all variables occurring in it is defined similar to the function which maps an atom to the set of all constants in it, see Definition 2.4.4

$$\texttt{variables}_A : \mathfrak{Atoms} \to \mathbb{P}(\mathfrak{Variables})$$

$$\texttt{variables}_A(A) = \begin{cases} \emptyset & \text{if } A \in \mathfrak{Predicates} \\ \{t_1, \ldots, t_n\} \cap \cap\ \mathfrak{Variables} & \text{if } A = P'('t_1, \ldots, t_n')', P \in \mathfrak{Predicates}, \\ & \quad t_1, \ldots, t_n \in \mathfrak{Terms} \end{cases}$$

$$\texttt{variables} : \mathit{Rules}_{\mathit{abstract}} \cup \mathit{Constraints}_{\mathit{abstract}} \to \mathbb{P}(\mathfrak{Variables})$$

$$\texttt{variables}(R) = \begin{cases} \bigcup\limits_{L \in H \cup B} \texttt{variables}_A(\texttt{atoms}_L(L)) & \text{if } R = (H, B) \\ \bigcup\limits_{L \in B} \texttt{variables}_A(\texttt{atoms}_L(L)) & \text{if } R = (B, l, w) \end{cases}$$

■

2.4.1 Grounding of Programs

Variables are just placeholders for constants. So the meaning of a program should not change if we substitute each occurrence of a variable in some abstract rule or constraint by all constants occurring in the program, one at a time, and consider the set of all abstract rules and constraints which are constructible in this way.

Before we proceed, we have to define what "to substitute a constant for a variable" formally means.

Definition 2.4.6 (Substitutions)
A substitution is a set of mappings from variables to terms, where for each variable at most one mapping may exist. A mapping from a variable to itself is

not allowed. Formally:

$$\mathfrak{Substitutions} = \{V_1 \mapsto t_1, \ldots, V_n \mapsto t_n \mid V_1, \ldots, V_n \in \mathfrak{Variables}, \\ t_1, \ldots, t_n \in \mathfrak{Terms}, n \geq 1 \\ \bigwedge 1 \leq i \leq n \qquad (2.1) \\ 1 \leq j \leq n \\ i \neq j$$

$\mathfrak{Substitutions} = \{V_1 \mapsto t_1, \ldots, V_n \mapsto t_n \mid$
$t_1, \ldots, t_n \in \mathfrak{Terms}, n \geq 1$

■

Definition 2.4.7 (Application of Substitutions)
If a substitution is to be applied to a syntactic structure, this is written in postfix notation. Let $\sigma \in \mathfrak{Substitutions}$, $t \in \mathfrak{Terms}$, $A \in \mathfrak{Atoms}$, $M \in \mathfrak{Literals}^{\neg}$, $L \in \mathfrak{Literals}$, $(H, B) \in \mathit{Rules}_{\mathit{abstract}}$, $(C, l, w) \in \mathit{Constraints}_{\mathit{abstract}}$*:*

$$t\sigma = \begin{cases} t_1 & \text{if } t \mapsto t_1 \in \sigma \\ t & \text{else} \end{cases}$$

$$A\sigma = \begin{cases} A & \text{if } A \in \mathfrak{Predicates} \\ P'('t_1\sigma, \ldots, t_n\sigma')' & \text{if } A = P'('t_1, \ldots, t_n')', \\ & \quad t_1, \ldots, t_n \in \mathfrak{Terms} \end{cases}$$

$$M\sigma = \begin{cases} M\sigma & \text{if } M \in \mathfrak{Atoms} \\ \neg\, A\sigma & \text{if } M = \neg\, A, A \in \mathfrak{Atoms} \end{cases}$$

$$L\sigma = \begin{cases} L\sigma & \text{if } L \in \mathfrak{Literals}^{\neg} \\ \texttt{not}\ M\sigma & \text{if } L = \texttt{not}\ M, M \in \mathfrak{Literals}^{\neg} \end{cases}$$

$$(H, B)\sigma = (\{A\sigma \mid A \in H\}, \{L\sigma \mid L \in B\})$$

$$(C, l, w)\sigma = (\{L\sigma \mid L \in C\}, l, w)$$

■

At this point we may formulate what we have intuitively described above: A function which maps general abstract programs to so-called *ground* abstract programs (i.e., no variables occur in the program).

Definition 2.4.8 (Grounding)
A ground program consists of all abstract rules and constraints of the original program, where the variables have been substituted by every combination of all constants occurring in the program. So for every rule (or constraint) R we determine the set of all substitutions Σ_R *which map the variables in it to some combination of constants in the program (recall that the scope of a variable is*

restricted to one rule or constraint), and apply every substitution in this set to R. The set of all resulting rules is the ground program.

$$\begin{aligned}&\texttt{grounding}: \quad \mathit{Programs}_{\mathit{abstract}} \rightarrow \mathit{Programs}_{\mathit{abstract}}\\&\texttt{grounding}(\mathcal{P})=\{R\sigma \mid R \in \mathcal{P}, \sigma \in \Sigma_R\}\end{aligned}$$

where

$$v_i \neq v_j,$$

$(c_1, \ldots, c_n) \in \texttt{constants}(\mathcal{P})^n$,
$n = |\texttt{variables}(R)|\}$
Σ_R=$\{\{v_1 \mapsto c_1, \ldots, v_n \mapsto c_n\} \mid v_1, \ldots, v_n \in \texttt{variables}(R), \bigwedge 1 \leq i \leq n1 \leq j \leq ni \neq jv_i \neq v_j, (c_1, \ldots, c_n) \in$

■

Observation 3
The grounding of a ground abstract program is idempotent.

$$\forall \mathcal{P} \in \mathit{Programs}_{\mathit{abstract}} : \texttt{grounding}(\mathcal{P}) = \texttt{grounding}(\texttt{grounding}(\mathcal{P}))$$

★

Example 2.4.1
Again we take a snippet of Santa's DATALOG$^{not,\vee,w}$ *program:*

$$\begin{aligned}&\mathcal{P}^2_{Santa} =\\&\{\texttt{naughty(john)} \vee \neg\ \texttt{naughty(john)}.,\\&\ \texttt{nice(sue)}.,\\&\ \texttt{naughty(jim)}.,\\&\ \texttt{med\_present(C)} \vee \neg\ \texttt{skip(C)} \leftarrow \texttt{not}\ \neg\ \texttt{nice(C)}, \texttt{child(C)}.,\\&\ \leftarrow \texttt{skip(C)}, \texttt{visit(C)}.,\\&\ \Leftarrow \texttt{not}\ \neg\ \texttt{skip(C)}, \texttt{nice(C)}.[2:100]\}\end{aligned}$$

$$\begin{aligned}&\mathcal{P}^{2,A}_{Santa} = \texttt{abstract}(\mathcal{P}^2_{Santa}) =\\&\{(\{\texttt{naughty(john)}, \neg\ \texttt{naughty(john)}\}, \emptyset),\\&\ (\{\texttt{nice(sue)}\}, \emptyset),\\&\ (\{\texttt{naughty(jim)}, \emptyset),\\&\ (\{\texttt{med\_present(C)}, \neg\ \texttt{skip(C)}\}, \{\texttt{not}\ \neg\ \texttt{nice(C)}, \texttt{child(C)}\}),\\&\ (\emptyset, \{\texttt{skip(C)}, \texttt{visit(C)}\}),\\&\ (\{\texttt{not}\ \neg\ \texttt{skip(C)}, \texttt{nice(C)}\}, 2, 100)\}\}\end{aligned}$$

For the first three abstract rules $\Sigma_R = \emptyset$, since no variables occur in them. The other abstract rules and the abstract constraint all contain one variable C, so the respective Σ_Rs for them are the variable C, substituted by all constants

occurring in the abstract program, $\texttt{constants}(\mathcal{P}^{2,A}_{Santa}) = \{\texttt{john}, \texttt{sue}, \texttt{jim}\}$*:* $\Sigma_R = \{\sigma_1, \sigma_2, \sigma_3\}$, $\sigma_1 = \{\texttt{C} \mapsto \texttt{john}\}, \sigma_2 = \{\texttt{C} \mapsto \texttt{sue}\}, \sigma_3 = \{\texttt{C} \mapsto \texttt{jim}\}$. *Note that it is just because of the simplicity of the example that all* Σ_R*s are equal.*
The result of applying the substitutions is:

$$
\begin{aligned}
&\texttt{grounding}(\mathcal{P}^{2,A}_{Santa})) = \\
&\{(\{\texttt{naughty(john)}, \neg\, \texttt{naughty(john)}\}, \emptyset), \\
&(\{\texttt{nice(sue)}\}, \emptyset), \\
&(\{\texttt{naughty(jim)}, \emptyset), \\
&(\{\texttt{med\_present(john)}, \neg\, \texttt{skip(john)}\}, \{\texttt{not} \neg\, \texttt{nice(john)}, \texttt{child(john)}\}), \\
&(\{\texttt{med\_present(sue)}, \neg\, \texttt{skip(sue)}\}, \{\texttt{not} \neg\, \texttt{nice(sue)}, \texttt{child(sue)}\}), \\
&(\{\texttt{med\_present(jim)}, \neg\, \texttt{skip(jim)}\}, \{\texttt{not} \neg\, \texttt{nice(jim)}, \texttt{child(jim)}\}), \\
&(\emptyset, \{\texttt{skip(john)}, \texttt{visit(john)}\}), \\
&(\emptyset, \{\texttt{skip(sue)}, \texttt{visit(sue)}\}), \\
&(\emptyset, \{\texttt{skip(jim)}, \texttt{visit(jim)}\}), \\
&(\{\texttt{not} \neg\, \texttt{skip(john)}, \texttt{nice(john)}\}, 2, 100)\}\} \\
&(\{\texttt{not} \neg\, \texttt{skip(sue)}, \texttt{nice(sue)}\}, 2, 100)\}\} \\
&(\{\texttt{not} \neg\, \texttt{skip(jim)}, \texttt{nice(jim)}\}, 2, 100)\}\}
\end{aligned}
$$

♦

2.5 Syntactically Equivalent Programs

Using abstract programs and the grounding mechanism, we are now able to express equivalences on a low syntactical level.

Definition 2.5.1 (Syntactically Equivalent Programs)
Two programs $\mathcal{P}_1, \mathcal{P}_2 \in (\Pi_{\text{DATALOG}^{not,\vee,s}} \cup \Pi_{\text{DATALOG}^{not,\vee,w}})$ *are syntactically equivalent, denoted as* $\mathcal{P}_1 \equiv_S \mathcal{P}_2$*, iff the corresponding grounded abstract programs are identical.*

$$\mathcal{P}_1 \equiv_S \mathcal{P}_2 \iff \texttt{grounding}(\texttt{abstract}(\mathcal{P}_1)) = \texttt{grounding}(\texttt{abstract}(\mathcal{P}_2))$$

■

Example 2.5.1
The following programs are equivalent:

$$
\begin{aligned}
\mathcal{P}^1_{eq} = \{&\texttt{a(X)} \vee \texttt{a(s)} \leftarrow \texttt{b(X)}., \\
&\Leftarrow \texttt{a(C)}, \neg\, \texttt{f(s)}.[1:4], \\
&\leftarrow \texttt{b(V)}, \texttt{b(V)}.\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}^2_{eq} = \{&\texttt{a(X)} \leftarrow \texttt{b(X)}., \\
&\Leftarrow \texttt{a(s)}, \neg\, \texttt{f(s)}.[1:4], \\
&\leftarrow \texttt{b(s)}.\}
\end{aligned}
$$

$$\mathcal{P}^3_{eq} = \{\texttt{a(s)} \leftarrow \texttt{b(X)}, \texttt{b(s)}.,$$
$$\Leftarrow \texttt{a(X)}, \neg\, \texttt{f(X)}.[\texttt{1} : \texttt{4}],$$
$$\leftarrow \texttt{b(V)}.\}$$

♦

2.6 Semantics of DATALOG$^{\neg,not,\vee,w}$

The background of this section is mathematical logic, but we do not define semantics for arbitrary first order languages (as it is done in [LMR92], e.g.). We simply adapt results of this broader field to our notion of abstract programs.

Since the extended languages and the languages containing weak constraints are comparatively novel in research, they will be dealt with later on.

Here are some abbreviations, which will be used throughout this and later chapters:

2.6.1 General Interpretations and Models

An interpretation associates some meaning to an abstract program. We describe this by transforming our syntactical constructs to a different domain. So we have to define a mapping from constants to objects in this different domain, and a mapping from predicate symbols to functions, which associate either **true** or **false** to any given n-tuple, where n must of course match the predicate symbol's arity. This describes so-called *total* interpretations (everything is either true or false). There is also the notion of *partial* interpretations, which extends total semantics by providing a third truth value (**undefined**).

One could loosely describe this formalism as a "context switch" or homomorphism between statements in the given language to statements in some different context.

One remark: We did not mention what to do with variables. Since variables represent constants, it is sufficient to consider grounded programs. Otherwise, it would have been necessary to rename the variables of each abstract rule or constraint (since the scope of meaning of one variable is a single abstract rule or constraint) and to define a variable assignment, mapping variables to some domain objects. Then, every variable assignment should have been considered with a given interpretation. When considering variable assignments instead of grounded programs, the results do not differ.

Definition 2.6.1 ((Total) Interpretations)
An interpretation $\mathcal{I}$ of a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, consists of:

- *a set D, called the* **domain**
- *a function $f_c : \texttt{constants}(\mathcal{P}) \rightarrow D$, mapping each constant of $\mathcal{P}$ to an element of D*
- *for each $p \in \texttt{predicates}(\mathcal{P})$ a function $f_p : D^{|\texttt{arity}(p)|} \rightarrow \{true, false\}$*
- *a function F, associating each $p \in \texttt{predicates}(\mathcal{P})$ with the appropriate f_p*

We denote $\mathcal{I}$ as the triple $\langle D, f_c, F\rangle$. ■

We now give the function which transforms atoms into the domain of an interpretation $\mathcal{I}$.

Definition 2.6.2 (Evaluation of Ground Atoms in an Interpretation)
Let $\mathcal{I} = \langle D, f_c, F\rangle$

$$\texttt{eval}_A^{\mathcal{I}} : \mathfrak{Atoms} \to \{true, false\}$$

$$\texttt{eval}_A^{\mathcal{I}} = \begin{cases} f_A & \text{if } A \in \mathfrak{Predicates}, F(A) = f_A \\ f_p(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) & \text{if } A = p'('t_1, \ldots, t_n')', \\ & \quad p \in \mathfrak{Predicates}, t_1, \ldots, t_n \in \mathfrak{Constants}, \\ & \quad F(p) = f_p, f_c(t_1) = t_1^{\mathcal{I}}, \ldots, f_c(t_n) = t_n^{\mathcal{I}} \end{cases}$$

■

Note that $\texttt{eval}_A^{\mathcal{I}}$ is only defined for ground atoms. This is no restriction since for every program a syntactically equivalent ground program exists: the one obtained by grounding it.

An interpretation describes a possible world. We are interested in those worlds which are consistent with our facts, rules and constraints. A fact, rule or constraint is said to be *satisfied by $\mathcal{I}$* if it is consistent with the hypothetical world described by $\mathcal{I}$. Note that we do not consider extended programs yet.

Definition 2.6.3 (Satisfaction of Abstract Rules and Constraints)
Given a (total) interpretation $\mathcal{I} = \langle D, f_c, F\rangle$, an abstract rule $(H, B) \in \mathit{Rules}_{abstract}$ is satisfied, if and only if at least one atom in the head evaluates to true in $\mathcal{I}$, or at least one literal in the body evaluates to false in $\mathcal{I}$.

$$\begin{aligned} \mathcal{I} \models (H, B) \iff & \exists A \in H : \texttt{eval}_A^{\mathcal{I}}(A) = true \\ & \vee \\ & \exists L \in B \cap \mathfrak{Atoms} : \texttt{eval}_A^{\mathcal{I}}(L) = false \\ & \vee \\ & \exists\, \texttt{not}\ A \in B - \mathfrak{Atoms} : \texttt{eval}_A^{\mathcal{I}}(A) = true \end{aligned}$$

Similarly, an abstract constraint $(C, l, w) \in \mathit{Constraints}_{abstract}$ is satisfied if and only if at least one literal in the constraint evaluates to false in $\mathcal{I}$.

$$\begin{aligned} \mathcal{I} \models (C, l, w) \iff & \exists L \in C \cap \mathfrak{Atoms} : \texttt{eval}_A^{\mathcal{I}}(L) = false \\ & \vee \\ & \exists\, \texttt{not}\ A \in C - \mathfrak{Atoms} : \texttt{eval}_A^{\mathcal{I}}(A) = true \end{aligned}$$

■

Definition 2.6.4 (Models)
An interpretation $\mathcal{I}$ is called a model of a grounded abstract program $\mathcal{P} =$ $\texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, if all abstract rules are satisfied. Note again that we defer the definition of the meaning of abstract constraints and thus weak constraints) to Section 2.6.7.

$$\texttt{Models}(\mathcal{P}) = \{\mathcal{I} \mid \forall r \in \texttt{grounding}(\mathcal{P}) \cap \mathit{Rules}_{abstract} : \mathcal{I} \models r\}$$

■

2.6.2 Herbrand Interpretations and Models

The model theory introduced in Section 2.6.1 is useful if one wants to prove that a given program satisfies a specification, but it is impracticable for computation or general analysis of the program, because one would have to consider infinitely many interpretations.

But a program in our language can not express more than is in its structure. Jacques Herbrand and Thoralf Skolem showed independently that there is an isomorphism between any model and one particular "structural" model. These "structural" interpretations are called Herbrand interpretations.

In order to construct this interpretation for a given ground abstract program, we provide some preliminary definitions first:

Definition 2.6.5 (Herbrand Universe)
The Herbrand Universe is the set of basic building blocks for the Herbrand Interpretation. Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{(}\text{DATALOG}^{not,\vee,w})$ *it is the set of constants in* $\mathcal{P}$*:*

$$HU(\mathcal{P}) = \texttt{constants}(\mathcal{P})$$

■

The Herbrand Universe is rather trivial here, but in general first order logic it is more complicated due to function symbols, which we do not have in our languages.

Definition 2.6.6 (Herbrand Base)
The Herbrand Base is what will be the domain set of the Herbrand Interpretation. It is simply the set of all atoms constructible out of the predicate and constant symbols in a given grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$.

$$\begin{aligned} HB(\mathcal{P}) = &\{p \mid p \in \texttt{predicates}(\mathcal{P}), \texttt{arity}(p) = 0\} \\ &\cup \\ &\{p'('t_1, \ldots, t_n')' \mid p \in \texttt{predicates}(\mathcal{P}), \\ &\qquad\qquad t_1, \ldots, t_n \in H_U(\mathcal{P}), n = \texttt{arity}(p)\} \end{aligned}$$

Sometimes we also need the notion of all negated atoms (negation-as-failure) in the Herbrand Base, denoted as $HB^{\texttt{not}}$*:*

$$HB^{\texttt{not}}(\mathcal{P}) = \{\texttt{not}\ a \mid a \in HB(\mathcal{P})\}$$

■

Observation 4
The Herbrand Base of a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *corresponds to the atoms in* $\mathcal{P}$*.*

$$\forall \mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee,w}} : HB(\mathcal{P}) = \texttt{atoms}(\mathcal{P})$$

★

Definition 2.6.7 ((Total) Herbrand Interpretation)
A Herbrand Interpretation of an abstract grounded program $\mathcal{P}$, where $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, is an interpretation $\mathcal{I}$ with fixed domain and constant-mapping function, $\mathcal{I} = \langle HB(\mathcal{P}), \texttt{id}, F\rangle$, where $\texttt{id}$ is the identity function, and F is arbitrary.

Representing $\mathcal{I}$ by F (the domain and constant-mapping function are fixed in all Herbrand Interpretations) is cumbersome, an easier method exists: Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, and a particular Herbrand Interpretation $\mathcal{I}$, we can identify two sets of atoms:

$$F_+ = \{A \mid eval_A^{\mathcal{I}}(A) = true, A \in HB(\mathcal{P})\}$$
$$F_- = \{\, \texttt{not}\ A \mid eval_A^{\mathcal{I}}(A) = false, A \in HB(\mathcal{P})\}$$
$$F_+ \cup \texttt{atoms}(F_-) = HB(\mathcal{P}), F_+ \cap \texttt{atoms}(F_-) = \emptyset$$

We can represent $\mathcal{I}$ simply by using $F_+ \cup F_-$, because F is completely determined in this way. We will therefore refer to $F_+ \cup F_-$ if we write a Herbrand Interpretation $\mathcal{I}$. ■

Definition 2.6.8 (Satisfaction in Herbrand Interpretations)
Using the characterisation in Definition 2.6.7, one can simplify Definition 2.6.3: Given a Herbrand Interpretation $\mathcal{I}$ of a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, an abstract rule $(H, B) \in \mathcal{P}$ is satisfied, if and only if at least one atom in the head is true in $\mathcal{I}$, or at least one literal in the body is false in $\mathcal{I}$.

$$\mathcal{I} \models (H, B) \iff (H \cap \mathcal{I} \neq \emptyset) \vee (B \not\subseteq \mathcal{I})$$

Similarly, an abstract constraint $(C, l, w) \in \mathcal{P}$ is satisfied, if and only if at least one literal of the constraint is false in $\mathcal{I}$.

$$\mathcal{I} \models (C, l, w) \iff C \not\subseteq \mathcal{I}$$

If an abstract rule or constraint is not satisfied w.r.t. a Herbrand Interpretation $\mathcal{I}$, it is called violated (w.r.t. $\mathcal{I}$). ■

Definition 2.6.9 (Herbrand Model)
A (total) Herbrand Model of a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ is a Herbrand Interpretation which is also a model as defined in Definition 2.6.4, possibly using the simplified satisfaction criterion in Definition 2.6.8.

$$\texttt{Models}^{\text{H}}(\mathcal{P}) = \{\mathcal{I}^+ \mid \mathcal{I} \subseteq (HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P})), \mathcal{I}^+ \cap \texttt{atoms}(\mathcal{I}^-) = \emptyset,$$
$$\mathcal{I}^+ \cup \texttt{atoms}(\mathcal{I}^-) = HB(\mathcal{P}), \forall r \in \mathcal{P} \cap \textit{Rules}_{\textit{abstract}} : \mathcal{I} \models r\}$$

■

For better readability, (total) Herbrand Models are referred to by just their positive atoms, in contrast to Herbrand Interpretations.

This definition of models and interpretations relies on the fact that all atoms are interpreted as either true or false but nothing else. In particular, we can not leave an atom undefined.

But sometimes, it makes indeed more sense to leave some atoms undefined: In cases where we do not know anything about the truth of an atom. See for instance [AHV95] for a more sophisticated motivation.

Interpretations which allow for the "truth value" *undefined* are referred to as "partial interpretations". In the literature, they are also called "3-valued interpretations", but according to [Prz91] this term is not the best choice, since the implication operator $\leftarrow$ has a different semantics than its counterpart in 3-valued logic.

Definition 2.6.10 (Partial (Herbrand) Interpretation)
Partial Interpretations and Partial Herbrand Interpretations are defined analogously to Definition 2.6.1 and Definition 2.6.7, in which Total Interpretations and Total Herbrand Interpretations have been defined. The difference is that the functions for a predicates in this case have the range $\{true, undefined, false\}$ *rather than just* $\{true, false\}$.

For this reason, also the $\texttt{eval}^{\mathcal{I}}_{A_P}$ *function for a Partial Interpretation is defined over the range* $\{true, undefined, false\}$ *instead of* $\{true, false\}$.

So Partial Herbrand Interpretations are Partial Interpretations of an abstract grounded program $\mathcal{P}$, *where* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$. *They are of the form* $\langle HB(\mathcal{P}), \texttt{id}, F\rangle$, *where* `id` *is the identity function and* F *is arbitrary.*

When we want to represent a Partial Herbrand Interpretation $\mathcal{I}$, *we can identify three sets of atoms (as opposed to two in the total case):*

$$F_+ = \{A \mid eval^{\mathcal{I}}_{A_P}(A) = true, A \in HB(\mathcal{P})\}$$
$$F_- = \{\texttt{not } A \mid eval^{\mathcal{I}}_{A_P}(A) = false, A \in HB(\mathcal{P})\}$$
$$F_u = \{A \mid eval^{\mathcal{I}}_{A_P}(A) = undefined, A \in HB(\mathcal{P})\}$$

$$F_+ \cup \texttt{atoms}(F_-) \cup F_u = HB(\mathcal{P})$$
$$F_+ \cap \texttt{atoms}(F_-) = \emptyset, F_+ \cap F_u = \emptyset, \texttt{atoms}(F_-) \cap F_u = \emptyset$$

Since the domain and the mapping of constants are fixed in a (Partial) Herbrand Interpretation $\mathcal{I}$, *we will refer to* $\mathcal{I}$ *by* $F_+ \cup F_-$, *because* $F_u = HB(\mathcal{P}) - (F_+ \cup \texttt{atoms}(F_-))$, *and thus* F *is fully determined by this representation.* ■

We will only describe satisfaction in Partial Herbrand Interpretations. For general Partial Interpretations the definition is similar.

Definition 2.6.11 (Satisfaction in Partial Herbrand Interpretations)
This definition is derived from [Prz90, Prz91]. Given a Partial Herbrand Interpretation $\mathcal{I}$ *of a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *an abstract rule* $(H, B) \in \mathcal{P}$ *is satisfied, if and only if at least one atom in the head is true in* $\mathcal{I}$, *or at least one literal in the body is false in* $\mathcal{I}$, *or there is at least one undefined atom in the head and at least one undefined literal in the body.*

$$\begin{aligned}\mathcal{I} \models (H, B) \Longleftrightarrow & (H \cap \mathcal{I} \neq \emptyset) \\ & \vee \\ & (B \cap \texttt{not}\ (\mathcal{I}) \neq \emptyset) \\ & \vee \\ & ((H - \texttt{atoms}(\mathcal{I}) \neq \emptyset) \wedge (\texttt{atoms}(B) - \texttt{atoms}(I) \neq \emptyset)\end{aligned}$$

Similarly, given a Herbrand Interpretation $\mathcal{I} \subseteq (HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}))$ *of a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}')), \mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *an abstract constraint* $(C, l, w) \in \mathcal{P}$ *is satisfied, if and only if at least one literal of the constraint is false in* $\mathcal{I}$.

$$\mathcal{I} \models (C, l, w) \Longleftrightarrow C \nsubseteq \mathcal{I}$$

■

Note that constraints cannot be satisfied by undefined literals only.

Definition 2.6.12 (Partial Herbrand Model)
A Partial Herbrand Model of a grounded abstract program $\mathcal{P}$, *where* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \text{DATALOG}^{not,\vee,w}$, *is a Partial Herbrand Interpretation of this program which satisfies all abstract rules of the grounded program.*

$$\begin{aligned}\texttt{Models}^{\texttt{H}}_{\texttt{P}}(\mathcal{P}) = \{\mathcal{I} \mid \mathcal{I} \subset (HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P})), \mathcal{I}^{+} \cap \texttt{atoms}(\mathcal{I}^{-}) = \emptyset \\ \forall r \in \mathcal{P} \cap \textit{Rules}_{abstract} : \mathcal{I} \models r\}\end{aligned}$$

■

Note that a Partial Herbrand Model cannot be represented by positive atoms only, since the information about the undefined atoms would be lost.

Observation 5 (Relation between Partial and Total Models)
Total Herbrand Models can be viewed as a special case of Partial Herbrand Models: A Partial Herbrand Model $\mathcal{I}$ *of a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \text{DATALOG}^{not,\vee,w}$, *is called Total Herbrand Model, iff* $\texttt{atoms}(\mathcal{I}) = HB(\mathcal{P})$. ★

A semantics of a program are Herbrand Models which possibly satisfy additional criteria. So a semantics is a subset of the set of Herbrand Models, and as in the case of Models total and partial semantics exist.

Definition 2.6.13 (Total Semantics)
A total semantics $SEM_T(\mathcal{P})$ *of some grounded abstract program* $\mathcal{P}$, *where* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \text{DATALOG}^{not,\vee,w}$, *is a subset of all Total Herbrand Models* $\texttt{Models}^{\texttt{H}}(\mathcal{P})$ *of this program.*

$$SEM_T(\mathcal{P}) \subseteq \texttt{Models}^{\texttt{H}}(\mathcal{P})$$

■

Definition 2.6.14 (Partial Semantics)
A partial semantics $SEM_P(\mathcal{P})$ of some grounded abstract program $\mathcal{P}$, where $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, is a subset of all Partial Herbrand Models $\texttt{Models}^{\texttt{H}}_{\texttt{P}}(\mathcal{P})$ of this program.

$$SEM_P(\mathcal{P}) \subseteq \texttt{Models}^{\texttt{H}}_{\texttt{P}}(\mathcal{P})$$

■

In this thesis, we will not consider any partial semantics in detail, but we will cite some references to partial semantics in Section 2.6.5, and our parametrised semantics for weak constraints, which will be defined in Section 2.6.7, also applies to partial semantics.

In the sequel, we will often refer to Herbrand Models simply as models, and to Herbrand Interpretations as interpretations, for the sake of simplicity.

2.6.3 Minimal Model Semantics

In general, a program can have several models:

Example 2.6.1
The following simple example describes the knowledge one might have about some person and the weather:

$$\begin{aligned}\mathcal{P}_{MM_1} =& \{happy., rains., happy \leftarrow sunny.\}\\ \mathcal{P}^{g,A}_{MM_1} =& \texttt{grounding}(\texttt{abstract}(\mathcal{P}_{MM_1})) =\\ & \{(\{happy\}, \emptyset), (\{rains\}, \emptyset), (\{happy\}, \{sunny\})\end{aligned}$$

We know that the person is happy and that it rains. We also know that if it is sunny, the person is always happy.

$\mathcal{P}_{MM_1}$ has the following models: $M_1 = \{happy, rains\}$, $M_2 = \{happy, rains, sunny\}$, i.e., $\texttt{Models}^{\texttt{H}}(\mathcal{P}^{g,A}_{MM_1}) = \{M_1, M_2\}$.

While M_1 is feasible and follows directly from our definition, M_2 is one possible situation, but the information sunny does not follow directly from our rules, but assuming its truth does not cause any inconsistency. ♦

As suggested by the example above, if $M_1, M_2 \in \texttt{Models}^{\texttt{H}}(\mathcal{P})$ of a ground abstract program $\mathcal{P}$, and $M_1 \subsetneq M_2$, then M_2 contains some information which can be safely assumed additionally. It is a possible scenario in a world which is described generically by M_1. Therefore, a model which is minimal w.r.t. $\subset$ describes a world in which only those atoms are true which are necessarily true in some family of similar worlds.

Definition 2.6.15 (Minimal Models)
Let $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, be a grounded abstract program, the set of minimal models of $\mathcal{P}$ (and also $\mathcal{P}$') is denoted as $MM(\mathcal{P})$ and defined as follows:

$$MM(\mathcal{P}) = \{M | M \in \texttt{Models}^{\texttt{H}}(\mathcal{P}), \nexists M_* \in \texttt{Models}^{\texttt{H}}(\mathcal{P}) : M_* \subset M\}$$

■

In the case of definite programs, one can show that a model exists for every program, and that exactly one minimal model exists w.r.t. set inclusion. For a proof, cf. [Llo87].

But in the general case of $\mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, there might be several Minimal Models, as the following example illustrates:

Example 2.6.2

$$\mathcal{P}_{MM_{II}} = \{a \vee b \leftarrow, c \leftarrow a\}$$
$$\mathcal{P}^{g,A}_{MM_{II}} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}_{MM_{II}})) = \{(\{a,b\},\emptyset),(\{c\},\{a\})\}$$

$$\texttt{Models}^{\text{H}}(\mathcal{P}^{g,A}_{MM_{II}}) = \{M_1, M_2, M_3, M_4\}$$
$$M_1 = \{b\} \quad M_2 = \{a,c\} \quad M_3 = \{b,c\} \quad M_4 = \{a,b,c\}$$

Since $M_1 \subset M_3 \subset M_4$ *and* $M_2 \subset M_4$, *but* $M_1 \not\subseteq M_2$ *and* $M_2 \not\subseteq M_1$, $MM(\mathcal{P}^{g,A}_{MM_{II}}) = \{M_1, M_2\}$ *holds.* ♦

2.6.4 Stable Model Semantics

The Stable Model Semantics was originally defined by Michael Gelfond and Vladimir Lifschitz for normal programs in [GL88], and has later been extended to disjunctive programs by Teodor Przymusinski in [Prz91] and by Gelfond and Lifschitz in [GL91].

Usually, this semantics is defined by the so-called Gelfond-Lifschitz transform. We will reproduce this transform here, but note that its semantics can also be defined in other terms, as shown by S. Brass and J. Dix in [BD97], which allows for a better comparison to other semantics.

It should also be mentioned that independently N. Bidoit and C. Froidevaux described the "Default Model Semantics", which is based on default logic, while the stable model semantics is based on autoepistemic logic. They showed in [BF91a, BF91b] that this semantics is equivalent to the Stable Model Semantics.

The Gelfond-Lifschitz transform eliminates negation-as-failure literals in the bodies of rules: For any interpretation, those rules are discarded, in which negation-as-failure literals occur that are false w.r.t. the interpretation. All remaining negation-as-failure literals are subsequently erased, yielding a negation-free program.

Definition 2.6.16 (Gelfond-Lifschitz transform [Prz91, GL88])
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *and an interpretation* $\mathcal{I}$ *of* $\mathcal{P}$*:*

$$\mathcal{P}^{\mathcal{I}} = \{(H, B') \mid (H,B) \in \mathcal{P}, \forall\, \texttt{not}\ A \in B : \texttt{not}\ A \in \mathcal{I}, B' = B \cap \mathfrak{Atoms}\}$$

■

Example 2.6.3
Consider the following program $\mathcal{P}_{stable}$*:*

$$\mathcal{P}_{stable} = \{a \vee b \leftarrow \texttt{not } c., \tag{2.2}$$
$$c \leftarrow a, \texttt{not } b.\} \tag{2.3}$$

The corresponding grounded abstract program $\mathcal{P}^{g,A}_{stable}$ *is:*

$$\mathcal{P}^{g,A}_{stable} = \{(\{a,b\}, \{\texttt{not } c\}), \tag{2.4}$$
$$(\{c\}, \{a, \texttt{not } b\})\} \tag{2.5}$$

since $\texttt{abstract}(\mathcal{P}_{stable})$ *is propositional and therefore already ground.* $HB(\mathcal{P}^{g,A}_{stable}) = \{a, b, c\}$ *and there are eight interpretations:*

$$\mathcal{I}_0 = \{\texttt{not } a, \texttt{not } b, \texttt{not } c\}, \quad \mathcal{I}_1 = \{a, \texttt{not } b, \texttt{not } c\}, \quad \mathcal{I}_2 = \{\texttt{not } a, b, \texttt{not } c\},$$
$$\mathcal{I}_3 = \{\texttt{not } a, \texttt{not } b, c\}, \quad \mathcal{I}_4 = \{a, b, \texttt{not } c\}, \quad \mathcal{I}_5 = \{a, \texttt{not } b, c\},$$
$$\mathcal{I}_6 = \{\texttt{not } a, b, c\}, \quad \mathcal{I}_7 = \{a, b, c\}$$

The corresponding Gelfond-Lifschitz transformed programs are:

$$\mathcal{P}^{g,A^{\mathcal{I}_0}}_{stable} = \{(\{a,b\}, \emptyset), (\{c\}, \{a\})\} \quad \mathcal{P}^{g,A^{\mathcal{I}_1}}_{stable} = \{(\{a,b\}, \emptyset), (\{c\}, \{a\})\}$$
$$\mathcal{P}^{g,A^{\mathcal{I}_2}}_{stable} = \{(\{a,b\}, \emptyset)\} \quad \mathcal{P}^{g,A^{\mathcal{I}_3}}_{stable} = \{(\{c\}, \{a\})\}$$
$$\mathcal{P}^{g,A^{\mathcal{I}_4}}_{stable} = \{(\{a,b\}, \emptyset)\} \quad \mathcal{P}^{g,A^{\mathcal{I}_5}}_{stable} = \{(\{c\}, \{a\})\}$$
$$\mathcal{P}^{g,A^{\mathcal{I}_6}}_{stable} = \emptyset \quad \mathcal{P}^{g,A^{\mathcal{I}_7}}_{stable} = \emptyset$$

Consider e.g. $\mathcal{I}_2 = \{b\}$*: By application of the Gelfond-Lifschitz transform we get* $(\{a,b\}, \emptyset)$ *from* $(\{a,b\}, \{\texttt{not } c\})$*, since for the negation-as-failure literal* $\texttt{not } c$ *in the body* $c \notin \mathcal{I}_2$ *holds, and it is therefore deleted from the body.* $(\{c\}, \{a, \texttt{not } b\})$ *is deleted as a whole since* $b \in \mathcal{I}_2$*.* ♦

Stable models are those interpretations whose positive part is one of the minimal models of the corresponding Gelfond-Lifschitz-transformed program.

Definition 2.6.17 (Stable Models [Prz91, GL88])
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*,* $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$*,* $SM(\mathcal{P})$ *denotes the set of all stable models of* $\mathcal{P}$ *(and* $\mathcal{P}$*'), defined as:*

$$SM(\mathcal{P}) = \{\mathcal{I}^+ \mid \mathcal{I} \in (HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P})), \mathcal{I}^+ \cap \texttt{atoms}(\mathcal{I}^-) = \emptyset,$$
$$\mathcal{I}^+ \cup \texttt{atoms}(\mathcal{I}^-) = HB(\mathcal{P}), \mathcal{I}^+ \in MM(\mathcal{P}^{\mathcal{I}})\}$$

■

We now examine which of the interpretations of Example 2.6.3 are Stable Models:

Example 2.6.4 (Example 2.6.3 ctd.)

$$
\begin{aligned}
MM(\mathcal{P}^{g,A^{\mathcal{I}_0}}_{stable}) &= \{\{b\},\{a,c\}\}, \quad MM(\mathcal{P}^{g,A^{\mathcal{I}_1}}_{stable}) = \{\{b\},\{a,c\}\}, \textit{ cf. Example 2.6.2}\\
MM(\mathcal{P}^{g,A^{\mathcal{I}_2}}_{stable}) &= \{\{a\},\{b\}\}, \quad MM(\mathcal{P}^{g,A^{\mathcal{I}_3}}_{stable}) = \{\emptyset\},\\
MM(\mathcal{P}^{g,A^{\mathcal{I}_4}}_{stable}) &= \{\{a\},\{b\}\}, \quad MM(\mathcal{P}^{g,A^{\mathcal{I}_5}}_{stable}) = \{\emptyset\},\\
MM(\mathcal{P}^{g,A^{\mathcal{I}_6}}_{stable}) &= \{\emptyset\}, \quad MM(\mathcal{P}^{g,A^{\mathcal{I}_7}}_{stable}) = \{\emptyset\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I}_0^+ = \emptyset \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_0}}_{stable}) \qquad & \mathcal{I}_1^+ = \{a\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_1}}_{stable})\\
\mathcal{I}_2^+ = \{b\} \in MM(\mathcal{P}^{g,A^{\mathcal{I}_2}}_{stable}) \qquad & \mathcal{I}_3^+ = \{c\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_3}}_{stable})\\
\mathcal{I}_4^+ = \{a,b\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_4}}_{stable}) \qquad & \mathcal{I}_5^+ = \{a,c\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_5}}_{stable})\\
\mathcal{I}_6^+ = \{b,c\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_6}}_{stable}) \qquad & \mathcal{I}_7^+ = \{a,b,c\} \notin MM(\mathcal{P}^{g,A^{\mathcal{I}_7}}_{stable})
\end{aligned}
$$

Therefore $SM(\mathcal{P}^{g,A}_{stable}) = \{\{b\}\}$, *since* $\mathcal{I}_2$ *is the only interpretation which satisfies the criterion in Definition 2.6.17.*

♦

We conclude this section with a larger toy problem:

Example 2.6.5
This example is a formulation of riddle number 167 in Raymond Smullyan's book [Smu78].

The problem is the following: Hypothetical Transsylvania is populated by Transsylvanians, who are either humans or vampires (rule 2.7), and who are either sane or insane beings (rule 2.8). Humans always tell the truth, whereas vampires are malicious and therefore cannot help lying all the time. Sane creatures have a correct picture of the world, whereas insane ones perceive the world inversely, i.e., things that are true in the real world are perceived as wrong by them and vice versa.

Therefore there are four types of people in Transsylvania: Sane humans, who tell true things about the world (rule 2.9); insane humans, who perceive everything in the wrong way, and therefore do not tell the truth about the world; sane vampires, whose beliefs about the world are correct, but must lie about it and hence do not tell the truth; finally, insane vampires lie about wrong percepts and therefore tell true things about the world (rule 2.10).

Now a particular Transsylvanian, call him $fred$ *(fact 2.5), says "I am human or I am sane." (fact 2.6). What type of inhabitant is he?*

If some Transsylvanian, who makes such a statement, tells the truth, (s)he is either human or sane (rule 2.11). Conversely, if a lying Transsylvanian makes that statement, (s)he must be a vampire and insane (rules 2.12 and 2.13).

Let $\mathcal{P}_{vampire}$ be the following program:

$$tr(fred). \tag{2.6}$$
$$statement(fred). \tag{2.7}$$
$$h(T) \vee v(T) \leftarrow tr(T). \tag{2.8}$$
$$i(T) \vee s(T) \leftarrow tr(T). \tag{2.9}$$
$$tt(T) \leftarrow h(T), s(T). \tag{2.10}$$
$$tt(T) \leftarrow v(T), i(T). \tag{2.11}$$
$$h(T) \vee s(T) \leftarrow tt(T), statement(T). \tag{2.12}$$
$$v(T) \leftarrow \texttt{not}\ tt(T), statement(T). \tag{2.13}$$
$$i(T) \leftarrow \texttt{not}\ tt(T), statement(T). \tag{2.14}$$

The grounded abstract program is shown next (the grounding step is easy in this case, since there is only one constant, $fred$).

$$\mathcal{P}^{g,A}_{vampire} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}_{vampire}))$$
$$\mathcal{P}^{g,A}_{vampire} = \{(\{tr(fred)\}, \emptyset), \tag{2.5'}$$
$$(\{statement(fred)\}, \emptyset), \tag{2.6'}$$
$$(\{h(fred), v(fred)\}, \{tr(fred)\}), \tag{2.7'}$$
$$(\{i(fred), s(fred)\}, \{tr(fred)\}), \tag{2.8'}$$
$$(\{tt(fred)\}, \{h(fred), s(fred)\}), \tag{2.9'}$$
$$(\{tt(fred)\}, \{v(fred), i(fred)\}), \tag{2.10'}$$
$$(\{h(fred), s(fred)\}, \{tt(fred), statement(fred)\}), \tag{2.11'}$$
$$(\{v(fred)\}, \{\texttt{not}\ tt(fred), statement(fred)\}), \tag{2.12'}$$
$$(\{i(fred)\}, \{\texttt{not}\ tt(fred), statement(fred)\})\} \tag{2.13'}$$

Now let us consider possible interpretations, and check whether they are stable models. To answer the question of what type of inhabitant $fred$ is, it is sufficient to examine whether $\{h(fred), s(fred)\}$, $\{h(fred), i(fred)\}$, $\{v(fred), s(fred)\}$, or $\{v(fred), i(fred)\}$ is in a stable model for the answer to be "He is a sane human.", "He is an insane human.", "He is a sane vampire.", or "He is an insane vampire.", respectively.

First, let us consider whether a model is possible which contains $\{h(fred), s(fred)\}$. Obviously, the model must contain $tr(fred)$ and $statement(fred)$ in order to satisfy (2.5') and (2.6'). By this inclusion, (2.7') and (2.8') are satisfied as well. By (2.9'), the model must must also include $tt(fred)$, thus also satisfying (2.11'), while (2.10') is satisfied as well at this point, since the body is not true.

So now let us check whether $\{h(fred), s(fred), tr(fred), statement(fred), tt(fred)\}$ is a stable model: The Gelfond-Lifschitz-transformed program is essentially the same program without the last two rules (2.12', 2.13'). $\{h(fred), s(fred), tr(fred), statement(fred), tt(fred)\}$ is indeed a minimal model of this program. Thus one possible answer to our question is that $fred$ is a sane human.

What about the other cases? Informally, for possible models containing $\{h(fred), i(fred)\}$ or $\{v(fred), s(fred)\}$, $tt(fred)$ cannot be derived directly.

But if $tt(fred)$ is not included in the candidate model, the last two rules are not discarded during the Gelfond-Lifschitz-transform as in the previous case. Rather `not` *$tt(fred)$ is eliminated from them, which means that a model for the Gelfond-Lifschitz-transformed program contains also $v(fred)$ and $i(fred)$ and therefore it also must contain $tt(fred)$ by (2.10'). So the minimal model of the Gelfond-Lifschitz-transformed program is not equal to the original candidate model and thus no stable model. On the other hand, if $tt(fred)$ was contained in the candidate model, the last two rules would be thrown away, but then $tt(fred)$ is not in a minimal model of the transformed program, since the premises of neither (2.9') nor (2.10') hold.*

The last case is a model which contains $\{v(fred), i(fred)\}$. By (2.10'), this model must contain $tt(fred)$. But then the Gelfond-Lifschitz-transform discards the last two rules, and a minimal model must contain $h(fred)$ and $s(fred)$ in order to satisfy (2.11'). As we have seen above, the model containing $h(fred)$ and $s(fred)$ but not $v(fred)$ and $i(fred)$ is already minimal, so the minimal model has to be different from the candidate model, which entails that no model containing $v(fred)$ and $i(fred)$ can be stable.

In total, we can conclude that $fred$ must be a sane human. ♦

2.6.5 Other Semantics

Most semantics for DATALOG$^{not,\vee}$ (or disjunctive logic programming in general) different from the Minimal Model Semantics and the Stable Model Semantics are partial semantics. The most important one of these is the Well-founded Semantics, generalised to the disjunctive case [Ros90]. Also the more recently introduced static semantics [Prz95] has been talked about much. [Dix95] gives a comprehensive survey of various semantics for several kinds of logic programs including function symbols, thus also subsuming our function-free DATALOG languages.

2.6.6 Answer Sets – Semantics for Extended Programs

So far we have only considered programs without explicitly negated literals. In [GL91], Michael Gelfond and Vladimir Lifschitz have defined the notion of *answer sets*, which is the rough counterpart of a model in the scope of programs including explicit literals.

The general difference between a stable model and an answer set (as defined in [GL91]) is that an answer set consists of explicitly negated ground atoms rather than just of atoms and that an answer set in which both A and $\neg A$, $A \in Atoms$ occur is defined to be the set of all literals ($\mathfrak{Literals}$). Otherwise answer sets are defined identically to stable models. Consequently, this approach has later been generalised to other semantics than the Stable Model Semantics.

Answer sets containing some pair A and $\neg A$, $A \in Atoms$ are called *inconsistent answer sets*, all others are referred to as *consistent answer sets*. The definition of answer sets implies that inconsistent answer sets are always the set of all literals.

Instead of defining answer sets as a separate concept, we give a translation from extended programs to non-extended programs, such that there is a one-to-one correspondence between the consistent answer sets of the extended program and the stable models of the translated program.

This transformation is built on the fact that we can use a new predicate p^* instead of $\neg p$. Of course one has to make sure that no ground atoms $p(c_1, \ldots, c_n)$ and $p^*(c_1, \ldots, c_n)$ occur tpgether in one model representing an answer set. But first, let us define the substitution of p^* for $\neg p$ formally:

Definition 2.6.18 (Predicates for Explicitly Negated Predicates)
For each explicitly negated literal $\neg\, p(t_1, \ldots, t_n)$ in a program $\mathcal{P}$ we define an atom with a unique predicate p^*, which does not occur in $\mathcal{P}$ and is also unique for each predicate in $\mathcal{P}$. (Such a new predicate p^* exists since the number of predicates in a program is finite, whereas the total number of predicates is infinite.)

$$\mathtt{pos}^{\mathcal{P}}_{eL} : \mathfrak{Literals}^{\neg} \to \mathfrak{Atoms}$$
$$\mathtt{pos}^{\mathcal{P}}_{eL}(L) = \begin{cases} L & \text{if } L \in \mathfrak{Atoms} \\ p^*(t_1, \ldots, t_n) & \text{if } L = \neg\, p(t_1, \ldots, t_n) \end{cases}$$

We extend this to general literals:

$$\mathtt{pos}^{\mathcal{P}}_{L} : \mathfrak{Literals} \to \mathfrak{Literals}^{\mathtt{not}}$$
$$\mathtt{pos}^{\mathcal{P}}_{L}(L) = \begin{cases} \mathtt{pos}_{eL}(L) & \text{if } L \in \mathfrak{Literals}^{\neg} \\ \mathtt{not}\ \mathtt{pos}_{eL}(M) & \text{if } L = \mathtt{not}\ M \end{cases}$$

Applying these operations to abstract rules and constraints yields $\mathtt{pos}^{\mathcal{P}}$:

$$\mathtt{pos}^{\mathcal{P}}_{a} : \mathit{Rules}_{\mathit{abstract}} \cup \mathit{Constraints}_{\mathit{abstract}} \to \mathit{Rules}_{\mathit{abstract}} \cup \mathit{Constraints}_{\mathit{abstract}}$$
$$\mathtt{pos}^{\mathcal{P}}_{a}((H, B)) = (\{\mathtt{pos}_{eL}(L) \mid L \in H\}, \{\mathtt{pos}_{L}(L) \mid L \in B)$$
$$\mathtt{pos}^{\mathcal{P}}_{a}((C, l, w)) = (\{\mathtt{pos}_{L}(L) \mid L \in C\}, l, w)$$

Finally, for each $\mathcal{P} \in \text{DATALOG}^{\neg, not, \vee, w}$, we define a $\mathcal{P}^*$

$$\mathcal{P}^* = \{\mathtt{pos}^{\mathcal{P}}_{a}(r) \mid r \in \mathtt{abstract}(\mathcal{P})\}$$

■

$\mathcal{P}^*$ captures the meaning of consistent answer sets. But what should we do about inconsistent answer sets? Clinging to the original definition does not make sense in our framework, since the set of all literals is infinite and it contains symbols which do not occur in the program. Also brave reasoning (considering a literal to be entailed by the program if it occurs in at least one model of the program) would not make any sense. Also we think that an answer set is inconsistent is no proper meaning of a program, so we do not consider inconsistent answer sets.

In our approach, a program, which has an inconsistent answer set is semantically equivalent to a program which does not have any answer sets at all. To this end we define constraints which disallow any atom and its corresponding explicitly negated literal to occur in the same model of $\mathcal{P}^*$.

Definition 2.6.19 (Answer Set Constraints)

$$\mathcal{P}^{Consistency} = \{\ (\emptyset, \{p(X\_1, \ldots, X\_n), p^*(X\_1, \ldots, X\_n)\}) \mid \\ p^* \in \texttt{predicates}(\mathcal{P}^*), \texttt{arity}(p) = n\}$$

■

So finally, we define the transformed program $\mathcal{P}^+ \in \mathit{Programs}_{\mathit{abstract}}$ for a program $\mathcal{P} \in \text{DATALOG}^{\neg,not,\vee,w}$ whose stable models correspond to the consistent answer sets of $\mathcal{P}$.

Definition 2.6.20 (Answer Set Transformation)

$$\mathcal{P}^+ = \mathcal{P}^* \cup \mathcal{P}^{Consistency}$$

■

2.6.7 Parametrised Semantics for Programs with Weak Constraints

We now define the semantics for weak constraints. We do not require any particular semantics for the part of the program without weak constraints. Given a total or partial semantics $SEM(\mathcal{P})$ (cf. Definition 2.6.13, Definition 2.6.14) of some program, we call the models $M \in SEM(\mathcal{P})$ *candidate models* of $\mathcal{P}$. These models satisfy all the mandatory facts, rules, and strong constraints, according to some semantics. The task is then to choose those models which satisfy as many weak constraints as possible, taking into account the layer and weight informations associated with each of the weak constraints.

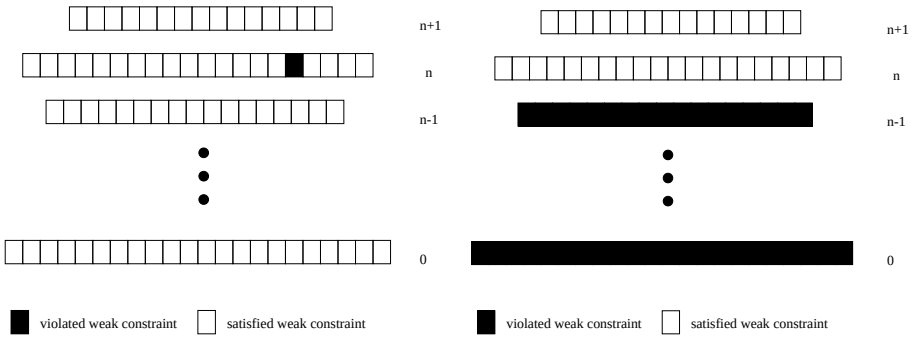
Maximising the number of satisfied constraints (w.r.t. the layer and weight information) among the candidate models is equivalent to minimising the number of violated constraints among the candidate models (again taking into account the layer and weight information). This is the approach we will follow in order to tackle this problem.

The meaning of layers should be as follows: Weak Constraints in a higher layer are more important than all Weak Constraint of lower layers together.

By "more important" we mean that if we have the choice between a model which violates some constraints of lower layers and one which violates some constraints of a higher layer, the former alternative should be chosen. In particular, a candidate model M_1 should be better than another candidate model M_2 if in M_1 all constraint of layers n and above are satisfied and all lower layer constraints are violated (Figure 2.1(b)) and in M_2 all constraints of a lower layer than n are satisfied and one constraint is violated in layer n (Figure 2.1(a)).

Given a set of candidate models, we first choose those models for which the highest layer in which weak constraints are violated is as low as possible.

Among the remaining models those are chosen which minimise the sum of weights of violated weak constraints in the highest layer in which weak constraints are violated. Among these models we choose those which minimise the sum of weights of violated weak constraints in the highest layer in which the



(a) Model which violates one weak constraint in layer n

(b) Model which violates all weak constraints in layers below n

Figure 2.1: Extreme Case of Weak Constraint Violations

sums of weights of violated constraints differ. This hierarchical choice process is continued until all layers have been considered.

Thus, for the remaining models, call them *preferred models*, the following holds:

1. The highest layer in which weak constraints are violated is equal and minimal for all of these models.

2. The layers in which weak constraints are violated are the same for all of these models.

3. The sum of weights of violated weak constraints in each layer in which weak constraints are violated is minimal among the models for which the sum of weights of violated weak constraints is minimal in all layers above the considered one and thus equal for all of these models.

We will define the preferred models of a program formally below, since we need some preliminary definitions for it.

First we need a formal method to determine all abstract constraints ($\equiv$ weak constraints) of some abstract program of some layer ($\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}$), we also need to determine the layers occurring in a program ($\mathtt{Layers}(\mathcal{P})$), and the greatest layer ($l^{\mathcal{P}}_{max}$) in a program.

Definition 2.6.21 (Layers)
Given a grounded abstract program $\mathcal{P} = \mathtt{grounding}(\mathtt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in$

$\Pi_{\text{DATALOG}^{not,\vee,w}}$, *we define the following:*

$$\texttt{c}_{\texttt{layer}_\mathcal{P}} : \mathbb{N} \rightarrow \mathbb{P}(\textit{Constraints}_{\textit{abstract}})$$
$$\texttt{c}_{\texttt{layer}_\mathcal{P}}(l) = \{(C, l, w) \mid (C, l, w) \in \mathcal{P}\}$$
$$\texttt{Layers} : \textit{Programs}_{\textit{abstract}} \rightarrow \mathbb{P}(\mathbb{N})$$
$$\texttt{Layers}(\mathcal{P}) = \{l \mid (C, l, w) \in \mathcal{P}\}$$

$$l^\mathcal{P}_{max} = \max_{l \in \texttt{Layers}(\mathcal{P})} l$$

■

We also need to determine which weak constraints are violated by a given candidate model.

Definition 2.6.22 (Violated Abstract (Weak) Constraints in a particular layer) *Given a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *and a candidate model* $M \in SEM(\mathcal{P})$, *the set of violated constraints of layer* l *is defined as*

$$N^{M,\mathcal{P}}_l = \{C \mid M \not\models C, C \in \texttt{c}_{\texttt{layer}_\mathcal{P}}(l)\}$$

■

We also define the largest ($w^\mathcal{P}_{max}$) and smallest ($w^\mathcal{P}_{min}$) weight in a program (we will need that later), and the weight associated to an abstract constraint:

Definition 2.6.23 (Weights) *Given a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *let*

$$w^\mathcal{P}_{max} = \max_{(C,l,w) \in \mathcal{P}} w$$
$$w^\mathcal{P}_{min} = \min_{(C,l,w) \in \mathcal{P}} w$$

For each $\mathcal{W} = (C_1, l_1, w_1)$ *we define* $w_\mathcal{W} = w_1$. ■

Intuitively, the preferred models for a program should be those models which are preferred for all layers in this program. As we will see, the set preferred models of a layer are defined recursively and the set of preferred models for each layer monotonically decreases, starting with the set of all candidate models, which is a superset of the set of preferred models of the highest layer, down to the set of all preferred models of the lowest layer.

Therefore, the set of preferred models of the lowest layer is a subset of the set of preferred models for an arbitrary layer. It follows that the set of preferred models of the lowest layer coincides with the set of preferred models for the whole program.

Definition 2.6.24 (Formal Definition of Preferred Models) *A candidate model of a grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, $M \in SEM(\mathcal{P})$, *is a* preferred model, *iff the sum of weights*

of weak constraints which are violated by M is minimal for the highest layer among all candidate models, and for all lower layers the summarized weights of weak constraints violated by M is minimal among those models for which this criterion holds for all higher layers.

As motivated above, we define the models which are preferred in an arbitrary layer. This is a bounded recursive definition, where the base case is the highest layer in the program. The recursion is defined over the layers in decreasing order.

The criterion for preferred models in any layer is the minimisation of the summarized weights of the weak constraints which are violated by the respective model out of the set from which the preferred models of the layer are determined.

For the base case of the highest layer this latter set of models coincides with the set of all candidate models. For any other layer this set is the set of preferred models determined in the layer immediately above the considered layer.

We formulate the base case:

$$PREF_{SEM}(\mathcal{P}, l^{\mathcal{P}}_{max}) = \{\, M \in SEM(\mathcal{P}) \mid \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}} = \min_{M_c \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N^{M_c,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}} \}$$

The (bounded) recursive step is as follows:

$$\forall i \in \texttt{Layers}(\mathcal{P}) - \{l^{\mathcal{P}}_{max}\} : PREF_{SEM}(\mathcal{P}, i) =$$
$$\{M \in PREF_{SEM}(\mathcal{P}, i+1) \mid \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{i}} w_{\mathcal{W}} = \min_{M_c \in PREF_{SEM}(\mathcal{P}, i+1)} \sum_{\mathcal{W} \in N^{M_c,\mathcal{P}}_{i}} w_{\mathcal{W}} \}$$

Having defined the set of preferred models for each layer, we now generalize this notion to the set of preferred models for the entire program $\mathcal{P}$, as indicated above:

$$PREF_{SEM}(\mathcal{P}) = PREF_{SEM}(\mathcal{P}, 1)$$

■

Chapter 3

Representing Knowledge in DATALOG$^{\neg,not,\vee,w}$

In this chapter we will give several examples of domains, in which problems occur which can elegantly be represented using DATALOG$^{\neg,not,\vee,w}$. We picked these problems such that the benefit of having weak constraints with layer and weight information becomes evident. Keep in mind that any problem up to a certain complexity bound (at least up to Σ_2^P problems, since DATALOG$^{not,\vee}$ captures this class [EGM97]) can be represented with DATALOG$^{\neg,not,\vee,w}$.

The choice of the problems in this chapter should also reflect the practical relevance of the DATALOG$^{\neg,not,\vee,w}$ language. We will proceed by starting at the very wide field of *abductive reasoning* problems Section 3.1, giving translations for several classes of such reasoning problems – then we will move on to the more restricted domain of *school timetabling* problems, where we will describe the encoding of one rather general kind of problem and also of a particular refinement of it, which has been published recently. Finally we will pick three closely interrelated problems of *graph theory* and present different ways of encoding them.

3.1 Abduction

Abduction is an important way of reasoning, for example it is the methodology behind *diagnostic* reasoning. Given a set of rules or theories and some observations, abductive reasoning tries to find out the reasons for the observations, given that the rules are correct. These reasons are also called explanations.

Its counterparts, *induction* and *deduction* can also be formulated using this terminology: *induction* means that given a set of reasons and a set of observations, the rules should be found; *deduction* means that reasons and rules are given, and the observations (or consequences) should be determined.

As *induction* and *deduction*, *abduction* has been recognised as a basic principle of common-sense reasoning, and many tasks of everyday life are tackled and solved by using this approach. The philosopher Charles Sanders Peirce has to be given credit for identifying this way of reasoning.

Abductive Logic Programming is one of the possible formulations of abduction. In this case, the theory is represented by a logic program, and the

observations are represented by ground literals. The task is then to find a set of explanations, which are represented by atoms, so that the explanations, the logic program, and the observations are consistent. In practice, it is feasible that the possible explanations are restricted to a given set of possible explanations, called *hypotheses*. (cf. [KKT93, EGL97, EG95])

Formally, we define an *abductive logic programming problem* (*LPAP*) as a triple of hypotheses, observations and the underlying theory, similar to [EGL97, EG95]:

Definition 3.1.1 (*LPAP*s)

$$\mathfrak{Lpap} = \{\langle Hyp, Obs, LP\rangle \mid Hyp \subseteq \mathfrak{Atoms}, Obs \subseteq \mathfrak{Literals}^{\mathtt{not}}, LP \in \Pi_{\text{DATALOG}^{not,\vee}}, \\ \mathtt{variables}_A(Hyp) \cup \mathtt{variables}_A(\mathtt{atoms}_L(Obs)) = \emptyset\}$$

■

The task is to find a suitable explanation, defined as follows:

Definition 3.1.2 (Explanation)
Given an LPAP $\langle Hyp, Obs, LP\rangle \in \mathfrak{Lpap}$*, for an explanation* E *the following must hold:*

$$E \subseteq Hyp$$
$$\exists M \in SM(LP \cup E) : Obs \subseteq M$$

■

So an explanation must be a subset of the hypotheses (as motivated above), and additionally all observations must be true in one of the stable models of the explanation together with the theory.

So an *LPAP* may have no, one, or many explanations because there may be no, one, or many stable models. Usually one wants to find a minimal explanation. There are multiple minimality criteria which all make sense. We will consider *Minimum Cardinality*, *Prioritisation*, and *Penalisation* [EG95].

3.1.1 Minimum Cardinality Abduction

This approach tries to minimise the number of hypotheses in the explanation, thereby preferring simpler explanations. It should be noted that all hypotheses can be in an explanation with uniform probability.

Definition 3.1.3 (Minimum Cardinality)
Given an LPAP $\langle Hyp, Obs, LP\rangle$*, an explanation* $E \subseteq Hyp$ *satisfies the minimum cardinality criterion, iff no other explanation* $E' \subseteq Hyp$ *exists, such that* $|E'| < |E|$*, where* $|X|$ *denotes the cardinality of set* X*.* ■

We can encode an *LPAP* $\mathcal{P}$ by a DATALOG$^{\neg,not,\vee,w}$ program $\mathcal{P}^{minc}$, such that each explanation satisfying the minimum cardinality criterion is also an answer set of $\mathcal{P}^{minc}$.

Definition 3.1.4 (Encoding of Minimum Cardinality *LPAP*s)
Given an LPAP $\mathcal{P} = \langle Hyp, Obs, LP \rangle$, *we define* $\mathcal{P}^{minc}$ *as:*

$$\mathcal{P}^{minc} = LP \cup \{h \vee \neg\, h. \mid h \in Hyp\} \tag{3.1}$$
$$\cup \{\leftarrow a. \mid \texttt{not}\ a \in Obs\} \tag{3.2}$$
$$\cup \{\leftarrow \texttt{not}\ a. \mid a \in Obs \cap \mathfrak{Atoms}\} \tag{3.3}$$
$$\cup \{\Leftarrow h. \mid h \in Hyp\} \tag{3.4}$$

■

The first extension (3.1) to LP "generates" models for all subsets of Hyp. The strong constraints (3.2) and (3.3) are used to enforce that the observations are in the model. In the case of a negative observation (3.2), which means that there is no information about this observation, we do not want this observation to be entailed by LP and a possible explanation and vice versa. Finally, the weak constraints (3.4), which are all in the same layer and are weighted uniformly, state that an answer set is better than another one, if less atoms of the hypotheses are in it, since one such constraint is violated if the corresponding atom is in the model. It follows that the preferred answer sets contain the least number of hypotheses among all candidate answer sets and therefore the preferred answer sets correspond exactly to those explanations which have minimum cardinality.

Example 3.1.1
Consider a computer network in a larger company. Usually, such networks are comprised of several subnetworks, connected by suitable gateways or routers, and also bridges and repeaters on a lower level. We will refer to all of these devices as connectors, since they all connect several subnets in some way.

If such connectors fail, the company's network may be partitioned, which is an undesirable and critical situation and should be remedied as soon as possible.

To this end we might want to design an intelligent agent, which aids the technicians in finding and fixing the error. The task posed to this agent is an abductive problem: The observations are the facts that some subnets are unreachable from some others, the hypotheses are that some subset of the connectors failed. The theory is some formalisation of the topology of the company's network, possibly enhanced by additional knowledge (for instance which of the devices are connected to the same power circuit etc.).

Consider the network topology depicted in Figure 3.1 on the following page. The network consists of three Ethernet subnets (eth1, eth2, eth3), two token ring subnets (tr1, tr2), and several connectors (c1, c2, c3, c4). Note that in general these graphs are bipartite (no direct connections between subnets or between connectors exist). We therefore represent the topology using a relation connected(S, C), which contains a tuple if subnet S is connected to connector C.

The representation of our example network is shown in Figure 3.2 on the following page. The theory behind the network is that one subnet is directly reachable (represented by subconnect/2) from another one, if both subnets are connected to the same connector, and this connector is not broken. We generalise from direct reachability to general reachability (represented by subreachable/2) by standard transitive closure. The resulting theory (including predicates which

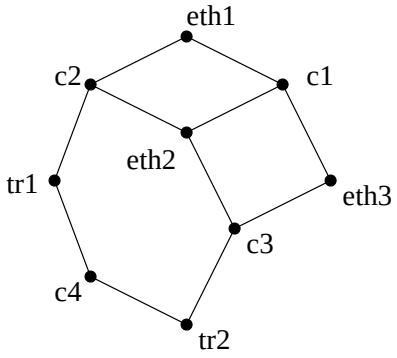


Figure 3.1: Example Network Topology

$$
\begin{array}{ll}
\mathit{connected}(\mathit{eth}1, c1). & \mathit{connected}(\mathit{eth}1, c2). \\
\mathit{connected}(\mathit{eth}2, c1). & \mathit{connected}(\mathit{eth}2, c2). \\
\mathit{connected}(\mathit{eth}2, c3). & \mathit{connected}(\mathit{eth}3, c1). \\
\mathit{connected}(\mathit{eth}3, c3). & \mathit{connected}(\mathit{tr}1, c2). \\
\mathit{connected}(\mathit{tr}1, c4). & \mathit{connected}(\mathit{tr}2, c3). \\
\mathit{connected}(\mathit{tr}2, c4). &
\end{array}
$$

Figure 3.2: Representation of the network shown in Figure 3.1

allow us to identify subnets and connectors, respectively) is depicted in Figure 3.3. Let us call all facts and rules of Figure 3.2 and Figure 3.3 LP_{net}.

Now the hypotheses our agent has is that some of the connectors are broken for some reason. We will refer to this as $Hyp_{net} = \{\mathit{broken}(c1), \mathit{broken}(c2), \mathit{broken}(c3), \mathit{broken}(c4)\}$.

Note that this theory is very simplified: For instance, it does not capture the possibility that a subnet is broken in case of failures – but extending the theory to capture additional sources of failure is straightforward.

Now assume a situation in which we observe that subnet $tr2$ cannot be reached from $eth1$. So let $Obs_{net} = \{\texttt{not}\ \mathit{subreachable}(\mathit{eth}1, \mathit{tr}2)\}$.

$$
\begin{aligned}
\mathit{subconnect}(S_1, S_2) &\leftarrow \mathit{connected}(S_1, C), \mathit{connected}(S_2, C), \texttt{not}\ \mathit{broken}(C). \\
\mathit{subreachable}(S_1, S_2) &\leftarrow \mathit{subconnect}(S_1, S_2). \\
\mathit{subreachable}(S_1, S_2) &\leftarrow \mathit{subreachable}(S_1, S_3), \mathit{subconnect}(S_3, S_2).
\end{aligned}
$$

$$
\begin{aligned}
\mathit{subnet}(S) &\leftarrow \mathit{connected}(S, C). \\
\mathit{connector}(C) &\leftarrow \mathit{connected}(S, C).
\end{aligned}
$$

Figure 3.3: Theory for network diagnosis problems

$$\begin{aligned}\mathcal{P}_{net}^{minc} = LP_{net} \cup \{&broken(c1) \vee \neg broken(c1).,\\ &broken(c2) \vee \neg broken(c2).,\\ &broken(c3) \vee \neg broken(c3).,\\ &broken(c4) \vee \neg broken(c4).,\\ &\leftarrow subreachable(eth1, tr2).,\\ &\Leftarrow broken(c1).,\\ &\Leftarrow broken(c2).,\\ &\Leftarrow broken(c3).,\\ &\Leftarrow broken(c4).\}\end{aligned}$$

Figure 3.4: The DATALOG$^{\neg,not,\vee,w}$ program which solves the minimum cardinality abduction problem $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$

In order to give possible explanations to this malfunction, the agent has to solve the LPAP $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$.

If we have no additional knowledge about the stability of the connectors, we may assume that the probability of their failure is uniformly distributed, and thus resort to minimal cardinality *abduction. Using the translation defined in Definition 3.1.4, the solution of this problem can be found by determining the preferred models of the* DATALOG$^{\neg,not,\vee,w}$ *program* $\mathcal{P}_{net}^{minc}$ *depicted in Figure 3.4.*

It can be verified that the preferred answer sets of $\mathcal{P}_{net}^{minc}$ *are* $\{broken(c1)$, $broken(c2)$, $\neg\, broken(c3)$, $\neg\, broken(c4)\}$, $\{\neg\, broken(c1), broken(c2), broken(c3)$, $\neg\, broken(c4)\}$, $\{\neg\, broken(c1)$, $\neg\, broken(c2)$, $broken(c3)$, $broken(c4)\}$. *There are other consistent answer sets which are not preferred, since they violate more weak constraints. For example, all connectors could be broken. While not impossible, it is a worst case scenario and a technician should first try to make additional observations before concluding this.* ♦

3.1.2 Priority Minimal Abduction

The minimum cardinality approach can be refined by partitioning the hypotheses in groups of different priorities. An explanation is better than another using this notion, if it contains less hypotheses from the lowest priority set and among the explanations for which this number is minimal, an explanation is better than another if it contains less hypotheses of the second lowest priority set, and so on. So it is less likely that a hypothesis from a lower priority set is in an explanation than one from a higher priority set [EG95].

In order to encode such problems, we simply add the priority layer information to the weak constraints, otherwise leaving Definition 3.1.4 untouched. Note that in our framework higher priorities are more important, whereas in the context of Priority Minimal Abduction lower priorities are more important. To be able to deal with this, we reverse the order of priorities: If we have n priority sets, priority i becomes $n - i + 1$ in the reversed ordering.

$$
\begin{aligned}
\mathcal{P}^{pmin}_{net} = LP_{net} \cup \{ & broken(c1) \vee \neg broken(c1)., \\
& broken(c2) \vee \neg broken(c2)., \\
& broken(c3) \vee \neg broken(c3)., \\
& broken(c4) \vee \neg broken(c4)., \\
& \leftarrow subreachable(eth1, tr2)., \\
& \Leftarrow broken(c1).[1:], \\
& \Leftarrow broken(c2).[2:], \\
& \Leftarrow broken(c3).[2:], \\
& \Leftarrow broken(c4).[1:]\}
\end{aligned}
$$

Figure 3.5: The DATALOG$^{\neg,not,\vee,w}$ program which solves the priority minimal abduction problem $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$ under prioritisation $H_1 = \{broken(c2), broken(c3)\}$, $H_2 = \{broken(c1), broken(c4)\}$

Definition 3.1.5 (Encoding of Priority Minimality *LPAP*s)
Given an LPAP $\mathcal{P} = \langle Hyp, Obs, LP\rangle$ *and a prioritisation* $H_1, \ldots, H_n$ *of* H *(a partition of* H*), we define* $\mathcal{P}^{pmin}$ *as:*

$$
\begin{aligned}
\mathcal{P}^{pmin} = LP \cup & \{h \vee \neg\, h. \mid h \in Hyp\} \\
& \cup \{\leftarrow a. \mid \texttt{not}\ a \in Obs\} \\
& \cup \{\leftarrow \texttt{not}\ a. \mid a \in Obs \cap \mathfrak{Atoms}\} \\
& \cup \{\Leftarrow h.[n-i+1:] \mid h \in H_i\}
\end{aligned}
$$

■

Example 3.1.2
We extend Example 3.1.1. For instance, assume that connectors $c2$ *and* $c3$ *have an independent auxiliary power supply which also does power regulation, while* $c1$ *and* $c4$ *do not have this feature.*

In this light it is much less likely that $c2$ *or* $c3$ *fails. We might express this additional information by giving the prioritisation* $H_1 = \{broken(c2), broken(c3)\}$, $H_2 = \{broken(c1), broken(c4)\}$ *for* Hyp_{net}.

The LPAP $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$ *using priority minimal abduction with* H_1, H_2 *can be translated into a* DATALOG$^{\neg,not,\vee,w}$ *program, as described in Definition 3.1.5. The resulting program* $\mathcal{P}^{pmin}_{net}$ *is depicted in Figure 3.5.*

The preferred answer sets of $\mathcal{P}^{pmin}_{net}$ *and thus explanations are* $\{broken(c1), broken(c2), \neg\ broken(c3), \neg\ broken(c4)\}$ *and* $\{\neg\ broken(c1), \neg\ broken(c2), broken(c3), broken(c4)\}$. *Note that* $\{\neg\ broken(c1), broken(c2), broken(c3), \neg\ broken(c4)\}$, *which was an explanation under minimum cardinality abduction, is not an explanation here. Indeed, it is less likely that both connectors with auxiliary power supply and stabilisation system fail, than just one of them.* ♦

3.1.3 Penalisation-based Abduction

Finally, penalisation is yet another refinement defined by assigning a penalty to each hypothesis. The preferred explanation is then the one which minimises the added penalties.

$$
\begin{aligned}
\mathcal{P}^{penmin}_{net} = LP_{net} \cup \{ & broken(c1) \vee \neg broken(c1).,\\
& broken(c2) \vee \neg broken(c2).,\\
& broken(c3) \vee \neg broken(c3).,\\
& broken(c4) \vee \neg broken(c4).,\\
& \leftarrow subreachable(eth1, tr2).,\\
& \Leftarrow broken(c1).[: 50000],\\
& \Leftarrow broken(c2).[: 20000],\\
& \Leftarrow broken(c3).[: 2000],\\
& \Leftarrow broken(c4).[: 200]\}
\end{aligned}
$$

Figure 3.6: The DATALOG$^{\neg,not,\vee,w}$ program which solves the penalisation-based abduction problem $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$ under the penalties defined in the text

This is straightforward using our weight mechanism. We simply reuse Definition 3.1.4, and add the penalties to the weak constraints.

Definition 3.1.6 (Encoding of Penalisation Minimality *LPAP*s)
Given an LPAP $\mathcal{P} = \langle Hyp, Obs, LP\rangle$ *and for each* $h \in Hyp$ *a penalty* w^h*, we define* $\mathcal{P}^{penmin}$ *as:*

$$
\begin{aligned}
\mathcal{P}^{penmin} = LP \cup & \{h \vee \neg\, h. \mid h \in Hyp\}\\
& \cup \{\leftarrow a. \mid \texttt{not}\ a \in Obs\}\\
& \cup \{\leftarrow \texttt{not}\ a. \mid a \in Obs \cap \mathfrak{Atoms}\}\\
& \cup \{\Leftarrow h.[: w^h] \mid h \in Hyp\}
\end{aligned}
$$

■

Example 3.1.3
Consider again Example 3.1.1 and let us forget what we know about the auxiliary power systems, but focus on the types of connectors. Usually manufacturers of hardware associate a number called mean time to failure (MTTF) to their products. This is a probabilistic expectancy value of how long the piece of hardware will stay operational. We can use this to assign a penalisation to the connectors: The MTTF minus the time the respective connector has been used is a measure of probability for the connector to fail. The lower this value, the higher the probability of failure.

In our example, assume that $c1$ *is brand new and our measure is* $w^{broken(c1)} = 50000$[1]*,* $c2$ *and* $c4$ *are the same type and are in use for the same time, and* $w^{broken(c2)} = w^{broken(c4)} = 20000$*, and* $c3$ *is rather old, soon reaching the MTTF,* $w^{broken(c3)} = 200$*.*

With this new information, the LPAP $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$ *can be translated to the* DATALOG$^{\neg,not,\vee,w}$ *program* $\mathcal{P}^{penmin}_{net}$ *in Figure 3.6 by the transformation presented in Definition 3.1.6.*

[1]MTTF is usually given in hours

$$\begin{aligned}\mathcal{P}_{net}^{comb} = LP_{net} \cup \{ & broken(c1) \vee \neg broken(c1).,\\ & broken(c2) \vee \neg broken(c2).,\\ & broken(c3) \vee \neg broken(c3).,\\ & broken(c4) \vee \neg broken(c4).,\\ & \leftarrow subreachable(eth1, tr2).,\\ & \Leftarrow broken(c1).[1:50000],\\ & \Leftarrow broken(c2).[2:20000],\\ & \Leftarrow broken(c3).[2:20000],\\ & \Leftarrow broken(c4).[1:200]\}\end{aligned}$$

Figure 3.7: The DATALOG$^{\neg,not,\vee,w}$ program which solves the penalisation-based abduction problem $\langle Hyp_{net}, Obs_{net}, LP_{net}\rangle$ under the penalties defined in the text

The preferred answer sets of $\mathcal{P}_{net}^{penmin}$ *are* $\{\neg\ broken(c1), broken(c2), broken(c3), \neg\ broken(c4)\}$ *and* $\{\neg\ broken(c1), \neg\ broken(c2), broken(c3), broken(c4)\}$. $\{broken(c1), broken(c2), \neg\ broken(c3), \neg\ broken(c4)\}$ *is not preferred, and indeed the probability of failure is lower.* ♦

Note that in our approach the combination of prioritisation and penalisation is also possible and straightforward, as suggested in [EG95].

Example 3.1.4
Consider again Example 3.1.1 and combine the knowledge introduced in 3.1.2 and 3.1.3.

The corresponding DATALOG$^{\neg,not,\vee,w}$ *program is shown in Figure 3.7.*

The only preferred answer set of $\mathcal{P}_{net}^{comb}$ *is* $\{\neg\ broken(c1), \neg\ broken(c2), broken(c3), broken(c4)\}$, *which matches intuition.* ♦

3.2 Planning

3.2.1 Automated Timetabling

From the large field of Planning, we pick out a particular subfield, which is Automated Timetabling. We will consider timetabling in schools, but this type of problem naturally occurs in many other situations, too.

Automated Timetabling has evolved as a separate, highly specialised field of research. There are special conferences on this topic [BR96], and one can observe a considerable increase of research papers on this topic [Bar96]. [Sch95] gives a comprehensive survey of the developments in this area. Most publications consider school or university timetabling problems.

There are several related instances of school timetabling problems, which somehow vary in complexity, but most of the relevant problems are NP-complete [CK96, Sch95].

3.2.2 School Timetabling

TTP1: The Basic Problem

Given a set of m classes $C = \{c_1, \ldots, c_m\}$, a set of n teachers $T = \{t_1, \ldots, t_n\}$, a set of p periods $P = \{p_1, \ldots, p_p\}$, and a set of requirements $R = \{(c_c, t_t, r_{c,t}) \mid c_c \in C, t_t \in T\}$, which contains triples which state that teacher t_t has to give $r_{c,t}$ lectures to class c_c, the problem is to assign lectures to periods in such a way that no teacher and no class is involved in more than one lecture during the same period, and that all of the teaching requirements are met.

More formally: Find a set $A \subseteq \{(c_c, t_t, p_p) \mid c_c \in C, t_t \in T, p_p \in P\}$, such that:

$$\forall (c_i, t_j, r_{i,j}) \in R: \quad \sum_{p_x \in P} \chi_A((c_i, t_j, p_x)) = r_{i,j} \tag{3.5}$$

$$\forall c \in C, p \in P: \quad (c, t_x, p) \in A \wedge (c, t_y, p) \in A \Longrightarrow x = y \tag{3.6}$$

$$\forall t_t \in T, p_p \in P: \quad (c_x, t, p) \in A \wedge (c_y, t, p) \in A \Longrightarrow x = y \tag{3.7}$$

where χ_M is the characteristic function for M, defined in the usual way as

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

Let us call this problem TTP1, as in [Sch95].

It should be noted that this problem is solvable in polynomial time [Sch95].

Formulation of TTP1 in DATALOG$^{not,\vee,w}$

The only feature which does not have a straightforward representation is the number of lectures given by some teacher to some class, in particular the constraints described by 3.5. There is no means to express something like "count the number of atoms looking like ..." in DATALOG$^{not,\vee,w}$, so we have to work around this.

We assign numbers to the lectures of a particular teacher given to a class. In other words, we assume that in our database facts $r(c_i, t_j, k)$ are stored, if $0 < k \leq r_{i,j}$ holds. In other words, $r(c, t, k)$ is in the database, if teacher t has to teach at least k periods to class c. Further we store the periods as $p(p)$, if $p \in P$.

The task is now to find a suitable assignment. We simply state that a teacher teaches a class the kth lecture at a given period or (s)he does not.

$$a(C, T, P, K) \vee na(C, T, P, K) \leftarrow p(P), r(C, T, K). \tag{3.8}$$

Rule 3.8 obviously guesses all subsets of possible assignments. We can spot a potential problem here: The fourth argument of a is induced by our representation and doesn't mean anything in the original problem, but we need it for discriminating between the different lectures. We will deal with this below.

It is clear that each lecture should be given only once, so there may not be two assignments which are identical up to the period.

$$\leftarrow a(C,T,P1,K), a(C,T,P,K), P \neq P1. \tag{3.9}$$

Note that we may consider $\neq$ a built-in predicate, which discards every ground instances where its arguments are equal during the grounding process and does not appear any longer in resulting ground abstract rules.

We must ensure that each lecture is in the timetable – there may be no requirement without the corresponding lecture being assigned:

$$planned(C,T,K) \leftarrow a(C,T,P,K). \tag{3.10}$$

$$\leftarrow r(C,T,K), \texttt{not}\ planned(C,T,K). \tag{3.11}$$

Also lectures with different numbers must not be given during the same period:

$$\leftarrow a(C,T,P,K), a(C,T,P,K1), K \neq K1. \tag{3.12}$$

There is still a problem: Due to the representation involving the number of the lecture there are many solutions in which just this lecture number is permuted.

We can identify classes of models with respect to this permutation and designate one model which represents the class. To this end we need an ordering over the class numbers and periods. In this way we can enforce that the first lecture of teacher t given to class c is not given after the second lecture of teacher t to class c and so on. Note that "after" in the last sentence implies an ordering of the periods and "first", "second", etc. implies an ordering of the lectures in the scope of a single teacher and class.

How can we obtain such an ordering? Either we assume that both periods and numbers of lectures are represented numerically and that there is some built-in predicate "$<$". We could then generate a range-restricted version of $<$ to periods or lecture numbers: $p\_gt(P1,P2) \leftarrow p(P1), p(P2), P1 < P2.$ and $k\_gt(K1,K2) \leftarrow r(C1,T1,K1), r(C2,T2,K2), K1 < K2.$.

Another way would be to alter the representation of the periods: Instead of representing them with a simple unary relation, we could represent the successor relation over the available periods. We would then store $p\_s(p_1,p_2)$, $p\_s(p_2,p_3)$, $\ldots$, $p\_s(p_{p-1},p_p)$ if we have p periods available. From this, we can easily deduce an ordering by transitive closure: $p\_gt(P1,P2) \leftarrow p\_s(P1,P2).$ and $p\_gt(P1,P3) \leftarrow p\_gt(P1,P2), p\_s(P2,P3).$.

For the lecture numbering, we would need an additional predicate representing the successor relation on this domain, that is $k\_s(1,2)$, $\ldots$, $k\_s(r_{max}-1, r_{max})$, where r_{max} indicates the maximum number of lectures to be taught to one class by a single teacher. The ordering can then be obtained analogously to the periods: $k\_gt(K1,K2) \leftarrow k\_s(K1,K2).$ and $k\_gt(K1,K3) \leftarrow k\_gt(K1,K2), k\_s(K2,K3).$.

In either case we will write $A < B$ instead of $p\_gt(A,B)$ and $k\_gt(A,B)$ in the sequel for better readability. So we add the constraint

$$\begin{aligned}
\mathcal{P}_{\texttt{TTP1}} = \{\ & a(C,T,P,K) \vee na(C,T,P,K) \leftarrow p(P), r(C,T,K). && (3.8)\\
& \leftarrow a(C,T,P1,K), a(C,T,P,K), P \neq P1. && (3.9)\\
& \mathit{planned}(C,T,K) \leftarrow a(C,T,P,K). && (3.10)\\
& \leftarrow r(C,T,K), \texttt{not}\ \mathit{planned}(C,T,K). && (3.11)\\
& \leftarrow a(C,T,P,K), a(C,T,P,K1), K \neq K1. && (3.12)\\
& \leftarrow a(C,T,P,K), a(C,T,P1,K1), P < P1, K1 < K. && (3.13)\\
& \leftarrow a(C,T1,P,K1), a(C,T2,P,K2), T1 \neq T2. && (3.14)\\
& \leftarrow a(C1,T,P,K1), a(C2,T,P,K2), C1 \neq C2.\} && (3.15)
\end{aligned}$$

Figure 3.8: The program for `TTP1`

$$\leftarrow a(C,T,P,K), a(C,T,P1,K1), P < P1, K1 < K. \tag{3.13}$$

which ensures that a lecture with a greater number is not given at an earlier period than a lecture with a lower number.

We have now modelled the first Constraints, 3.5. Indeed, the modelling was not trivial, but still straightforward.

The constraints defined by 3.6 mean that no two teachers should teach the same class during the same period. Note that the number of the lecture is not relevant.

$$\leftarrow a(C,T1,P,K1), a(C,T2,P,K2), T1 \neq T2. \tag{3.14}$$

Similarly, 3.7 (the same teacher can not give a lecture to two distinct classes simultaneously) is expressed as:

$$\leftarrow a(C1,T,P,K1), a(C2,T,P,K2), C1 \neq C2. \tag{3.15}$$

Let us call the resulting program $\mathcal{P}_{\texttt{TTP1}}$; it is shown in Figure 3.8.

`TTP2`: Extension to the Basic Problem

Usually, not all teachers are available at all periods, and the same may hold for classes, too. So, in addition to the constraints in `TTP1`, we have to ensure that no lectures are scheduled at periods where either the teacher or the class is unavailable.

To this end, we define the sets of unavailable periods for each teacher and each class: $\forall c \in C : U_c \subseteq P$, $\forall t \in T : U_t \subseteq P$.

$$\begin{aligned}
\forall c \in C, t \in T: \quad & p \in U_c \Longrightarrow (c,t,p) \notin A && (3.16)\\
\forall c \in C, t \in T: \quad & p \in U_t \Longrightarrow (c,t,p) \notin A && (3.17)
\end{aligned}$$

We call this problem `TTP2`, as in [Sch95]. Note that with this extension, the problem becomes NP-complete (cf. [CK96, Sch95]).

$$
\begin{aligned}
\mathcal{P}_{\texttt{TTP1}} = \{\ & a(C,T,P,K) \vee na(C,T,P,K) \leftarrow p(P), r(C,T,K). && (3.8)\\
& \leftarrow a(C,T,P1,K), a(C,T,P,K), P \neq P1. && (3.9)\\
& \mathit{planned}(C,T,K) \leftarrow a(C,T,P,K). && (3.10)\\
& \leftarrow r(C,T,K), \texttt{not}\ \mathit{planned}(C,T,K). && (3.11)\\
& \leftarrow a(C,T,P,K), a(C,T,P,K1), K \neq K1. && (3.12)\\
& \leftarrow a(C,T,P,K), a(C,T,P1,K1), P < P1, K1 < K. && (3.13)\\
& \leftarrow a(C,T1,P,K1), a(C,T2,P,K2), T1 \neq T2. && (3.14)\\
& \leftarrow a(C1,T,P,K1), a(C2,T,P,K2), C1 \neq C2. && (3.15)\\
& \leftarrow \mathit{class\_unavailability}(C,P), a(C,T,P,K). && (3.18)\\
& \leftarrow \mathit{teacher\_unavailability}(T,P), a(C,T,P,K).\} && (3.19)
\end{aligned}
$$

Figure 3.9: The program for TTP1

Formulation of TTP2 in DATALOG$^{not,\vee,w}$

Representing unavailabilities is rather straightforward. We require them to be contained in two relations, $\mathit{class\_unavailability}/2$ and $\mathit{teacher\_unavailability}/2$, defined for each pair of class (or teacher, respectively) and period, in which it (or (s)he, respectively) is not available.

We then simply extend $\mathcal{P}_{\texttt{TTP1}}$ by the following constraint for class unavailability, stating that it is not allowed to assign a lecture to a class at during a period, in which it is unavailable, corresponding to 3.16:

$$\leftarrow \mathit{class\_unavailability}(C,P), a(C,T,P,K). \tag{3.18}$$

In analogy we define one for unavailabilities of teachers, corresponding to 3.17:

$$\leftarrow \mathit{teacher\_unavailability}(T,P), a(C,T,P,K). \tag{3.19}$$

Let us call the resulting program $\mathcal{P}_{\texttt{TTP2}}$. It is depicted in Figure 3.9.

TTOP2: Optimisation Problem to TTP1 and TTP2

While unavailabilities are usually constraints which have to be met strictly, there is a number of additional constraints, which are not that strict, but more or less desirable.

A simple approach would be to assign a desirability factor to each triple of class, teacher, and period. This is rather general and the representation in our framework is straightforward by using weights, yet it is not capable of expressing various desiderata which occur in praxis. One example of such a desirable property, which cannot be modelled by assigning factors, is that lectures with the same teacher should be uniformly distributed over the week. Additionally it is very difficult, if not impossible, to represent different, possibly conflicting wishes in that way.

There are other approaches to specify desired properties of the timetable, which seem to be more rewarding. One is described in [CDM98] and identifies several classes of constraints in the school timetabling domain:

- Feasibility Conditions σ
- Didactic Requirements Δ
- Organisational Requirements Ω
- Teacher Preferences Π

1. σ is basically what we have defined as `TTP1`, with the additional constraint, that there should not be any "holes" in the generated timetable. This requires a grouping of periods as weekdays or similar.
2. Δ describes several desirable criteria, which a timetable should have from the pedagogical point of view. [CDM98] states the following constraints they encountered in the school they took as an example:
   - no more than x teaching hours a day for each teacher
   - a class should not have the same teacher every day during the last hour
   - uniform distribution of the hours with the same teacher over the week
   - some lectures should be given in pairs of periods
3. Ω are requirements which are useful from the organisational point of view:
   - at least 2 teaching hours per day for each teacher
   - as few holes as possible in the teachers' schedules
   - concentrate holes on one day preferably (for parent-teacher meeting hours)
   - do not assign all available teachers in one period (to leave room for supplements in case of illness)
4. Π are each teacher's individual preferences, e.g.
   - some teachers do not like teaching in the morning
   - some want to teach in the morning rather than in the afternoon
   - a particular day off
   - there is a teacher ranking – preferences of teachers in higher ranks should be preferred to those of lower-rank teachers

[CDM98] describe the objective function to this problem as

$$\alpha \cdot \mathcal{I} + \beta_1 \cdot s_\Delta + \beta_2 \cdot s_\Omega + \beta_3 \cdot s_\Pi \tag{3.20}$$

where s_Δ, s_Ω, and s_Π are the weighted sums of violated constraints in the respective constraint-classes, $\mathcal{I}$ is the number of violated constraints in σ, called "infeasibilities" by [CDM98]. $\alpha, \beta_1, \beta_2, \beta_3$ are weights, where "$\alpha \gg \beta_1 \approx \beta_2 \approx$

β_3 induces a hierarchical structure". In addition they state that Δ and Ω constraints should be more important than Π constraints.

In our opinion, this approach goes into the right direction, albeit the representation is not optimal. The main point of criticism from the representational point of view is that compulsory constraints are represented in the same way as the desired properties. Also the approach of choosing a large constant for the strong constraints is not very general and might fail sometimes.

Formulation of `TTOP2` in DATALOG$^{not,\vee,w}$

As suspected, we will use our weak constraint system and fully exhaust the possibilities of weights and layers.

But first we need to define the notion of days, which occurs throughout the description of `TTOP2`. So, what we need to do is to require an input relation which relates a period to a weekday. Let us call it *period_day*/2.

Using this predicate, the additional "hard" constraint in σ, which states that there should be no "holes" in the schedules of classes during days can be formulated as follows:

$$planned1(C, P) \leftarrow a(C, T, P, K). \tag{3.21}$$

$$\begin{aligned} &\leftarrow period\_day(P1, D), period\_day(P2, D), period\_day(P3, D), \\ &\quad P1 < P2), P2 < P3, \\ &\quad a(C, T1, P1, K1), \texttt{not}\ planned_1(C, P2), a(C, T3, P3, K3). \end{aligned} \tag{3.22}$$

Constraint 3.22 states that given 3 periods of the same day, one class should not have a lecture during the earliest of these 3 periods, then have no lecture in the middle one, and then again have some lecture in the greatest of these periods. Note that we had to introduce the new predicate *planned*1 because we needed to express that no lesson is taught to a particular class at a fixed period.

So much for the compulsory constraints. Δ, Ω, and Π contain desirable criteria of different importance – both on the level of these classes and also within them. [CDM98] suggests to view Δ and Ω as on about the same level of importance and Π to be less important. However, this choice is arbitrary and we believe that didactic requirements should in general be more important than organisational ones. So we will create a hierarchy of three levels of non-strict constraints instead of two.

Concerning Δ, we introduce the following weak constraints, in order of appearance above:

- "No more than x teaching hours a day for each teacher." If x is 4, the

corresponding constraint is then

$$\begin{aligned}
\Leftarrow\ & period_day(P1, D), period_day(P2, D), period_day(P3, D), \\
& \quad period_day(P4, D), period_day(P5, D), \\
& a(C1, T, P1, K1), a(C2, T, P2, K2), a(C3, T, P3, K3), \\
& \quad a(C4, T, P4, K4), a(C5, T, P5, K5), \\
& P1 \neq P2, P1 \neq P3, P1 \neq P4, P1 \neq P5, \\
& P2 \neq P3, P_2 \neq P4, P2 \neq P5, \\
& P3 \neq P4, P3 \neq P5, \\
& P4 \neq P5. \qquad [3 : \Delta_1]
\end{aligned} \tag{3.23}$$

In general:

$$\begin{aligned}
\Leftarrow\ & period_day(P1, D), \ldots, period_day(P\mathtt{n}, D), \\
& a(C1, T, P1, K1), \ldots, a(C\mathtt{n}, T, P\mathtt{n}, K\mathtt{n}), \\
& P1 \neq P2, \ldots, P1 \neq P\mathtt{n}, \\
& \quad \vdots \\
& P\mathtt{n} - \mathtt{1} \neq P\mathtt{n}. \qquad [3 : \Delta_1]
\end{aligned}$$

Δ_1 is the weight relative to other didactic requirements.

- For the requirement "A class should not have the same teacher every day during the last hour" we have to define several predicates first, which allow us to express the notion "class I has teacher J in period P, which is the last lecture for class I on day D".

  We first define a predicate expressing "$a(C, T, P, K)$ is the last lecture on day D". The easiest way to do this is to write a predicate *notlast_lecture*, which is true for all assigned lectures on a certain day which are not the last one, that is, if a greater period to which a lecture is assigned exists on the same day. Then the last lectures are simply those which are not covered by *notlast_lecture*. In fact, we would only need class-teacher-day triples, but the information which period is the last one and which number this lecture has, which is redundant for this particular purpose, might be useful for other constraints, so we keep it.

$$\begin{aligned}
notlast_lecture(C, T, P, K, D) \leftarrow\ & a(C, T, P, K), period_day(P, D), \\
& a(C, T1, P1, K1), period_day(P1, D), \\
& P < P1. \qquad (3.24) \\
last_lecture(C, T, P, K, D) \leftarrow\ & a(C, T, P, K), period_day(P, D), \\
& \mathtt{not}\ notlast_lecture(C, T, P, K, D). \qquad (3.25)
\end{aligned}$$

  We need to determine those class-teacher pairs which are together in all the last lectures of this class. Clearly such a pair doesn't exist if the class has different teachers in its last lectures. So we might define a predicate $difflast/1$ which is true for those classes which have different lecturers in their last lectures. So if a class has teacher T in its last lecture, and

it does not have different teachers in its last lectures, teacher T is always with class C in its last lecture.

$$\begin{aligned} difflast(C) &\leftarrow last\_lecture(C,T,P,K,D), \\ &\qquad last\_lecture(C,T1,P1,K1,D1), T \neq T1. \qquad (3.26) \\ alwayslast(C,T) &\leftarrow last\_lecture(C,T,P,K,D), \texttt{not}\ difflast(C). \qquad (3.27) \end{aligned}$$

Now the weak constraint is easy to formulate:

$$\Leftarrow alwayslast(C,T). \qquad [3:\Delta_2] \qquad (3.28)$$

Again, Δ_2 is the weight relative to other didactic requirements.

- "Uniform distribution of the hours with the same teacher over the week"[2]. For any lectures given by teacher t to class c, for which it is desirable (we will see shortly that there are lectures for which this uniform distribution is undesirable) we add a constraint

$$\begin{aligned} &\Leftarrow a(c,t,P1,K1), a(c,t,P2,K2), \\ &\quad period\_day(P1,D), period\_day(P2,D). \qquad [3:\Delta_3] \end{aligned} \qquad (3.29)$$

  stating that two lectures should possibly not be scheduled on the same day. One probably does not want to do that with lectures of the type described next. At least, one should use a smaller weight for them, so in this case it makes sense to view Δ_3 as a range of values rather than a fixed one.

- "Some lectures should be given in pairs of periods", means that two lectures k_1, k_2 of some teacher c to class t should be given in consecutive periods on the same day. Otherwise expressed, there should not be any period between them, and they should not be scheduled on separate days. So we introduce two weak constraints:

$$\begin{aligned} &\Leftarrow a(c,t,P1,k_1), a(c,t,P2,k_2), \\ &\quad p\_gt(P1,P), p\_gt(P2,P). \qquad [3:\Delta_4] \qquad (3.30) \\ &\Leftarrow a(c,t,P1,k_1), a(c,t,P2,k_2), \\ &\quad period\_day(P1,D1), period\_day(P2,D1), \\ &\quad D1 \neq D2. \qquad [3:\Delta_4] \qquad (3.31) \end{aligned}$$

  Note that only consecutive lecture numbers should be chosen for these double periods.

So much for the didactic desiderata Δ. Of course, one might be able to find more such properties, these are just examples. Now we describe the formulation of the organisational constraints presented above:

[2]The original paper states that subjects should be uniformly distributed, but we did not define any subjects.

- "At least 2 teaching hours per day for each teacher". This should rather read "Not a singleton teaching hour on any day for a teacher", since teachers may have a free day. The criterion can then be formulated in a rather straightforward manner:

$$\begin{aligned} \mathit{severalhours}(T,D) \leftarrow\ & a(C,T,P,K), a(C1,T,P1,K1), \\ & \mathit{period\_day}(P,D), \mathit{period\_day}(P1,D), \\ & P \neq P1. \qquad (3.32) \\ \Leftarrow\ & a(C,T,P,K), \mathit{period\_day}(P,D), \\ & \texttt{not}\ \mathit{severalhours}(T,D). \qquad [2:\Omega_1] \qquad (3.33) \end{aligned}$$

  Note that the layer is lower than in the Δ cases. Ω_1 describes the weight within the organisational constraints.

- "As few holes as possible in the teachers' schedules", this is to be read as a per-day constraint and is represented very much like in the case where we wanted to forbid holes in the students' timetables. The difference is that this constraint is weak, and that teachers instead of classes are considered.

$$\begin{aligned} \mathit{planned2}(T,P) \leftarrow\ & a(C,T,P,K). \qquad (3.34) \\ \mathit{hole\_on} \leftarrow\ & \mathit{period\_day}(P1,D), \mathit{period\_day}(P2,D), \\ & \mathit{period\_day}(P3,D), P1 < P2, P2 < P3, \\ & a(C1,T,P1,K1), \texttt{not}\ \mathit{planned}_1(T,P2), \\ & a(C3,T,P3,K3). \qquad (3.35) \\ \Leftarrow\ & \mathit{hole\_on}(T,D) \qquad [2:\Omega_2] \qquad (3.36) \end{aligned}$$

  $Omega_2$ is again a weight relative to the other organisational constraints.

- "Concentrate holes on one day preferably (for parent-teacher meeting hours)" means that it is bad if different teachers have holes on different days. We reuse the predicate defined in the last point and thereby get a simple formulation:

$$\Leftarrow \mathit{hole\_on}(T1,D1), \mathit{hole\_on}(T2,D2), T1 \neq T2, D1 \neq D2. \qquad [2:\Omega_3] \qquad (3.37)$$

  Ω_3 is the penalty associated to these constraints.

- "Do not assign all available teachers in one period" means that it is bad if no teacher is free in a period. First, we need a predicate which describes the teachers:

$$t(J) \leftarrow r(C,T,P,K). \qquad (3.38)$$

Next, we need a predicate to determine during what period which teachers are teaching.

$$teaching(T, P) \leftarrow a(C, T, P, K). \tag{3.39}$$

There is a spare teacher at some period if a teacher exists who is not teaching and who is not unavailable at this period.

$$reserve(P) \leftarrow t(T), p(P), \texttt{not}\ teaching(T, P), \texttt{not}\ unavailable(T, P). \tag{3.40}$$

If there is a period in which there is no spare teacher, this costs Ω_4 within the organisational constraints.

$$\Leftarrow p(P), \texttt{not}\ reserve(P). \qquad [2 : \Omega_4] \tag{3.41}$$

We will now define the lowest level constraints – those of the teachers' preferences.

- "Some teachers do not like teaching in the morning." For those, one should define several weak constraints for early hours p and teacher t, probably with greater weight for earlier hours.

$$\Leftarrow a(I, t, p, K). \qquad [1 : \Pi_1^t] \tag{3.42}$$

  where Π_1^t is the weight relative to a particular teacher.

- "Some teachers want to teach in the morning rather than in the afternoon." This is treated in the same way as with the type of constraint, just with different weights.

- "A particular day off." This is easy. Lectures which are scheduled in periods on the particular day d for teacher t, get the weight Π_2^t.

$$\Leftarrow a(I, t, P, K), period\_day(P, d). \qquad [1 : \Pi_2^t] \tag{3.43}$$

- The teacher ranking can be enforced by choosing higher weights for higher-ranked teachers. After all, the weights in this class of constraints should be teacher dependent, because some teacher might deposit more preferences than others, and those should then be assigned lower weights so that they add up to a sum which corresponds to the ranking of the teacher (lower rank – lower total weight). In other words, the weights should be normalised and after that step the ranking should be taken into account.

3.3 Graph Problems

3.3.1 Graph-theoretic Preliminaries

Let us first recall some basics from graph theory, and thereby present our terminology:

Definition 3.3.1 (Graphs)
An undirected graph G_u *is a pair of sets* (V, E)*, where* V *is an arbitrary set and* E *is a set of unordered pairs* $\{v_1, v_2\}$*, where* $v_1, v_2 \in V$*. The elements of* V *are referred to as nodes or vertices, those of* E *as edges. Note that* $\{v_1, v_2\} = \{v_2, v_1\}$ *holds.*

A directed graph G_d *is a pair of sets* (V, A)*, where* V *is an arbitrary set and* A *is a set of ordered pairs* (v_1, v_2)*, where* $v_1, v_2 \in V$*. The elements of* V *are again referred to as nodes or vertices, those of* A *as arcs. Note that* $(v_1, v_2) \neq (v_2, v_1)$ *holds.*

The directed graph $\overrightarrow{G_u}$ *of an undirected graph* $G_u = (V, E)$ *is defined as* $\overrightarrow{G_u} = (V, \overrightarrow{E})$*, where* $\overrightarrow{E} = \{(v_1, v_2), (v_2, v_1) \mid \{v_1, v_2\} \in E\}$.

The undirected graph $\overline{G_d}$ *of a directed graph* $G_d = (V, A)$ *is defined as* $\overline{G_d} = (V, \overline{A})$*, where* $\overline{A} = \{\{v_1, v_2\} \mid (v_1, v_2) \in A\}$*. Note that with this transformation information is lost, since several differing directed graphs are mapped onto the same undirected graph.*

A weighted directed (resp. undirected) graph is a directed (resp. undirected) graph, which has some cost $c_{i,j}$ *associated with each edge* (i, j) *(resp.* $\{i, j\}$*).* ■

We will only consider graphs with a finite set of nodes. This implies that the set of arcs (resp. edges) of these graphs is of finite size as well.

Definition 3.3.2 (Paths and Cycles, Chains and Circles)
A path *in a directed graph* $G_d = (V, A)$ *is a sequence of one or more arcs of* A $(v_1^1, v_1^2), \ldots, (v_n^1, v_n^2)(n \in \mathbb{N})$*, where* $\forall 1 \leq i < n : v_i^2 = v_{i+1}^1$.

A cycle *is a path* $(v_1^1, v_1^2), \ldots, (v_n^1, v_n^2)$*, where* $v_1^1 = v_n^2$.

A chain *in an undirected graph* $G_u = (V, E)$ *is the undirected counterpart to a path, and thus a sequence of one or more edges out of* E $\{v_1^1, v_1^2\}, \ldots, \{v_n^1, v_n^2\}$*, where* $\forall 1 \leq i < n : v_i^2 = v_{i+1}^1$*. Note that chains are reversible.*

A circle *is the undirected counterpart of a cycle, and thus a chain* $\{v_1^1, v_1^2\}$, $\ldots$, $\{v_n^1, v_n^2\}$*, where* $v_1^1 = v_n^2$.

■

Definition 3.3.3 (Connected and Weakly Connected Graphs)
An undirected graph $G_u = (V, E)$ *is said to be* connected*, if for all distinct vertices* $v_1, v_2 \in V$ *a chain* $\{v_1, v_1^2\}, \ldots, \{v_n^1, v_2\}$ *exists.*

A directed graph $G_d = (V, A)$ *is called* weakly connected*, if* $\overline{G_d}$ *is connected.*

■

Definition 3.3.4 (Partial Graphs and Subgraphs)
A directed (resp. undirected) graph $G' = (V, L')$ *is a* partial graph *of a directed (resp. undirected) graph* $G = (V, L)$ *if* $L' \subseteq L$ *holds.*

A directed (resp. undirected) graph $G' = (V', L')$ *is a* subgraph *of a directed (resp. undirected) graph* $G = (V, L)$ *if* $V' \subseteq V$ *and* $L' = \{(v_1, v_2) \mid (v_1, v_2) \in L \wedge v_1, v_2 \in V'\}$ *(resp.* $L' = \{\{v_1, v_2\} \mid \{v_1, v_2\} \in L \wedge v_1, v_2 \in V'\}$ *hold.* ■

Definition 3.3.5 (Trees)
An undirected graph G_u is called a tree, *if it is connected and no circle can be constructed with its edges.*

A directed graph $G_d = (V, A)$ is called a tree, *if $|\{v \mid (w, v) \in A\}| = |V| - 1$ and $\nexists v, w_1, w_2 \in V : (w_1, v), (w_2, v) \in A \wedge w_1 \neq w_2$. This condition implies that each node except for one, which is called root of the tree, is the final node of exactly one arc. It also implies that $|A| = |V| - 1$.*

■

3.3.2 Classical Minimum Spanning Tree

Graph Theoretic Formulation

In the literature this problem also appears as "Shortest Spanning Tree". Usually (for instance in [Chr75, Sed88, Pap94]) it is formulated over undirected connected graphs.

A related problem dealing with general weakly connected directed graphs is dealt with in Section 3.3.3.

Definition 3.3.6 (Spanning Tree)
A spanning tree of an undirected connected graph $G_u = (V, E)$ (cf. Definition 3.3.1, Definition 3.3.3) is any partial graph $G'_u = (V, E')$ of G_u, which is a tree.

Let us denote the set of all spanning trees of an undirected connected graph G_u by $ST(G_u)$.

■

Definition 3.3.7 (Minimum Spanning Tree)
A minimal spanning tree of a connected weighted undirected graph $G_u = (V, E)$ is called G_u^{MST} and defined as follows:

$$G_u^{MST} = \min_{(V,E') \in ST(G_u)} \sum_{\{i,j\} \in E'} c_{i,j}$$

That is, the spanning tree of G_u which is minimal w.r.t. the sum of the costs of its edges.

■

Example 3.3.1
Given the weighted undirected graph G depicted in Figure 3.10 on the following page, all its spanning trees and the associated weights are shown in Figure 3.11 on the next page. One can immediately see that the spanning tree in Figure 3.11(c) on the following page is the one with minimal total weight.

♦

Applications

Applications of instances of this problem are manifold. Some examples in the literature are:

- Terminals in an electric network should be connected together such that the total length of wire to be used is minimised in order to reduce stray effects. [Chr75]

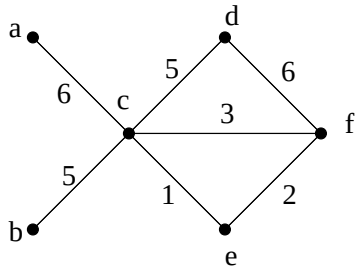
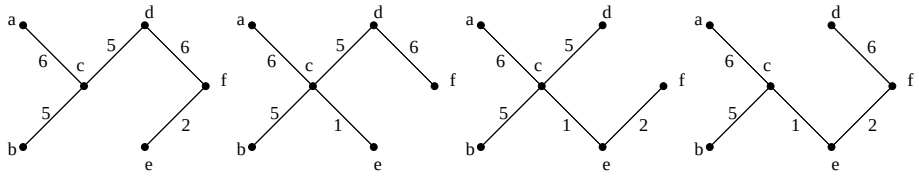
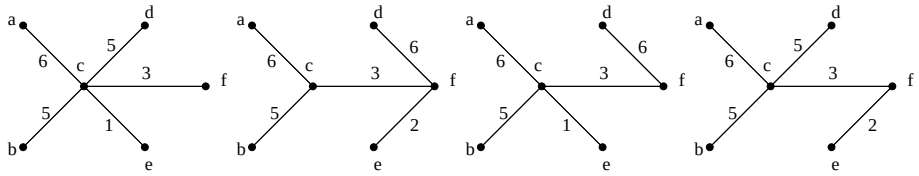


Figure 3.10: Example graph G



(a) Weight: 24 (b) Weight: 23 (c) Weight: 19 (d) Weight: 20



(e) Weight: 20 (f) Weight: 22 (g) Weight: 21 (h) Weight: 21

Figure 3.11: All spanning trees of G in Figure 3.10

- A pipeline network should be constructed such that out-of-town junctions are not allowed and the amount of pipe material is minimised. [Chr75]
- In a network of agencies, different information channels exist, each of which has some associated probability of interception. Find the way in which a message can be spread to all agencies with minimal probability of being heard by others. [Chr75]
- In the field of land use planning (or regional science) one has to consider distribution channels between distributors and consuments. Usually, at a place where there are distributors, there are also consuments and vice versa. Such distribution networks may be represented by weighted undirected graphs, in which the edges represent the distribution channels, where their weights can be viewed as overheads. When planning new distribution channels (usually railways, roads etc.), all possible variants of these channels are packed into the graph, and the solution for the MINIMUM SPANNING TREE problem over this graph gives the plan with minimum distribution overhead [Bök82].

  It should be noted that in the scope of this problem multiple political preferences have to be honored (e.g. a railway line should possibly not be built in a recreational area). Our framework would provide sufficient means to incorporate these considerations into the problem formulation.

Representation in DATALOG$^{not,\vee,w}$ – A Guess and Check Solution

The first representation of this problem in our language is straightforward from the definition of a Minimum Spanning Tree.

But before, we consider the representation of the graph: We assume that we only want to treat undirected graphs which do not have isolated nodes. It thus suffices to store $edge(X, Y)$ if there's an edge $\{X, Y\}$. Therefore an edge $\{a, b\}$ is represented by either $edge(a, b)$ or $edge(b, a)$, but not both.

Since Spanning Trees are partial graphs, it is straightforward to consider all partial graphs. An edge is either contained in a subgraph or it is not:

$$st(X, Y) \vee nst(X, Y) \leftarrow edge(X, Y). \tag{3.44}$$

Having only rule 3.44 in the program, there is one model corresponding to each possible partial graph.

Now we have to check which of these partial graphs are actually Spanning Trees. The partial graph may not contain any circles. A graph contains a circle iff there is one node from which a chain leads back to itself, with each edge in the path being unique.

For simplicity, we will not represent chains, but paths in $\overrightarrow{E}$, if the partial graph is $P = (V, E)$.

$$dst(X, Y) \leftarrow st(X, Y). \tag{3.45}$$
$$dst(Y, X) \leftarrow st(X, Y). \tag{3.46}$$

Later on we will need a predicate which tests the equality of two edges, which we will define now.

$$node(X) \leftarrow edge(X, Y). \tag{3.47}$$
$$node(Y) \leftarrow edge(X, Y). \tag{3.48}$$

$$eq\_edge(X, Y, X, Y) \leftarrow node(X), node(Y). \tag{3.49}$$

Since we have paths representing chains, we do not want to have an arc (X, Y) and its counterpart (Y, X) in the path, if the corresponding chain should be unique (clearly (X, Y) and (Y, X) represent the same edge). But since the criterion we want to check is whether a chain starting and ending in the same node exists, it suffices for us to require that the retrograde arc (Y, X) of the initial arc (X, Y) representing the first edge of the chain must not be in the path. We will call (Y, X) the "forbidden" arc.

Therefore, when defining paths which could be cycles and thus circles in the original potential spanning tree we have to keep track of the forbidden arc in a path, so that it is not chosen for such a path, since such a cycle does not correspond to a proper circle in the original graph.

$$stpath\_f(X, Y, Y, X) \leftarrow dst(X, Y). \tag{3.50}$$

Now, if we've got a path from X to $Y1$ and an arc between $Y1$ and Y, and this is not the forbidden arc, we also have a path from X to Y which is a cycle iff it is a circle in the original graph.

$$\begin{aligned} stpath\_f(X, Y, Xf, Yf) \leftarrow\ & stpath\_f(X, Y1, Xf, Yf), dst(Y1, Y), \\ & \texttt{not}\ eq\_edge(Y1, Y, Xf, Yf). \end{aligned} \tag{3.51}$$

Not that we can not write $Y1 \neq Xf, Y \neq Yf$ instead of $\texttt{not}\ eq\_edge(Y1, Y, Xf, Yf)$, since it would not be possible to construct any path starting in X, which goes beyond Xf.

Now, our undirected subgraph is free of circles iff there is no path without the forbidden arc from one node to itself. We state this using a strong constraint (since every spanning tree, and the minimum spanning tree in particular, must satisfy this):

$$\leftarrow stpath\_f(X, X, Xf, Yf). \tag{3.52}$$

So the possible models for the program so far are all partial graphs not containing a circle.

The remaining criterion is that the subgraph has to be connected to be a spanning tree. A graph is connected, if for any pair of nodes there exists a chain between them. We have almost defined this before as $stpath\_f/4$; but we did not want the nodes to have a path to themselves. concerning reachability, a node should always be reachable from itself, so we just need to extend our path definition to become reflexive:

$$st\_reach(X, X) \leftarrow node(X). \tag{3.53}$$
$$st\_reach(X, Y) \leftarrow stpath\_f(X, Y, Xf, Yf). \tag{3.54}$$

Now we formulate the constraint: Given two nodes, it shouldn't be the case that either one is not reachable from the other.

$$\leftarrow node(X), node(Y), \texttt{not } st\_reach(X, Y). \tag{3.55}$$

So far the stable models of the program correspond to the spanning trees. Now we want this spanning tree to have minimum added cost. Therefore we create a weak constraint for each edge, assigning the weight of the original edge to the respective constraint. All of these weak constraints are in the same default layer.

$$\Leftarrow st(X, Y).[: c_{XY}]\forall edge(X, Y) \tag{3.56}$$

The resulting program – let us call it $\mathcal{P}_{mst\_c}$ is shown in Figure 3.3.2 on the following page. The models of this program combined with a graph representation correspond to the minimum spanning trees, which are formed by the *st* atoms in the respective model.

Note that the program is modular in the sense of [EGM97]: We can differentiate between the guess part π_1 (3.44), the circle-freeness part π_2 (3.45, 3.46, 3.50, 3.51, 3.47, 3.48, 3.49, 3.52), the connectedness part π_3 (3.53, 3.54, 3.55), and the minimality part π_4 (3.56). In the terminology of [EGM97]: $\pi_3 \triangleright \pi_2$, $\pi_2 \triangleright \pi_1$, and $\pi_4 \triangleright \pi_1$, where $\triangleright$ can sloppily be described as "uses".

Representation in DATALOG$^{not,\vee,w-}$ A Solution Using Unstratified Negation

This representation tries to model the Prim algorithm as good as possible, using our particular implementation of computing stable models with weak constraints, explained in Chapter 4. Of course, doing this requires substantial knowledge of the algorithm used to compute the models. Additionally, since the representation is declarative, we have no method to express imperative statements like "choose an arbitrary node".

For this reason we assume that our database includes a fact $root(n)$. defined for exactly one arbitrary node n of the graph. This models the assignment "Choose an arbitrary node" in the Prim algorithm. Any node can be selected, because in this case of undirected graphs there is no determined root node. We will transform the undirected graph (V, E) (given by $edge(X, Y)$. facts, one for each $\{X, Y\} \in E$; V is defined implicitly since we only consider connected graphs) into a directed one using two rules:

$$arc(X, Y) \leftarrow edge(X, Y). \tag{3.57}$$
$$arc(Y, X) \leftarrow edge(X, Y). \tag{3.58}$$

$$st(X,Y) \vee nst(X,Y) \leftarrow edge(X,Y). \tag{3.44}$$
$$stpath\_f(X,Y,Y,X) \leftarrow dst(X,Y). \tag{3.50}$$
$$stpath\_f(X,Y,Xf,Yf) \leftarrow stpath\_f(X,Y1,Xf,Yf), dst(Y1,Y), \texttt{not } eq\_edge(Y1,Y,Xf,Yf). \tag{3.51}$$
$$\leftarrow stpath\_f(X,X,Xf,Yf). \tag{3.52}$$
$$st\_reach(X,X) \leftarrow stpath\_f(X,Y,Xf,Yf). \tag{3.53}$$
$$st\_reach(X,Y) \leftarrow stpath\_f(X,Y,Xf,Yf). \tag{3.54}$$
$$\leftarrow node(X), node(Y), \texttt{not } st\_reach(X,Y). \tag{3.55}$$
$$\Leftarrow st(X,Y).[: c_{XY}] \forall edge(X,Y) \tag{3.56}$$

$$dst(X,Y) \leftarrow st(X,Y). \tag{3.45}$$
$$dst(Y,X) \leftarrow st(X,Y). \tag{3.46}$$
$$node(X) \leftarrow edge(X,Y). \tag{3.47}$$
$$node(Y) \leftarrow edge(X,Y). \tag{3.48}$$
$$eq\_edge(X,Y,X,Y) \leftarrow node(X), node(Y). \tag{3.49}$$

Figure 3.12: A constraint solution for Minimum Spanning Tree

Note that for any model M, $arc(X,Y) \in M \Leftrightarrow (X,Y) \in \overrightarrow{E}$ holds.

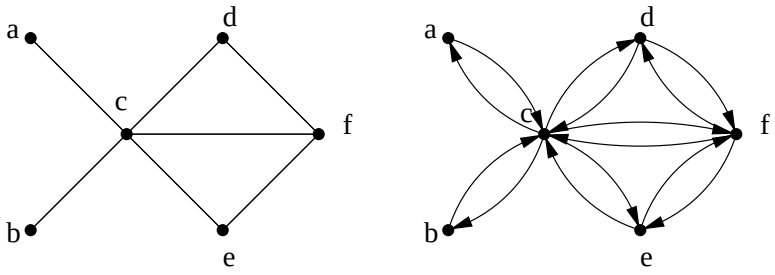
Next we determine the partial subgraphs, which are directed trees, the root of which is n (Call them directed spanning trees with root in n). These correspond one-to-one to the spanning trees of the undirected graph. In particular, the minimal spanning tree corresponds to the minimal directed spanning tree of the directed version of the graph.

Therefore, we view the given undirected graph as a directed one (for each undirected edge there are two arcs). The strategy is to start with a one-node tree and add an arc if its initial vertex is reached by the tree and its final vertex is not yet reached . Of course, this description is procedural, but we can reformulate it avoiding the "not yet" and "add". To achieve this, let us first have a look at the following example:

Example 3.3.2
Consider the undirected graph depicted in Figure 3.13(a) on the next page, call it $G = (V,E)$, *where* $V = \{a,b,c,d,e,f\}$, $E = \{\{a,c\}, \{b,c\}, \{c,d\}, \{c,f\}, \{c,e\}, \{d,f\}, \{e,f\}\}$, *Figure 3.13(b) on the following page shows the directed version* $\overrightarrow{G} = (V, \overrightarrow{E})$, *where* $\overrightarrow{E} = \{(a,c), (c,a), (b,c), (c,b), (c,d), (d,c), (c,f), (f,c), (c,e), (e,c), (d,f), (f,d), (e,f), (f,e)\}$.

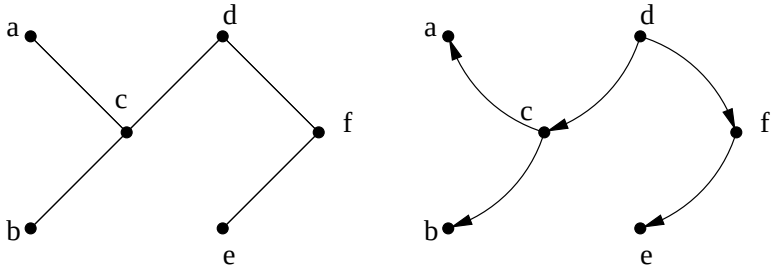
Now, consider a particular spanning tree of G *(Figure 3.14(a) on the following page), call it* $T = (V,E_T)$, *where* $E_T = \{\{a,c\},\{b,c\},\{c,d\},\{d,f\},\{e,f\}\}$. *If we choose* d *to be the root node, we get the directed tree* $\overrightarrow{T_d} =$, *as shown in Figure 3.14(b) on the next page*

If we look at the arcs of $\overrightarrow{G}$ *which are not in* $\overrightarrow{T_d}$ *(the dotted arcs in Figure 3.15 on the following page), each node but the root node (*d *in the case of Figure 3.15 on the next page), are reached by exactly one arc of* $\overrightarrow{T_d}$. *Indeed, this follows*



(a) G (b) $\overrightarrow{G}$

Figure 3.13: Undirected and Directed Example Graph G



(a) T (b) $\overrightarrow{T_d}$ (rooted in d)

Figure 3.14: A Spanning Tree of G (as in Figure 3.13)

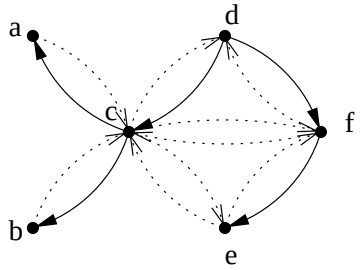


Figure 3.15: $\overrightarrow{T_d}$, arcs in $\overrightarrow{G}$, but not in $\overrightarrow{T_d}$ are dotted

directly from Definition 3.3.5. ♦

Having this example in mind, we may declare every arc in a graph *outside*, if its final node is the root node or a final node of some different arc in the directed spanning tree.

Now, one can describe the directed spanning tree declaratively: An arc is part of the directed spanning tree, if its starting node is in the tree, and if the arc is not outside.

$$st(X,Y) \leftarrow reached(X), \texttt{not}\ outside(Y,X), arc(X,Y). \tag{3.59}$$

A node is in the tree (or *reached*), if it is the root node or if it is the final node of an arc in the directed spanning tree.

$$reached(X) \leftarrow root(X). \tag{3.60}$$
$$reached(X) \leftarrow st(Y,X). \tag{3.61}$$

A formal declaration of the *outside* property is still missing. An arc is outside if its final node is the root node or if its final node is the final node of some arc of the spanning tree and the considered arc is not itself part of the spanning tree.

$$outside(Y,X) \leftarrow root(Y), arc(X,Y). \tag{3.62}$$
$$outside(Y,X) \leftarrow st(Z,Y), arc(X,Y), Z \neq X. \tag{3.63}$$

Up to now we have described a program which computes the directed spanning trees of a graph. Our final goal is now to compute the minimum directed spanning tree. To this end, add the following weak constraints:

$$\Leftarrow st(X,Y).[: c_{XY}] \forall edge(X,Y) \tag{3.64}$$
$$\Leftarrow st(Y,X).[: c_{XY}] \forall edge(X,Y) \tag{3.65}$$

The resulting program – call it $\mathcal{P}_{st\_u}$ – is shown in Figure 3.16 on the following page.

3.3.3 Minimum Spanning Tree of a Directed Graph

Although similar to the Classical Minimum Spanning Tree Problem described in Section 3.3.2, some peculiarities arise when considering Minimum Spanning Trees of directed graphs. It turns out that some assumptions about the graph and the spanning trees have to be dropped. But first, let us define the problem formally:

Graph Theoretic Formulation

Definition 3.3.8 (Spanning Tree of a Directed Graph)
A spanning tree of a directed connected graph $G_d = (V, A)$ (cf. Definition 3.3.1, Definition 3.3.3) is any partial graph $G'_d = (V, A')$ of G_d which is a tree.

$$st(X,Y) \leftarrow reached(X), \texttt{not}\ outside(Y,X), arc(X,Y). \quad (3.59)$$
$$reached(X) \leftarrow root(X). \quad (3.60)$$
$$reached(X) \leftarrow st(Y,X). \quad (3.61)$$
$$outside(Y,X) \leftarrow root(Y), arc(X,Y). \quad (3.62)$$
$$outside(Y,X) \leftarrow st(Z,Y), arc(X,Y), Z \neq X. \quad (3.62)$$
$$\Leftarrow st(X,Y).[: c_{XY}]\forall edge(X,Y) \quad (3.64)$$
$$\Leftarrow st(Y,X).[: c_{XY}]\forall edge(X,Y) \quad (3.65)$$

$$arc(X,Y) \leftarrow edge(X,Y). \quad (3.57)$$
$$arc(Y,X) \leftarrow edge(X,Y). \quad (3.58)$$

Figure 3.16: A solution using unstratified negation for Minimum Spanning Tree

Let us denote the set of all spanning trees of an undirected connected graph G_d as $ST(G_d)$.

■

Definition 3.3.9 (Minimum Spanning Tree of a Directed Graph)
A minimal spanning tree of a connected weighted directed graph $G_d = (V, A)$, called G_d^{MST}, is defined as follows:

$$G_d^{MST} = \min_{(V,A') \in ST(G_d)} \sum_{(i,j) \in A'} c_{i,j}$$

That is, the spanning tree of G_d which is minimal w.r.t. the sum of the cost of its arcs.

■

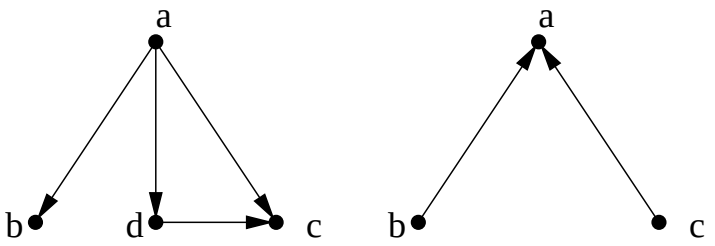
Definition 3.3.8 and Definition 3.3.9 are very similar to Definition 3.3.6 and Definition 3.3.7, respectively.

But there is a fundamental difference if we try to describe those graphs which have spanning trees, let alone minimum ones. In the undirected case we could say that every connected graph has at least one spanning tree, and every disconnected graph has none.

In the case of directed graphs we have several flavours of connectedness – *strongly connected*, iff there is at least one path from every node to every other node, *unilateral*, iff there is at least one path between two distinct nodes, *weakly connected*, iff the corresponding undirected graph is connected, or *disconnected*. Clearly, a strongly connected graph is also unilateral, and a unilateral graph is also weakly connected.

It turns out that unilateral graphs always have spanning trees:

Proposition 3.3.1
If a directed graph $G = (V, A)$ is unilateral, it has a directed spanning tree $S = (V, A'), A' \subseteq A$.



(a) Spanning Trees Exist (b) No Spanning Trees

Figure 3.17: Weakly Connected Graphs With and Without Spanning Trees

Proof There is at most one node in V, which is not the final node of an arc. If there were more than one, a path between these two couldn't exist. But that means that a subset of A exists, so that each node in V is the final node of exactly one arc, except for one node, which is not a final node of an arc. The last sentence is the criterion for a tree, so G has spanning trees. △

Weakly connected graphs may or may not have spanning trees, as can be seen in Figure 3.17. Disconnected graphs obviously do not have spanning trees.

Applications

Reconsidering the example of land use planning, we made the assumption that at a place where there are distributors, there are also consuments. If we drop this, we arrive at adirected graph, since distribution channels only go from distributors to consuments. Still we want to minimise the distribution overhead, so we are again interested in the minimum directed spanning tree.

Representation in DATALOG$^{not,\vee,w}$

A substantial difference to the case of undirected graphs is, that we may not choose an arbitrary node as root node. For instance every spanning tree of the graph shown in Figure 3.17(a) must be rooted in a.

We can extend our unstratified program presented in Figure 3.16 on the page before to handle this difference. The graph representation here is by the predicate $arc/2$, so the *node* predicate has to be slightly altered:

$$node(X) \leftarrow arc(X, Y). \tag{3.66}$$
$$node(Y) \leftarrow arc(X, Y). \tag{3.67}$$

We must take precautions that the given graph might not have any spanning tree. In the program in Figure 3.16 on the preceding page, we assumed that the graph is connected, do a spanning tree had to exist.

So we have to add a constraint which states that a node in the graph, which is not reached should not exist:

$$\leftarrow node(X), \texttt{not } reached(X). \tag{3.68}$$

Further, we may not assume any node to be the root node, so one has to be guessed:

$$root(X) \vee nroot(X) \leftarrow node(X). \tag{3.69}$$

We want to have at least one root node:

$$have\_root \leftarrow root(X). \tag{3.70}$$
$$\leftarrow \texttt{not } have\_root. \tag{3.71}$$

But we do not want more than one:

$$\leftarrow root(X), root(Y), X \neq Y. \tag{3.72}$$

Last, the weak constraints need to be updated for the arc representation:

$$\Leftarrow st(X,Y).[: c_{XY}] \forall arc(X,Y) \tag{3.73}$$

The complete program with the remaining rules from the program in Figure 3.16 on page 70 is depicted in Figure 3.18 on the following page.

3.3.4 Minimum Steiner Trees

Graph Theoretic Formulation

This is similar to the Minimum Spanning Tree problem described in Section 3.3.2. The difference is that the task is to find a minimal spanning tree of a subgraph of the given graph, which includes all of a specified set of nodes, which are called Steiner nodes. So the Minimum Spanning Tree problem is a particular instance of this, where the set of nodes, which should be included in the Steiner Tree coincides with the set of nodes of the original graph.

Definition 3.3.10 (Steiner Tree of an Undirected Graph)
Given an undirected connected graph $G_u = (V, E)$ and a set of Steiner nodes $S \subseteq V$, a Steiner tree of G_u is any subgraph $G'_u = (V', E')$ of G_u which is a tree and for which $S \subseteq V'$ holds.

Let us denote the set of all Steiner trees of an undirected connected graph G_u as $STT(G_u)$.

■

Definition 3.3.11 (Minimum Steiner Tree of an Undirected Graph)
A minimum Steiner tree of a connected weighted undirected graph $G_u = (V, E)$, called G_d^{MSTT}, is defined as follows:

$$G_u^{MSTT} = \min_{(V,E') \in STT(G_u)} \sum_{(i,j) \in E'} c_{i,j}$$

■

$$\leftarrow node(X), \texttt{not}\ reached(X). \tag{3.68}$$
$$\leftarrow \texttt{not}\ have\_root. \tag{3.71}$$
$$st(X,Y) \leftarrow reached(X), \texttt{not}\ outside(Y,X), arc(X,Y). \tag{3.59}$$
$$reached(X) \leftarrow root(X). \tag{3.60}$$
$$reached(X) \leftarrow st(Y,X). \tag{3.61}$$
$$outside(Y,X) \leftarrow root(Y), arc(X,Y). \tag{3.62}$$
$$outside(Y,X) \leftarrow st(Z,Y), arc(X,Y), Z \neq X. \tag{3.62}$$
$$\Leftarrow st(X,Y).[:c_{XY}]\forall arc(X,Y) \tag{3.73}$$

$$root(X) \vee nroot(X) \leftarrow node(X). \tag{3.69}$$
$$have\_root \leftarrow root(X). \tag{3.70}$$
$$\leftarrow root(X), root(Y), X \neq Y. \tag{3.72}$$

$$node(X) \leftarrow arc(X,Y). \tag{3.66}$$
$$node(Y) \leftarrow arc(X,Y). \tag{3.67}$$

Figure 3.18: A solution for Directed Minimum Spanning Tree

Actually, this is the graph theoretical version of the so-called Euclidean Steiner Problem, which occurs in geometry. For more details, we refer to [Chr75] and [Pap94].

Applications

There are many applications which have to solve this problem: For example the application of connecting cities with pipelines described above did not allow out-of-town junctions. If we assume that the company, which wants to build this pipeline networks, has identified several places where such out-of-town junctions might be built, the problem becomes the following: The nodes of the graph represent cities and possible junctions, but only the cities are in the set of Steiner nodes, and the task is to find a minimum Steiner tree.

Representation in Datalog$^{not,\vee,w}$

This problem is easy to represent reusing the program shown in Figure 3.3.2 on page 67.

The only thing which needs altering is constraint 3.55, since not all nodes have to be reached, but just the Steiner nodes. The requirement for circle-freeness still holds.

If we assume that these nodes are represented with a predicate *steiner*, the constraint becomes:

$$\leftarrow steiner(X), steiner(Y), notst\_reach(X,Y). \tag{3.74}$$

$$st(X,Y) \vee nst(X,Y) \leftarrow edge(X,Y). \tag{3.44}$$
$$stpath\_f(X,Y,Y,X) \leftarrow dst(X,Y). \tag{3.50}$$
$$stpath\_f(X,Y,Xf,Yf) \leftarrow stpath\_f(X,Y1,Xf,Yf), dst(Y1,Y), \texttt{not } eq\_edge(Y1,Y,Xf,Yf). \tag{3.51}$$
$$\leftarrow stpath_f(X,X,Xf,Yf). \tag{3.52}$$
$$st\_reach(X,X) \leftarrow stpath\_f(X,Y,Xf,Yf). \tag{3.53}$$
$$st\_reach(X,Y) \leftarrow stpath\_f(X,Y,Xf,Yf). \tag{3.54}$$
$$\leftarrow steiner(X), steiner(Y), \texttt{not } st\_reach(X,Y). \tag{3.74}$$
$$\Leftarrow st(X,Y).[: c_{XY}]\forall edge(X,Y) \tag{3.56}$$

$$dst(X,Y) \leftarrow st(X,Y). \tag{3.45}$$
$$dst(Y,X) \leftarrow st(X,Y). \tag{3.46}$$
$$node(X) \leftarrow edge(X,Y). \tag{3.47}$$
$$node(Y) \leftarrow edge(X,Y). \tag{3.48}$$
$$eq\_edge(X,Y,X,Y) \leftarrow node(X), node(Y). \tag{3.49}$$

Figure 3.19: A constraint solution for Minimum Steiner Tree

The resulting program is depicted in Figure 3.19.

Chapter 4

Algorithms

The algorithms which we are going to present in this chapter rely on the one presented for DATALOG$^{not,\vee}$ in [LRS97]. We will give some of the definitions, propositions and theorems taken from there, but refer to [LRS97] for proofs and the full background.

Note that by defining an algorithm which computes the preferred stable models of DATALOG$^{not,\vee,w}$ we also have a means to compute the consistent preferred answer sets, as shown in Section 2.6.6.

4.1 Unfoundedness

As we will see later on, *unfounded sets* and the related *unfounded-free interpretations* are fundamental for the algorithms. In this section, we will formally define these and some related notions. In Section 4.2 we will then give an efficient algorithm for checking unfounded-freeness.

First we define the notion of *unfounded sets*, which is also crucial in the definition of the well-founded semantics. Intuitively, a set of ground atoms is an unfounded set w.r.t. a program and a possibly partial interpretation if there is reason to assume that these atoms are false w.r.t. the interpretation.

Definition 4.1.1 (Unfounded sets [LRS97])
Given a program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$*, its grounded abstract version* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*, and a (possibly partial) interpretation* $\mathcal{I}$*, the set of unfounded sets w.r.t.* $\mathcal{P}$ *(and* $\mathcal{P}'$*) and* $\mathcal{I}$ *is* $U_{\mathcal{P},\mathcal{I}}$*, defined as*

$$U_{\mathcal{P},\mathcal{I}} = \{U \mid U \subseteq HB(\mathcal{P}), \forall (H,B) \in \mathcal{P} : U \cap H \neq \emptyset \rightarrow$$
$$(\ B \cap \texttt{not}\ (\mathcal{I}) \neq \emptyset \tag{4.1}$$
$$\vee\ B^+ \cap U \neq \emptyset \tag{4.2}$$
$$\vee\ (H - U) \cap \mathcal{I} \neq \emptyset)\} \tag{4.3}$$

■

So an unfounded set must satisfy at least one of the conditions (4.1, 4.2, 4.3) for every rule in which one of the members of the unfounded set occurs in the head.

4.1 means that the body is false w.r.t. $\mathcal{I}$, 4.2 states that some positive literal in the body belongs to the unfounded set $U_{\mathcal{P},\mathcal{I}}$, finally 4.3 holds if an atom in the head of the rule, which is not in $U_{\mathcal{P},\mathcal{I}}$, is already true w.r.t. $\mathcal{I}$. Note that constraints cannot have any significance in this definition, since the precondition can never be true.

Example 4.1.1
Let $\mathcal{P}_1$ be a program modelling some incomplete knowledge about creatures and their genders:

$$\begin{aligned}\mathcal{P}_1 = \{\ & \texttt{woman} \vee \texttt{man} \leftarrow \texttt{creature}, \texttt{not animal}.,\\ & \texttt{stallion} \vee \texttt{mare} \leftarrow \texttt{creature}, \texttt{animal}, \texttt{horse}.,\\ & \texttt{creature}.\}\end{aligned}$$

Given the interpretation $\mathcal{I}_1 = \{\texttt{creature}, \texttt{not horse}\}$, then

$$\begin{aligned}U_{\mathcal{P}_1,\mathcal{I}_1} = \{\ & \{\texttt{stallion}\},\\ & \{\texttt{mare}\},\\ & \{\texttt{stallion}, \texttt{mare}\},\\ & \emptyset\}\end{aligned}$$

All the nontrivial unfounded sets exist because of 4.1, as `not horse` *falsifies the body of the second rule.*
Given another interpretation $\mathcal{I}_2 = \{\texttt{creature}, \texttt{not animal}, \texttt{man}, \texttt{woman}\}$ to $\mathcal{P}_1$, then

$$\begin{aligned}U_{\mathcal{P}_1,\mathcal{I}_2} = U_{\mathcal{P}_1,\mathcal{I}_1} \cup \{\ & \{\texttt{stallion}, \texttt{woman}\},\\ & \{\texttt{mare}, \texttt{woman}\},\\ & \{\texttt{stallion}, \texttt{mare}, \texttt{woman}\},\\ & \{\texttt{stallion}, \texttt{man}\}\\ & \{\texttt{mare}, \texttt{man}\},\\ & \{\texttt{stallion}, \texttt{mare}, \texttt{man}\}\}\end{aligned}$$

Now, in addition to the unfounded sets w.r.t. $\mathcal{I}_1$, any extension of the sets contained in $U_{\mathcal{P}_1,\mathcal{I}_1}$ by either `man` *or* `woman`*, but not both, is an unfounded set because of 4.3.* ♦

We will be interested in interpretations not containing any atoms which are also contained in some unfounded set with respect to the considered interpretation and some program. These interpretations are called *unfounded-free* w.r.t. a program. After all, an interpretation which contains atoms for which the interpretation itself gives reason to be false is not really desirable, since it contradicts itself in some way, as indicated by Example 4.1.1.

Definition 4.1.2 (Unfounded-free Interpretations [LRS97])
The set of unfounded-free interpretations w.r.t. a program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *and its grounded abstract counterpart* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$ *is comprised of those (possibly partial) interpretations* $\mathcal{I}$ *which are disjoint with all unfounded sets w.r.t.* $\mathcal{P}$ *and* $\mathcal{I}$*.*

$$UF_{\mathcal{P}} = \{\mathcal{I} \mid \mathcal{I} \subseteq HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}), \mathcal{I}^+ \cap \texttt{atoms}(\mathcal{I}^-) = \emptyset, \\ \forall U \in U_{\mathcal{P},\mathcal{I}} : \mathcal{I} \cap U = \emptyset\}$$

■

Example 4.1.2
Considering Example 4.1.1, we see that $\mathcal{I}_1 \in UF_{\mathcal{P}_1}$*, but* $\mathcal{I}_2 \notin UF_{\mathcal{P}_1}$*.* ♦

Later on, we will also use the notion of the *greatest unfounded set*. This is the maximal set of atoms w.r.t. set inclusion, for which an interpretation gives reason to assume their falsity. Note that such a set need not exist in the case of disjunctive programs – it is guaranteed to exist in the non-disjunctive case.

Definition 4.1.3 (Greatest Unfounded Set [LRS97])
The greatest unfounded set $GUS_{\mathcal{P}}(\mathcal{I})$ *of a program* $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *and its grounded abstract version* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$ *w.r.t. a (possibly partial) interpretation* $\mathcal{I}$ *is the union of all unfounded sets of* $\mathcal{P}$ *w.r.t.* $\mathcal{I}$ *which is itself an unfounded set.*

$$GUS_{\mathcal{P}}(\mathcal{I}) = \begin{cases} \bigcup_{U \in U_{\mathcal{P},\mathcal{I}}} U & \text{if } \bigcup_{U \in U_{\mathcal{P},\mathcal{I}}} U \in U_{\mathcal{P},\mathcal{I}} \\ \textit{undefined} & \textit{else} \end{cases}$$

We also define the set of interpretations for a given program for which the greatest unfounded set exists, and call it $\mathbf{I}_{\mathcal{P}}$*:*

$$\mathbf{I}_{\mathcal{P}} = \{\mathcal{I} \subseteq (HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}) \mid \texttt{atoms}(\mathcal{I}^+) \cap \texttt{atoms}(\mathcal{I}^-) = \emptyset, \\ GUS_{\mathcal{P}}(\mathcal{I}) \textit{ is defined } \}$$

■

Example 4.1.3
Consider $\mathcal{P}_1$ *from Example 4.1.1.* $GUS_{\mathcal{P}_1}(\mathcal{I}_1) = \{\texttt{stallion}, \texttt{mare}\}$ *whereas* $GUS_{\mathcal{P}_1}(\mathcal{I}_2)$ *does not exist, since* $\{\texttt{stallion}, \texttt{mare}, \texttt{man}, \texttt{woman}\} \notin U_{\mathcal{P}_1,\mathcal{I}_2}$*. Therefore* $\mathcal{I}_1 \in \mathbf{I}_{\mathcal{P}_1}$*, but* $\mathcal{I}_2 \notin \mathbf{I}_{\mathcal{P}_1}$*.* ♦

Note that by Proposition 3.7 in [LRS97] $UF_{\mathcal{P}} \subseteq \mathbf{I}_{\mathcal{P}}$ holds, which means that $\mathcal{I}_1 \in UF_{\mathcal{P}_1}$, $\mathcal{I}_2 \notin UF_{\mathcal{P}_1}$ and $\mathcal{I}_1 \in \mathbf{I}_{\mathcal{P}_1}$, $\mathcal{I}_2 \notin \mathbf{I}_{\mathcal{P}_1}$ in our example is not simply a coincidence.

4.2 Checking Unfounded-freeness

In this section, we will define a function which checks unfounded-freeness. This function relies on the operator $\mathcal{R}_{\mathcal{P},\mathcal{I}}$ and several results which were presented and proved in [LRS97] and we will use this check in our algorithm.

We start by defining the operator $\mathcal{R}_{\mathcal{P},\mathcal{I}}$, which derives from a given set of atoms those which are unfounded.

Definition 4.2.1 ($\mathcal{R}_{\mathcal{P},\mathcal{I}}$ – the unfoundedness operator)
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$ *of some* $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *and a (possibly partial) interpretation* $\mathcal{I}$*, let*

$$\begin{aligned}
&\mathcal{R}_{\mathcal{P},\mathcal{I}} : \mathbb{P}(HB(\mathcal{P})) \to \mathbb{P}(HB(\mathcal{P}))\\
&\mathcal{R}_{\mathcal{P},\mathcal{I}}(X) = \{a \mid\ a \in X,\\
&\qquad\qquad \forall (H,B) \in \mathcal{P} :\\
&\qquad\qquad a \in H \to (B \cap (\texttt{not}\ (\mathcal{I}) \cup X) \neq \emptyset \vee (H - \{a\}) \cap \mathcal{I} \neq \emptyset)\}
\end{aligned}$$

■

By Proposition 6.5 in [LRS97] for any grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, where $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ and any total interpretation $\mathcal{I}$ for $\mathcal{P}$, if the sequence $R_0 = \mathcal{I}^+, \ldots, R_n = \mathcal{R}_{\mathcal{P},\mathcal{I}}(R_{n-1})$ converges to the limit $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}(\mathcal{I}^+) = \emptyset$, then $\mathcal{I} \in UF_{\mathcal{P}}$. Note that constraints again do not alter the behaviour of this operator.

We will now consider a special class of programs called head-cycle-free programs (HCF for short), for which unfounded-freeness checking will be very efficient. First we show what head-cycle freeness is and how to determine whether a program is HCF. To this end we have to define dependency graphs:

Definition 4.2.2 (Dependency Graphs)
Given a program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *and its grounded abstract version* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*, the directed graph* $DG_{\mathcal{P}} = (V, A)$ *is defined as follows:*

$$\begin{aligned}
V &= \{\quad p \mid\ p \in \texttt{predicates}(\mathcal{P})\}\\
A &= \{(p_1, p_2) \mid\ \exists (H,B) \in \mathcal{P} :\\
&\qquad\qquad (\exists b \in B^+ : p_1 = \texttt{predicates}_A(b))\\
&\qquad\qquad \wedge\ (\exists a \in H : p_2 = \texttt{predicates}_A(a))\}
\end{aligned}$$

So for every predicate there is a node, and an arc from one node to another exists if its starting node occurs positively in the body of a rule and its final node occurs in the head of the same rule. ■

Definition 4.2.3 (Collapsed Dependency Graphs)
The collapsed dependency graph $\widehat{DG_{\mathcal{P}}} = (\hat{V}, \hat{A})$ *of a program* $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$*, the grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*, and its dependency graph* $DG_{\mathcal{P}}$ *is defined as*

$$\begin{aligned}
\hat{V} &= \{\, Q \subseteq V \mid\ v_1, v_2 \in Q \Leftrightarrow \textit{a path from } v_1 \textit{ to } v_2 \textit{ and from } v_2 \textit{ to } v_1 \textit{ exists,}\\
&\qquad\qquad \textit{or } v_1 = v_2\}\\
\hat{A} &= \{(v_1, v_2) \mid\ \exists u_1 \in v_1, u_2 \in v_2 : (u_1, u_2) \in A\}
\end{aligned}$$

This means that all nodes in a cycle are collapsed to one node, preserving the arcs up to eliminating duplicates. The nodes in this graph are called components. ■

We are now ready to give the definition of HCF programs, which also serves as a method for checking this property.

Definition 4.2.4 (Head-cycle-freeness)
A program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ *and* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*, its grounded abstract version, is head-cycle-free, iff no two predicates occurring in the head of one rule depend recursively on each other, i.e., they are not both in the same node of the collapsed dependency graph* $\widehat{DG_{\mathcal{P}}} = (\hat{V}, \hat{A})$*. So the following must hold for any HCF program:*

$$\begin{aligned}\forall p_1, p_2 \in \texttt{predicates}(\mathcal{P}) :((p_1 \neq p_2 \wedge p_1 = \texttt{predicates}_A(a_1) \\ \wedge\, p_2 = \texttt{predicates}_A(a_2) \\ \wedge\, a_1, a_2 \in H \wedge (H, B) \in \mathcal{P}) \\ \rightarrow (\nexists Q \in \hat{V} : p_1, p_2 \in Q))\end{aligned}$$

■

Theorem 6.9 in [LRS97] states that for HCF programs the property $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}(\mathcal{I}^+) = \emptyset$ holds for every unfounded-free interpretation $\mathcal{I}$. However, in the general case there may be unfounded-free interpretations for which $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}(\mathcal{I}^+) = \emptyset$ does not hold.

For further optimisation, Lemma 6.4 in [LRS97] indicates that for every unfounded set $U \in U_{\mathcal{P},\mathcal{I}}$ (for a total interpretation $\mathcal{I}$) for which $U \subseteq J \subseteq HB(\mathcal{P})$ holds, also $U \subseteq \mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}(J)$ must hold. So if a program is not HCF, it is not necessary to check for all subsets of a given interpretation $\mathcal{I}$ whether it is an unfounded set – it suffices to check all subsets of $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}(I^+)$ for unfoundedness.

Yet another improvement is achieved by considering only components of a program. Before we go into detail, we need to define a few technicalities:

Definition 4.2.5
Given a set $A \subseteq \mathfrak{Atoms}$ *and a set* $P \subseteq \mathfrak{Predicates}$ *let*

$$\frac{A}{P} = \{a \in A \mid \texttt{predicates}_A(a) \in P\}$$

Additionally, given a program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$*, the grounded abstract program* $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$*, and a set* $P \subseteq \mathfrak{Predicates}$ *let*

$$\begin{aligned}sub_{\mathcal{P}}(P) = \{(H, B) \mid (H, B) \in \mathcal{P}, \\ \texttt{predicates}(H) \cap \{p \mid a \in H, p = \texttt{predicates}_A(a)\} \neq \emptyset\}\end{aligned}$$

■

For non-HCF programs we can utilise Proposition 6.11 in [LRS97]: If an interpretation $\mathcal{I}$ contains an unfounded set w.r.t. $\mathcal{P}$, there exists a component (a node Q of $\widehat{DG_{\mathcal{P}}}$) such that the restriction of $\mathcal{I}^+$ to Q ($\frac{I^+}{Q}$) contains an unfounded set w.r.t. the program which has been restricted to this component ($sub_{\mathcal{P}}(Q)$).

This means that it is sufficient to check unfounded-freeness for all restrictions of the interpretation and the program to the components of the dependency graph. If one of them is not unfounded-free, the check fails, and only if all of them are unfounded-free, the whole interpretation is unfounded-free.

Function *unfounded_free*($\mathcal{P}$: AbstractProgram; $\mathcal{I}$: SetOfLiterals) : Boolean;
var X, Y, J: SetOfLiterals;
    Q: SetOfPredicates;
begin
    for $Q \in \hat{V}, \widehat{DG_{\mathcal{P}}} = (\hat{V}, \hat{A})$ **do** (* for each component Q *)
        $X := \frac{\mathcal{I}^+}{Q}$;
        repeat (* compute $\mathcal{R}^{\omega}_{sub_{\mathcal{P}}(Q),\mathcal{I}}(\frac{\mathcal{I}^+}{Q})$ *)
                $J := X$;
                $X := \mathcal{R}_{sub_{\mathcal{P}}(Q),\mathcal{I}}(J)$
        until $J = X$;
        if $X \neq \emptyset$ (* $\mathcal{R}^{\omega}_{sub_{\mathcal{P}}(Q),\mathcal{I}}(\frac{\mathcal{I}^+}{Q}) \neq \emptyset$ *)
          then if $sub_{\mathcal{P}}(Q)$ is HCF
                then return False; (* by Theorem 6.9 in [LRS97] *)
                else (* component is not HCF *)
                    for $Y \subseteq X$ **do**
                        (* check each subset of $\mathcal{R}^{\omega}_{sub_{\mathcal{P}}(Q),\mathcal{I}}(\frac{\mathcal{I}^+}{Q})$ for unfoundedness *)
                        if $(Y \neq \emptyset) \wedge (Y \in U_{sub_{\mathcal{P}}(Q),\mathcal{I}})$
                          then return False;
                    end;
    end; (* all components are unfounded-free *)
    return True; (* by Lemma 6.4 in [LRS97] *)
end;

Figure 4.1: A function which checks unfounded-freeness of $\mathcal{I}$ given a program $\mathcal{P}$, originally defined in [LRS97]

Combined with the optimisation for HCF programs, we arrive at the algorithm for checking unfounded-freeness depicted in Figure 4.1 on the next page, which originally appeared in [LRS97].

For each restriction of the program and the interpretation to a component the algorithm first computes the fixpoint of $\mathcal{R}$. If the program restricted to the component is head-cycle-free and the fixed point is not $\emptyset$, the original interpretation is not unfounded-free w.r.t. the complete program by virtue of Lemma 6.4, Theorem 6.9, and Proposition 6.11 in [LRS97]. We may thus return False.

However, if the program restricted to the component is not head-cycle-free, we have to consider every subset of the fixed point and check whether it is an unfounded set w.r.t. the program restricted to the component and the original interpretation. If one such set is unfounded, we know by virtue of Lemma 6.4 and Proposition 6.11 in [LRS97] that the original interpretation is not unfounded-free w.r.t. the complete program, and so False is returned.

If the function never returned False while considering all restrictions to components, then no non-empty interpretation (restricted to a component) contains an unfounded set w.r.t. the program (restricted to the same component). By Proposition 6.11 in [LRS97], the original interpretation is then unfounded-free w.r.t. the entire program, and we may return True.

4.3 Operators for the Computation of Stable Models

In order to compute stable models, we introduce several operators, which will be directly used in the algorithm. The first one is the immediate consequence operator adapted to the disjunctive case.

$\mathcal{T}_\mathcal{P}$ derives those atoms from an arbitrary (possibly partial) interpretation which are sure to belong to every model which is a superset of the given partial interpretation. This is the case for a head atom which occurs in a rule, in which the body evaluates to true and all head atoms except the one we consider evaluate to false. Note that only positive literals are derived from an initial set of (negation-as-failure) literals.

Definition 4.3.1 ($\mathcal{T}_\mathcal{P}$ – the immediate consequence operator)
Given a program $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ and its grounded abstract version $\mathcal{P} =$ $\texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{T}_\mathcal{P}$ is defined as

$$\mathcal{T}_\mathcal{P} : \ \mathbb{P}(HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}))) \ \rightarrow \ \mathbb{P}(HB(\mathcal{P}))$$
$$\mathcal{T}_\mathcal{P}(\mathcal{I}) = \{a \in HB(\mathcal{P}) \mid \exists (H, B) \in \mathcal{P} : (a \in H) \wedge (H - \{a\} \subseteq \texttt{not}\ (\mathcal{I})) \wedge (B \subseteq \mathcal{I})\}$$

■

Note that $\mathcal{T}_\mathcal{P}$ cannot infer anything from strong constraints, since their head never contains any atom.

A generalisation of $\mathcal{T}_\mathcal{P}$ is $\overline{\mathcal{T}}_\mathcal{P}$, which does not only infer new atoms, but also keeps all literals from its input set. $\overline{\mathcal{T}}_\mathcal{P}$ is often termed *inflationary immediate consequence operator*.

Definition 4.3.2 ($\overline{\mathcal{T}}_\mathcal{P}$ – inflationary immediate consequence operator)
Given the preconditions of Definition 4.3.1, let

$$\overline{\mathcal{T}}_\mathcal{P} : \mathbb{P}(HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P})) \rightarrow \mathbb{P}(HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}))$$
$$\overline{\mathcal{T}}_\mathcal{P}(\mathcal{I}) = \mathcal{I} \cup \mathcal{T}_\mathcal{P}(\mathcal{I})$$

■

Obviously, this operators are too weak to generate models since no negative information can be derived.

To this end we define an extension of the operator used in [vRS91] to define the well-founded model, called $\mathcal{W}_\mathcal{P}$, as proposed in [LRS97]. In addition to $\mathcal{T}_\mathcal{P}(\mathcal{I})$, it derives the negation of the greatest unfounded set as negative information. Since the greatest unfounded set need not exist in the disjunctive case, the domain of this operator is $\mathbf{I}_\mathcal{P}$, the interpretations which have such a greatest unfounded set. This complies with the notion of negation-as-failure, since the greatest unfounded set is the largest set of atoms which are failed to be derived in this step of the computation.

Definition 4.3.3 ($\mathcal{W}_\mathcal{P}$ – the well-founded operator)
Given the preconditions of Definition 4.3.1, let

$$\mathcal{W}_\mathcal{P} : \mathbf{I}_\mathcal{P} \rightarrow \mathbb{P}(HB(\mathcal{P}) \cup HB^{\texttt{not}}(\mathcal{P}))$$
$$\mathcal{W}_\mathcal{P}(I) = \mathcal{T}_\mathcal{P}(I) \cup \texttt{not}\ (GUS_\mathcal{P}(I))$$

■

However, we have not given a constructive definition of $GUS_\mathcal{P}(\mathcal{I})$ yet. We define an operator $\Phi_{\mathcal{I},\mathcal{P}}$ which derives the complementary set of $GUS_\mathcal{P}(\mathcal{I})$ w.r.t. $HB(\mathcal{P})$, i.e., $HB(\mathcal{P}) - GUS_\mathcal{P}(\mathcal{I})$.

$\Phi_{\mathcal{I},\mathcal{P}}$derives exactly those atoms which are not contained in any unfounded set of the given interpretation, which must be in $\mathbf{I}_\mathcal{P}$. This means that we choose those head atoms for which some rule exists, so that the following conditions are met: the body is sure not to be false, that no positive body literal is in an unfounded set (note that this works if we apply the operator to sets which do not contain any unfounded atom only), and no head atom is already in $\mathcal{I}$.

Definition 4.3.4 ($\Phi_{\mathcal{I},\mathcal{P}}$ – inverse greatest unfounded set operator)
Given the preconditions of Definition 4.3.1, and additionally an interpretation which has the greatest unfounded set w.r.t. $\mathcal{P}$, $\mathcal{I} \in \mathbf{I}_\mathcal{P}$, we define

$$\Phi_{\mathcal{I},\mathcal{P}} : HB(\mathcal{P}) \rightarrow HB(\mathcal{P})$$
$$\Phi_{\mathcal{I},\mathcal{P}}(X) = \{a \mid @,\exists(H,B) \in \texttt{grounding}(\texttt{abstract}(\mathcal{P})) : a \in H \wedge B \cap \texttt{not}\ (I) = \emptyset \wedge B^+ \subseteq X \wedge H \cap I = \emptyset\}$$

■

Note that neither strong nor weak constraints interfere with this operator. In fact strong constraints do not have any impact on unfounded sets at all because these are defined by atoms occuring in heads.

In [LRS97], it has been proved that the sequence $\phi_0 = I, \ldots, \phi_k = I \cup \Phi_{\mathcal{I},\mathcal{P}}(\phi_{k-1})$ converges to a limit ϕ_λ, where λ is polynomial in $|HB(\mathcal{P})|$, and can therefore be used to efficiently compute $GUS_{\mathcal{P}}(\mathcal{I})$ of an arbitrary $\mathcal{I} \in \mathbf{I}_{\mathcal{P}}$.

Now we are finally in the position to compute $\mathcal{W}_{\mathcal{P}}$. By Proposition 5.6 in [LRS97], the sequence $W_0 = \emptyset, \ldots, W_n = \mathcal{W}_{\mathcal{P}}(W_{n-1})$ converges to a limit $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$ for every grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ and $\forall M \in SM(\mathcal{P}) : M \supseteq \mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$. Note that we could extend this result from $\text{DATALOG}^{not,\vee}$ to $\text{DATALOG}^{not,\vee,w}$ because strong constraints do not interfere with the operators as noted above, and weak constraints are not even considered in the definition, so they do not alter the behaviour of these operators either.

4.4 Possibly-true Conjunctions

Considering possibly-true conjunctions is a crucial point in the computation. One may think of possibly-true conjunctions as those negative rules in the body combined with one atom in the head which are needed to satisfy a rule which is not yet completely satisfied because of negation-as-failure literals in the body, which are left undefined by some partial interpretation.

Definition 4.4.1 (Possibly-true Conjunctions)
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, where $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ and a (possibly partial) interpretation $\mathcal{I}$, a possibly-true conjunction is the set of one positive literal appearing in the head of some rule and all negative literals in the body (4.4), if the head is not true w.r.t. $\mathcal{I}$ (4.5), the positive part of the body is true w.r.t. $\mathcal{I}$ (4.6), and the negative part of the body is not false w.r.t. $\mathcal{I}$ (4.7).

$$
\begin{aligned}
PT_{\mathcal{P}}(\mathcal{I}) = \{\{a, \texttt{not } b_1, \ldots, \texttt{not } b_n\} \mid\ & \exists (H, B) \in \mathcal{P} : \\
& a \in H, \{\texttt{not } b_1, \ldots, \texttt{not } b_n\} = B^-, && (4.4)\\
& H \cap I = \emptyset, && (4.5)\\
& B^+ \subseteq I, && (4.6)\\
& B^- \cap \texttt{not } (I) = \emptyset\} && (4.7)
\end{aligned}
$$

■

Example 4.4.1
Consider the program $\mathcal{P}_1$ from Example 4.1.1 and let $\mathcal{I} = \{\texttt{creature}\}$. Then, the possibly-true conjunctions are:

$$PT_{\mathcal{P}}(\mathcal{I}) = \{\{\texttt{woman}, \texttt{not animal}\}, \{\texttt{man}, \texttt{not animal}\}\}$$

Note that the second rule does not contribute any possibly-true conjunction, since the positive part of the body is not true w.r.t. $\mathcal{I}$. ♦

Note that neither strong nor weak constraints can contribute any possibly-true conjunction.

4.5 A Preliminary Algorithm for DATALOG$^{not,\vee}$ Programs

We are now able to reproduce the algorithm given in [LRS97] for the computation of stable models of DATALOG$^{not,\vee}$ programs. Here it is defined for DATALOG$^{not,\vee,w}$ programs, since no constraints whatsoever are considered in this algorithm. This is due to the fact that the semantics of the operators does not change when defined over programs with constraints.

The algorithm relies mainly on the well-founded operator (Definition 4.3.3), and on the notion of possibly-true conjunctions (Definition 4.4.1) combined with the inflationary immediate consequence operator (Definition 4.3.2). It also uses the efficient algorithm for checking the unfounded-free property of interpretations defined in Section 4.1. The complete algorithm is depicted in Figure 4.5 on page 86, and we will now analyse it, starting with the part marked as (* Main *).

As stated above, $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is contained in every stable model of $\mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee}}$ (if any stable model exists for $\mathcal{P}$) by Proposition 5.6 in [LRS97]. So the main function starts by calculating $\mathcal{W}_{\mathcal{P}}(\emptyset)$, $\mathcal{W}_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}(\emptyset))$ and so on, until the fixpoint $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is reached. This is done in the **repeat** ... **until** loop.

Then, we have to check whether $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is a stable model. By virtue of Proposition 6.16 in [LRS97] this is the case if $PT_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)) = \emptyset$. Corollary 6.17 in [LRS97] also tells us that if this condition holds, $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is the only stable model of $\mathcal{P}$. This is achieved by the test right after the **repeat** ... **until** loop.

However, if $PT_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)) \neq \emptyset$, we have to consider those possibly-true conjunctions. To this end, in [LRS97] a *computation* is defined:

Definition 4.5.1 (Computation [LRS97])
A sequence of sets of ground negation-as-failure literals $\{\mathcal{V}_n^{\mathcal{P}} \mid n \in \mathbb{N}\}$ (note that this set has infitely many members) is called a computation of $\mathcal{P}$, if

$$\mathcal{V}_0 = \mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$$

$$\mathcal{V}_{n+1} = \begin{cases} \overline{T}_{\mathcal{P}}(\mathcal{V}_n) & \text{if } \overline{T}_{\mathcal{P}}(\mathcal{V}_n) \neq \mathcal{V}_n \\ X \cup \mathcal{V}_n, X \in PT_{\mathcal{P}}(\mathcal{V}_n) & \text{if } \overline{T}_{\mathcal{P}}(\mathcal{V}_n) = \mathcal{V}_n, PT_{\mathcal{P}}(\mathcal{V}_n) \neq \emptyset \\ \mathcal{V}_n & \text{if } \overline{T}_{\mathcal{P}}(\mathcal{V}_n) = \mathcal{V}_n, PT_{\mathcal{P}}(\mathcal{V}_n) = \emptyset \end{cases}$$

■

For all computations $\{\mathcal{V}_n^{\mathcal{P}} \mid n \in \mathbb{N}\}$ $\exists k : \forall j > k : \mathcal{V}_k^{\mathcal{P}} = \mathcal{V}_j^{\mathcal{P}}$ holds. We refer to the respective $\mathcal{V}_k^{\mathcal{P}}$s as $\mathcal{V}_{\omega}^{\mathcal{P}}$. It follows directly from our definition (and is in part also formally proved as Lemma 6.21 of [LRS97]), that $\mathcal{V}_{\omega}^{\mathcal{P}}$ must be a fixpoint of $\overline{T}_{\mathcal{P}}$ and $PT_{\mathcal{P}}(\mathcal{V}_{\omega}^{\mathcal{P}}) = \emptyset$ must hold, too. For this reason, it is safe to split an actual computation into two alternating phases: computing a fixpoint to $\overline{T}_{\mathcal{P}}$ and choosing a possibly-true conjunction – until no possibly-true conjunction exists.

Theorem 6.22 of [LRS97] states that for each computation $\{\mathcal{V}_n^{\mathcal{P}} \mid n \in \mathbb{N}\}$, if $\mathcal{V}_{\omega}^{\mathcal{P}} \cup \texttt{not}\ (HB(\mathcal{P}) - \mathcal{V}_{\omega}^{\mathcal{P}})$ (i.e., the set $\mathcal{V}_{\omega}^{\mathcal{P}}$ plus all atoms which are undefined w.r.t. $\mathcal{V}_{\omega}^{\mathcal{P}}$ considered negated) is unfounded-free, then its positive part is a stable model. Finally, Theorem 6.23 in [LRS97] states that every stable model of some program $\mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee}}$ is of the form $\mathcal{V}_{\omega}^{\mathcal{P}} \cup \texttt{not}\ (HB(\mathcal{P}) - \mathcal{V}_{\omega}^{\mathcal{P}})$ for some computation $\{\mathcal{V}_n^{\mathcal{P}} \mid n \in \mathbb{N}\}$.

We describe the computation of all $\mathcal{V}_\omega^\mathcal{P}$ as a recursive procedure, which essentially encodes a depth first search in the tree in which $\mathcal{W}_\mathcal{P}^\omega(\emptyset)$ is the root node and each of the other nodes has as many successors as possibly-true conjunctions exist, where each successor represents the set of literals after a particular possibly-true conjunction has been chosen and subsequently the fixpoint of $\overline{T}_\mathcal{P}$ has been computed. Every path in this tree represents a computation up to $\mathcal{V}_\omega^\mathcal{P}$.

The procedure starts at an arbitrary node of this tree. We know that V_n, the set of literals representing this node, is a fixpoint of $\overline{T}_\mathcal{P}$. There are two cases:

1. There are no possibly-true conjunctions. Then we have arrived at some $\mathcal{V}_\omega^\mathcal{P}$ and need to check whether $\mathcal{V}_\omega^\mathcal{P} \cup \texttt{not}\ (HB(\mathcal{P}) - \mathcal{V}_\omega^\mathcal{P})$ is unfounded-free. If it is, we have found a stable model (by Theorem 6.22 in [LRS97]) and output it. Then the procedure backtracks.

2. Possibly-true conjunctions exist. Then we have to consider them all, one at a time. We do that depth first, i.e., we choose one possibly-true conjunction, add it to the preliminary interpretation, compute its fixpoint w.r.t. $\overline{T}_\mathcal{P}$, and call the procedure recursively, since we have then arrived at a new node in the tree. After the procedure has returned, we add the next possibly-true conjunction , compute the fixpoint w.r.t. $\overline{T}_\mathcal{P}$, call the procedure recursively and so on until every possibly-true conjunction has been chosen.

   Before the procedure is called recursively, a consistency check is made. Since proper interpretations and therefore also models and stable models have to be consistent by definition and subsequent applications of operators can only add literals, no model can be computed in the subtree below a node represented by an inconsistent set of literals. We may thus safely skip the recursive call in this case.

For the output, the negative part of the models is removed to be consistent with Definition 2.6.9.

4.6 Strong Constraints

So far we did not consider any constraints at all. Let us now focus on how we can integrate strong constraints in the algorithm.

It is clear that we need not change much of it, since strong constraints just invalidate some of the models which have been computed. So a naïve approach would be to add constraint checks after a stable model has been found and before doing the output.

However, one can do better: If during a computation we arrive at an interpretation which makes the body of a strong constraint true and thus violates it, it is clear that we can abandon this computation, since the inflationary operator and the subsequent additions of possibly true conjunctions cannot add anything which makes the strong constraint false or undefined again.

But one has to be careful with atoms which are yet undefined w.r.t. the preliminary interpretation. In contrast to the satisfaction criterion for partial interpretations, we have to take into account that their truth may be derived later in the computation and thus if there is any undefined negation-as-failure

Input: A grounded abstract program $\mathcal{P}$
Output: A stable model of $\mathcal{P}$ (if any).

Procedure *Compute_Stable*(V_n:SetOfLit); (* Recursive Procedure *)
var X, V'_n, V'_{n+1}: SetOfLit;
  if $PT_{\mathcal{P}}(V_n) = \emptyset$ (* V_n^+ is a model of $\mathcal{P}$ *)
    then if *unfounded_free*($\mathcal{P}$, $V_n \cup \texttt{not}\ (HB(\mathcal{P}) - V_n)$)
      then $M := V_n^+$; **output** "M is a stable model of $\mathcal{P}$";
    end_if
    else for $X \in PT_{\mathcal{P}}(V_n)$ **do**
      $V'_{n+1} := V_n \cup X$; (* Choose a possibly-true conjunction *)
      repeat (* Compute a fixpoint of $\overline{T}_{\mathcal{P}}$ *)
        $V'_n := V'_{n+1}$;
        $V'_{n+1} := \overline{T}_{LP}(V'_n)$;
      until $V'_{n+1} = V'_n$ or $V'_{n+1} \cap \texttt{not}\ (V'_{n+1}) \neq \emptyset$;
      if $V'_{n+1} \cap \texttt{not}\ (V'_{n+1}) = \emptyset$ (* V'_{n+1} is consistent *)
        then *Compute_Stable*(V'_{n+1});
    end_for
end_procedure

begin (* Main *)
  var I, J: SetOfLit;
  $I := \emptyset$;
  repeat (* Computation of $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ *)
    $J := I$;
    $I := \mathcal{W}_{\mathcal{P}}(I)$;
  until $I = J$;
  if $PT_{\mathcal{P}}(\mathrm{I}) = \emptyset$ (* $\mathcal{W}_{\mathcal{P}}(I)$ is the only stable model *)
    then $J := I^+$; **output** "J is the unique stable model of $\mathcal{P}$";
    else *Compute_Stable*(I);
end

Figure 4.2: An algorithm for the computation of Stable Models of DATALOG$^{not,\vee}$ programs, originally defined in [LRS97]

```
Function strong_violated(P: Program; I:SetOfLiterals) : Boolean
begin
    for (∅, B) ∈ P do
        if B ⊆ I
            then return True;
    end;
    return False;
end;
```

Figure 4.3: A function to determine whether a (possibly) partial interpretation violates some constraints

literal in a strong constraint in a preliminary check, we have to assume that the constraint is satisfied. In short, only constraints which are definitely violated may be considered violated.

The function which checks whether a constraint is violated is depicted in Figure 4.3.

However, when a model has been computed, we have to make sure that the totalised interpretation (in which all undefined atoms are considered false) does not violate any strong constraints. We can use the function *strong_violated* (Figure 4.3) for this purpose too, since the totalised interpretation is considered which does not contain any undefined literal.

The updated algorithm with the integrated strong constraint checks is shown in Figure 4.6 on the next page.

4.7 Where Weak Constraints Fit In

Still missing is a feature in the algorithm which admits only those models which are optimal in the sense of Definition 2.6.24. In this section we give the final extension of the algorithm which will enable us to compute the preferred models.

4.7.1 Objective Function

When optimising, one usually tries to find an extreme value (minimum or maximum) of some objective function. If we can define a function for each model with respect to some program, which is minimised exactly by its preferred models, we can implant the computation of this function into our algorithm in order to single out the preferred models. We recall the criterion for such an objective function:

Criterion 1 (Requirement for the Objective Function)
We have to find an objective function $H_M^{\mathcal{P}}$ which assigns a value to each candidate model in the context of a particular program $\mathcal{P}$, such that for two candidate models $M_1, M_2 \in SEM(\mathcal{P})$ the following holds:
If all sums of weights of violated constraints are equal in M_1 and M_2 in layers greater than n, and the sum of the weights in layer n in M_1 is greater than the corresponding sum in M_2, then $H_{M_1}^{\mathcal{P}} > H_{M_2}^{\mathcal{P}}$. ◎

Input: An abstract program $\mathcal{P}$
Output: A stable model of $\mathcal{P}$ (if any).

Procedure *Compute_Stable*(V_n:SetOfLit); (* Recursive Procedure *)
var X, V'_n, V'_{n+1}: SetOfLit;
  if $PT_{\mathcal{P}}(V_n) = \emptyset$ (* V_n^+ is a model of $\mathcal{P}$ *)
    then if (**not** *strong_violated*($\mathcal{P}$, $V_n \cup \texttt{not}\ (HB(\mathcal{P}) - V_n)$))
      $\wedge$ *unfounded_free*($\mathcal{P}$, $V_n \cup \texttt{not}\ (HB(\mathcal{P}) - V_n)$)
      then $M := V_n^+$; **output** "M is a stable model of $\mathcal{P}$";
    end_if
  else for $X \in PT_{\mathcal{P}}(V_n)$ **do**
    $V'_{n+1} := V_n \cup X$; (* Choose a possibly-true conjunction *)
    repeat (* Compute a fixpoint of $\overline{T}_{\mathcal{P}}$ *)
      $V'_n := V'_{n+1}$;
      $V'_{n+1} := \overline{T}_{LP}(V'_n)$;
    until $V'_{n+1} = V'_n$ or $V'_{n+1} \cap \texttt{not}\ (V'_{n+1}) \neq \emptyset$;
    if $V'_{n+1} \cap \texttt{not}\ (V'_{n+1}) = \emptyset$ (* V'_{n+1} is consistent *)
      $\wedge$ (**not** *strong_violated*($\mathcal{P}$,V'_{n+1}))
      then *Compute_Stable*(V'_{n+1});
  end_for
end_procedure

begin (* Main *)
  var I, J: SetOfLit;
  $I := \emptyset$;
  repeat (* Computation of $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ *)
    $J := I$;
    $I := \mathcal{W}_{\mathcal{P}}(I)$;
  until $I = J$;
  if $PT_{\mathcal{P}}(\mathrm{I}) = \emptyset$ (* $\mathcal{W}_{\mathcal{P}}(I)$ is the only stable model *)
    then if not *strong_violated*($\mathcal{P}$, $I \cup \texttt{not}\ (HB(\mathcal{P}) - I)$)
      then $J := I^+$; **output** "J is the unique stable model of $\mathcal{P}$";
    end_if
  else *Compute_Stable*(I); **end**

Figure 4.4: An algorithm for the computation of Stable Models of DATALOG$^{not,\vee,s}$ programs, extended from the algorithm in Figure 4.5 on page 86

If we then take those models which minimise this function, we get exactly those models which violate constraints at the lowest possible layer and whose sum of weights in this lowest layer is minimal, too.

Obviously, this function must incorporate some information about the layers in which constraints are violated – the higher the layer in which violated constraints exist, the higher the function value – and also some information about the added weights of the violated constraints, but one must be careful that the weight information does not interfere with the layer information.

So we need some function for the layers, which delivers a higher value for layer l if one constraint with some tiny weight is violated in it than the value of the objective function for a scenario in which all constraints of levels lower than l with enormous weights are violated.

Some problem arises here: The weights may be negative or zero. We therefore normalise the weights to be strictly positive in our functions as follows: Given some numbers in the interval $[min, max]$, we simply add $-min + 1$, so that all numbers are positive and in the range $[1, max - min + 1]$.

Definition 4.7.1 (Positivised Weights)
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *the positivised weights are defined as:*

$$\begin{aligned} w_C^+ =& (w_C - w_{min}^{\mathcal{P}} + 1), C \in \mathcal{P} \\ w_{max}^{\mathcal{P},+} =& (w_{max}^{\mathcal{P}} - w_{min}^{\mathcal{P}} + 1) \end{aligned}$$

■

Definition 4.7.2
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *we denote the number of (abstract) weak constraints in* $\mathcal{P}$ *as* $WC^{\mathcal{P}}$.

$$WC^{\mathcal{P}} = |\mathit{Constraints}_{\mathit{abstract}} \cap \mathcal{P}|$$

■

Using this we can define a penalty function for a layer, as described above. Of course, this function must interact nicely with our objective function, which will be defined shortly. In particular, the layer penalty function must be strictly greater than the objective function of some model and the same program, in which any number of weak constraints in layers lower than the considered one are violated.

Definition 4.7.3 (Layer Penalty Function)
Given a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, *the layer penalty function is defined as:*

$$\begin{aligned} & f_{\mathcal{P}} : \texttt{Layers}(\mathcal{P}) \to \mathbb{N} \\ & f_{\mathcal{P}}(1) = 1 \\ & f_{\mathcal{P}}(n) = f_{\mathcal{P}}(n-1) \cdot WC^{\mathcal{P}} \cdot w_{max}^{\mathcal{P},+} + 1, n \in \texttt{Layers}(\mathcal{P}), n > 1 \end{aligned}$$

■

Now we define the objective function as the sum of the added weights of the violated weak constraints of each layer, multiplied by the respective layer penalty.

Definition 4.7.4 (Objective Function for a Given Model)
Let $\mathcal{P}$ be a grounded abstract program $\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}'))$, $\mathcal{P}' \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, and M one of its candidate models. The objective function value $H_M^{\mathcal{P}}$ is then defined as follows:

$$H_M^{\mathcal{P}} = \sum_{i=1}^{l_{max}^{\mathcal{P}}} (f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{M,\mathcal{P}}} w_N^{\mathcal{P},+})$$

■

4.7.2 Preferred Models and Objective Function Minima Coincide

It is important that this objective function is minimal for the preferred models of a program but for no other candidate models. We will now prove that a candidate model is a preferred model iff it is a minimum of the objective function.

First, we show that the layer penalty function is always strictly greater than the sum of the objective function up to the preceding layer.

Lemma 4.7.1
Given a program $\mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee,w}}$ and one of its candidate models $M \in SEM(\mathcal{P})$, and let

$$H_M^{\mathcal{P}}(k) = \sum_{i=1}^{k} (f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{M,\mathcal{P}}} w_N^{\mathcal{P},+})$$

then

$$\forall 0 \leq k < l_{max}^{\mathcal{P}} : f_{\mathcal{P}}(k+1) > H_M^{\mathcal{P}}(k)$$

holds.

Proof Let $k = 0$, then $f_{\mathcal{P}}(1) = 1$ and $H_M^{\mathcal{P}}(0) = 0$, since 0 is neutral w.r.t. the sum, therefore $f_{\mathcal{P}}(k+1) > H_M^{\mathcal{P}}(k)$ holds.

Let $0 < k < l^{\mathcal{P}}_{max}$, then

$$
\begin{aligned}
H^{\mathcal{P}}_M(k) &= \sum_{i=1}^{k}(f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{M,\mathcal{P}}} w_N^{\mathcal{P},+}) = \\
&= f_{\mathcal{P}}(1) \cdot \overbrace{\sum_{N \in N_1^{M,\mathcal{P}}} w_N^{\mathcal{P},+}}^{\leq |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(1)| \cdot w^{\mathcal{P}}_{max}} + \cdots + f_{\mathcal{P}}(k) \cdot \overbrace{\sum_{N \in N_k^{M,\mathcal{P}}} w_N^{\mathcal{P},+}}^{\leq |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(k)| \cdot w^{\mathcal{P}}_{max}} \leq \\
&\leq f_{\mathcal{P}}(1) \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(1)| \cdot w^{\mathcal{P}}_{max} + \cdots + f_{\mathcal{P}}(k) \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(k)| \cdot w^{\mathcal{P}}_{max} = \\
&= (\overbrace{f_{\mathcal{P}}(1)}^{\leq f_{\mathcal{P}}(k)} \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(1)| + \cdots + \overbrace{f_{\mathcal{P}}(k)}^{\leq f_{\mathcal{P}}(k)} \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(k)|) \cdot w^{\mathcal{P}}_{max} \leq \\
&\leq (f_{\mathcal{P}}(k) \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(1)| + \cdots + f_{\mathcal{P}}(k) \cdot |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(k)|) \cdot w^{\mathcal{P}}_{max} = \\
&= f_{\mathcal{P}}(k) \cdot \overbrace{(|\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(1)| + \cdots + |\mathtt{c}_{\mathtt{layer}_{\mathcal{P}}}(k)|)}^{\leq WC^{\mathcal{P}}} \cdot w^{\mathcal{P}}_{max} \leq \\
&\leq f_{\mathcal{P}}(k) \cdot WC^{\mathcal{P}} \cdot w^{\mathcal{P}}_{max} < \\
&< f_{\mathcal{P}}(k) \cdot WC^{\mathcal{P}} \cdot w^{\mathcal{P}}_{max} + 1 \\
&= f_{\mathcal{P}}(k+1)
\end{aligned}
$$

So in total $\forall 0 \leq k < l^{\mathcal{P}}_{max} : f_{\mathcal{P}}(k+1) > H^{\mathcal{P}}_M(k)$ holds. △

We are now in the position to prove the essential theorem:

Theorem 4.7.1
A subset N of candidate models of a program $\mathcal{P} \in \Pi_{\text{DATALOG}^{not,\vee,w}}$, $N \subseteq SEM(\mathcal{P})$, is the set of preferred models iff $N = \{\min_{M \in SEM(\mathcal{P})} H^{\mathcal{P}}_M\}$.

Proof ($\Rightarrow$): Preferred models must minimise the sum of the weights of violated weak constraints of the greatest layer in the program, i.e., they must be in $PREF_{SEM}(\mathcal{P}, l^{\mathcal{P}}_{max})$. We will show that for any two candidate models M and M', where M minimises the sum of weak constraints of layer $l^{\mathcal{P}}_{max}$ and M' does not, $H^{\mathcal{P}}_M < H^{\mathcal{P}}_{M'}$ holds – actually we show $H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_M > 0$:

As it can be seen from Definition 2.6.24,

$$\sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}} < \sum_{\mathcal{W} \in N^{M',\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}}$$

must hold for these two models, since M is minimal and M' is not. It can be easily verified that the above also holds for positivised weights:

$$\sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w^{\mathcal{P},+}_{\mathcal{W}} < \sum_{\mathcal{W} \in N^{M',\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w^{\mathcal{P},+}_{\mathcal{W}} \tag{4.8}$$

$$H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_{M} = H^{\mathcal{P}}_{M'}(l^{\mathcal{P}}_{max} - 1) + f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) \cdot \sum_{N \in N^{M',\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w^{\mathcal{P},+}_{N}$$
$$- H^{\mathcal{P}}_{M}(l^{\mathcal{P}}_{max} - 1) - f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) \cdot \sum_{N \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w^{\mathcal{P},+}_{N}$$

by (4.8) above and since all weights are positive integers,

$$= \overbrace{H^{\mathcal{P}}_{M'}(l^{\mathcal{P}}_{max} - 1)}^{\geq 0} - H^{\mathcal{P}}_{M}(l^{\mathcal{P}}_{max} - 1) + f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) \cdot k, \quad k \geq 1$$
$$\geq - H^{\mathcal{P}}_{M}(l^{\mathcal{P}}_{max} - 1) + f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) > 0$$

because by virtue of Lemma 4.7.1 $f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) > H^{\mathcal{P}}_{M}(l^{\mathcal{P}}_{max} - 1)$ holds, and therefore in total

$$H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_{M} > 0$$

We have shown that $H^{\mathcal{P}}_{M}$is minimal for those models which minimise the sum of weights of the violated constraints in the greatest layer, i.e., which are in $\mathit{PREF}_{SEM}(\mathcal{P}, l^{\mathcal{P}}_{max})$. Among those for any layer $1 \leq i < l^{\mathcal{P}}_{max}$ the preferred models must minimise the sum of weights of the violated weak constraints in layer i, which is the criterion for being in set $\mathit{PREF}_{SEM}(\mathcal{P}, i)$. Let again be M a model which minimises all sums of violated weak constraints in layers greater or equal than i, and M' one which does not minimise the sum of violated weak constraints in layer i, but all in layers greater than i. Again we show that $H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_{M} > 0$ holds:

Again, let us observe that the following holds:

$$\sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{i}} w^{\mathcal{P},+}_{\mathcal{W}} < \sum_{\mathcal{W} \in N^{M',\mathcal{P}}_{i}} w^{\mathcal{P},+}_{\mathcal{W}} \tag{4.9}$$

Also, since M and M' both minimise the added weights of violated weak constraints in layers greater than i, we have:

$$\sum_{j=i+1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(j) \cdot \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{j}} w_{\mathcal{W}}) = \sum_{j=i+1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(j) \cdot \sum_{\mathcal{W} \in N^{M',\mathcal{P}}_{j}} w_{\mathcal{W}}) \tag{4.10}$$

We proceed similar to the case of the greatest layer:

$$H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_{M} = H^{\mathcal{P}}_{M'}(i-1) + f_{\mathcal{P}}(i) \cdot \sum_{N \in N^{M',\mathcal{P}}_{i}} w^{\mathcal{P},+}_{N} + \sum_{j=i+1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(j) \cdot \sum_{\mathcal{W} \in N^{M',\mathcal{P}}_{j}} w_{\mathcal{W}})$$
$$- H^{\mathcal{P}}_{M}(i-1) - f_{\mathcal{P}}(i) \cdot \sum_{N \in N^{M,\mathcal{P}}_{i}} w^{\mathcal{P},+}_{N} - \sum_{j=i+1}^{l^{\mathcal{P}}_{max}} (f_{\mathcal{P}}(j) \cdot \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{j}} w_{\mathcal{W}})$$

by (4.10) above

$$= H^{\mathcal{P}}_{M'}(i-1) + f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{M',\mathcal{P}}} w_N^{\mathcal{P},+} - H^{\mathcal{P}}_{M}(i-1) - f_{\mathcal{P}}(i) \cdot \sum_{N \in N_i^{M,\mathcal{P}}} w_N^{\mathcal{P},+}$$

by (4.9) above and since all weights are integers

$$= \overbrace{H^{\mathcal{P}}_{M'}(i-1)}^{\geq 0} - H^{\mathcal{P}}_{M}(i-1) + f_{\mathcal{P}}(l^{\mathcal{P}}_{max}) \cdot k, \quad k \geq 1$$
$$\geq - H^{\mathcal{P}}_{M}(i-1) + f_{\mathcal{P}}(i) > 0$$

because by virtue of Lemma 4.7.1 $f_{\mathcal{P}}(i) > H^{\mathcal{P}}_{M}(i-1)$ holds, and therefore in total

$H^{\mathcal{P}}_{M'} - H^{\mathcal{P}}_{M} > 0$

We have proven the ($\Rightarrow$) direction. ($\Leftarrow$): We have to show that

$$M \in \{M_1 \mid H^{\mathcal{P}}_{M_1} = \min_{M_2 \in SEM(\mathcal{P})} H^{\mathcal{P}}_{M_2}\} \Rightarrow M \in PREF_{SEM}(\mathcal{P})$$

We start by showing

$$M \in \{M_1 \mid H^{\mathcal{P}}_{M_1} = \min_{M_2 \in SEM(\mathcal{P})} H^{\mathcal{P}}_{M_2}\} \Rightarrow M \in PREF_{SEM}(\mathcal{P}, l^{\mathcal{P}}_{max})$$

We do this indirectly. So assume $M_1 \in SEM(\mathcal{P})$ so that

$$H^{\mathcal{P}}_{M_1} = \min_{M \in SEM(\mathcal{P})} H^{\mathcal{P}}_{M}$$

but

$$\sum_{\mathcal{W} \in N^{M_1,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}} > \min_{M \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}}$$

This means that some $M_2 \in SEM(\mathcal{P})$ exists, so that

$$\sum_{\mathcal{W} \in N^{M_2,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}} = \min_{M \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N^{M,\mathcal{P}}_{l^{\mathcal{P}}_{max}}} w_{\mathcal{W}}$$

We show that $H^{\mathcal{P}}_{M_2} < H^{\mathcal{P}}_{M_1}$, which is a contradiction to the assumption that $H^{\mathcal{P}}_{M_1}$ is minimal. Again $H^{\mathcal{P}}_{M_1} - H^{\mathcal{P}}_{M_2} > 0$ must hold, analogously to the proofs above.

Next we show

$$\forall 1 \leq i < l^{\mathcal{P}}_{max} : M \in \{M_1 \mid H^{\mathcal{P}}_{M_1} = \min_{M_2 \in SEM(\mathcal{P})} H^{\mathcal{P}}_{M_2}\} \Rightarrow M \in PREF_{SEM}(\mathcal{P}, i)$$

under the assumption that $M \in \mathit{PREF}_{SEM}(\mathcal{P}, i+1)$, which also entails membership in $\mathit{PREF}_{SEM}(\mathcal{P}, j), i+1 < j < l^{\mathcal{P}}_{max}$.

As above, we prove by contradiction: So assume $M_1 \in SEM(\mathcal{P})$ so that

$$H^{\mathcal{P}}_{M_1} = \min_{M \in SEM(\mathcal{P})} H^{\mathcal{P}}_{M}$$

but

$$\sum_{\mathcal{W} \in N_i^{M_1, \mathcal{P}}} w_{\mathcal{W}} > \min_{M \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N_i^{M, \mathcal{P}}} w_{\mathcal{W}}$$

This means that some $M_2 \in SEM(\mathcal{P})$ exists, so that

$$\sum_{\mathcal{W} \in N_i^{M_2, \mathcal{P}}} w_{\mathcal{W}} = \min_{M \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N_i^{M, \mathcal{P}}} w_{\mathcal{W}}$$

Since for $M \in \mathit{PREF}_{SEM}(\mathcal{P}, j), i < j < l^{\mathcal{P}}_{max}$

$$\forall i < j < l^{\mathcal{P}}_{max} : \sum_{\mathcal{W} \in N_j^{M_1, \mathcal{P}}} w_{\mathcal{W}} = \min_{M \in SEM(\mathcal{P})} \sum_{\mathcal{W} \in N_j^{M, \mathcal{P}}} w_{\mathcal{W}}$$

holds, we can again show $H^{\mathcal{P}}_{M_1} - H^{\mathcal{P}}_{M_2} > 0$ analogously to the proofs above, thus deriving a contradiction to our assumption, and completing the proof for the $\Leftarrow$ direction of Theorem 4.7.1. △

4.7.3 Extension of the Algorithm to Compute One Preferred Model

Having proved Theorem 4.7.1, we have a way to decide among all models of a program whether it is a preferred model. A naïve approach would be to store each computed model (after the strong constraint check) together with its objective function value. As we will see later, this would have a very bad impact on space complexity.

As in the case of strong constraints, we can do better by considering the weak constraints which are already violated for sure even if the computation is not complete. Thereby we get a lower bound for the objective function. If we know that this lower bound is greater than a function value of an already computed model, we may abandon the current computation. However, if no model has been computed yet, nothing can be said for sure.

A suitable function for computing a lower bound of $H^{\mathcal{P}}_{M}$, given a possibly partial interpretation $\mathcal{I}^+ \subseteq M$ is shown in Figure 4.5 on the following page. Note that in the case of a total interpretation ($\mathcal{I}^+ = M$ in our case) the function computes the exact value of $H^{\mathcal{P}}_{M}$.

Different strategies for searching the preferred models in the choice tree are feasible. We decided to choose a greedy algorithm which computes a first model (if any) without considering the preliminary objective function values at all, thus potentially being able to prune the search space effectively.

The search process remains the same until the first model is computed. After that, only those interpretations are pursued for which the lower bound of the

```
Function compute_objective(P: Program; I: SetOfLit) : Integer
var H: Integer;
begin
      H := 0;
      for (C, l, w) ∈ P do
          if C ⊆ I
             then H := H + (f_P(l) · w);
      end;
      return H;
end;
```

Figure 4.5: A function which computes a lower bound of the objective function for a partial interpretation and the exact value of the objective function for total interpretations

objective function is less than the objective function value for the most recently computed model. Clearly, also when a leaf of the computation is reached, we have to compute the objective function value and look whether it is still smaller than the optimal value up to this step.

In this way we compute exactly the first model for which the objective function is minimal, the resulting algorithm is shown in Figure 4.7.3 on the next page.

Although for many problems the algorithm operates efficiently, this strategy might not be the best for several types of problems.

In an implementation, one could consider to enable the user to choose among different search strategies. It is even possible to implement metaheuristics like tabu search or simulated annealing, which are used successfully to solve optimisation problems mainly in the field of scheduling.

4.7.4 An Example

To show how the algorithm works, let us consider a simple example.

Example 4.7.1
Imagine some humanoid agent, which acts according to the following program (imagine that the agent has just got up, looked out of the window and realized that it is sunny without wondering whether it is a weekday or weekend. However, he[1] *is aware of several categorical principles of his existence.*

[1] without loss of generality, assume it is a male agent

Input: A grounded abstract program $\mathcal{P}$
Output: A preferred model of $\mathcal{P}$ (if any).

Procedure *Compute_Preferred*(V_n:SetOfLit;**var** M: SetOfAt; **var** *minCost*: Integer);
var X, V'_n, V'_{n+1}: SetOfLit;
  if $PT_{\mathcal{P}}(V_n) = \emptyset$ (* V_n^+ is a model of $\mathcal{P}$ *)
    then if (**not** *strong_violated*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$)))
      $\wedge$ *unfounded_free*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$))
      $\wedge$ *compute_objective*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$)) $<$ *minCost*;
      then $M := V_n^+$;
        minCost := *compute_objective*($\mathcal{P}$,$V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$));
        output "M is the preliminary preferred model.";
      end_if
    else for $X \in PT_{\mathcal{P}}(V_n)$ **do**
      $V'_{n+1} := V_n \cup X$; (* Choose a possibly-true conjunction *)
      repeat (* Compute a fixpoint of $\overline{T}_{\mathcal{P}}$ *)
        $V'_n := V'_{n+1}$;
        $V'_{n+1} := \overline{T}_{LP}(V'_n)$;
      until $V'_{n+1} = V'_n$ or $V'_{n+1} \cap$ `not` $(V'_{n+1}) \neq \emptyset$;
      if $V'_{n+1} \cap$ `not` $(V'_{n+1}) = \emptyset$ (* V'_{n+1} is consistent *)
        $\wedge$ (**not** *strong_violated*($\mathcal{P}$,V'_{n+1}))
        $\wedge$ *compute_objective*($\mathcal{P}$,V'_{n+1}) $<$ *minCost*
        then *Compute_Preferred*($V'_{n+1}, M, minCost$);
    end_for
end_procedure

begin (* Main *)
  var I, J: SetOfLit; M: SetOfAt; *minCost*: Integer;
  $I := \emptyset$; $minCost := +\infty$;
  repeat (* Computation of $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ *)
    $J := I$;
    $I := \mathcal{W}_{\mathcal{P}}(I)$;
  until $I = J$;
  if $PT_{\mathcal{P}}(\mathrm{I}) = \emptyset$ (* $\mathcal{W}_{\mathcal{P}}(I)$ is the only stable model *)
    then if not *strong_violated*($\mathcal{P}$, $I \cup$ `not` ($HB(\mathcal{P}) - I$))
      then $M := I^+$;
        minCost := *compute_objective*($\mathcal{P}$, $M \cup$ `not` ($HB(\mathcal{P}) - M$));
      end_if
    else *Compute_Preferred*(I,M,*minCost*);
  if $minCost \neq +\infty$
    then output "M is one of the preferred stable models of $\mathcal{P}$.";
    else output "$\mathcal{P}$ does not have any stable models.";
end

Figure 4.6: An algorithm for the computation of Preferred Stable Models of DATALOG$^{not,\vee,w}$ programs, extended from the algorithm in Figure 4.6 on page 88

$$\begin{aligned}\mathcal{P}' = \{work \vee leisure &\leftarrow \texttt{not}\ sleep.,\\ earn\_money &\leftarrow work.,\\ amusement &\leftarrow leisure.,\\ sunny.,&\\ &\Leftarrow \texttt{not}\ weekend, amusement.[2:3],\\ &\Leftarrow sunny, work.[2:1],\\ &\Leftarrow \texttt{not}\ amusement.[1:2]\}\end{aligned}$$

This means that if there is no evidence that he sleeps, he works or has leisure. If he works, he earns money. If he has leisure, he has amusement. We know that it is sunny.

Then he has several preferences: The agent is very rigorous and therefore unless he is sure that it is weekend he does not need any amusement. This is very important to him. On the equal level of importance but somewhat less strict is the preference that working while it is sunny is not a good idea. Finally, on a lower level of importance he believes that having no amusement is not good.

Will the agent decide to go to work or to stay at home (without further thinking)?

The grounded abstract program is:

$$\begin{aligned}\mathcal{P} = \texttt{grounding}(\texttt{abstract}(\mathcal{P}')) = \{&(\{work, leisure\}, \{\texttt{not}\ sleep\}),\\ &(\{earn\_money\}, \{work\}),\\ &(\{amusement\}, \{leisure\}),\\ &(\{sunny\}, \emptyset),\\ &(\{\texttt{not}\ weekend, amusement\}, 2, 3),\\ &(\{sunny, work\}, 2, 1),\\ &(\{\texttt{not}\ amusement\}, 1, 2)\}\end{aligned}$$

Next we show the Herbrand Base and the values of the layer penalty function. Also note that all weights are positive and the smallest weight is 1, so the positivised weights are identical to the original weights.

$$\begin{aligned}HB(\mathcal{P}) &= \{work, leisure, sleep, earn\_money, amusement, sunny, weekend\}\\ w^{\mathcal{P}}_{max} &= 3\\ \texttt{Layers}(\mathcal{P}) &= \{1, 2\}\\ WC^{\mathcal{P}} &= 3\\ f_{\mathcal{P}}(1) &= 1\\ f_{\mathcal{P}}(2) &= 1 \cdot 3 \cdot 3 = 9\end{aligned}$$

We will now analyse the computation of one preferred model:
We start by computing $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset)$.

$$
\begin{aligned}
\mathcal{W}_{\mathcal{P}}(\emptyset) &= \mathcal{T}_{\mathcal{P}}(\emptyset) \cup \texttt{not}\ (\mathit{GUS}_{\mathcal{P}}(\emptyset)) \\
\mathcal{T}_{\mathcal{P}}(\emptyset) &= \{sunny\} \\
\mathit{GUS}_{\mathcal{P}}(\emptyset) &= \mathit{HB}(\mathcal{P}) - \phi_\lambda \\
\phi_0 &= \emptyset \\
\phi_1 &= \Phi_{\emptyset,\mathcal{P}}(\emptyset) = \{sunny, work, leisure\} \\
\phi_2 &= \phi_1 \cup \Phi_{\emptyset,\mathcal{P}}(\phi_1) \\
\Phi_{\emptyset,\mathcal{P}}(\phi_1) &= \{earn\_money, amusement\} \\
\phi_2 &= \{sunny, work, leisure, earn\_money, amusement\} \\
\phi_3 &= \phi_2 \cup \Phi_{\emptyset,\mathcal{P}}(\phi_2) = \phi_2 = \phi_\lambda \\
\mathit{HB}(\mathcal{P}) - \phi_\lambda &= \{sleep, weekend\} \\
\texttt{not}\ (\mathit{GUS}_{\mathcal{P}}(\emptyset)) &= \{\texttt{not}\ sleep, \texttt{not}\ weekend\} \\
\mathcal{W}_{\mathcal{P}}(\emptyset) &= \{sunny, \texttt{not}\ sleep, \texttt{not}\ weekend\}
\end{aligned}
$$

This is what we get in the first step. It can be verified that this is also a fixpoint, since $\mathcal{T}_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}(\emptyset)) = \{sunny\}$ *and neither* $work$ *nor* $leisure$ *can be derived, although the prerequisite* `not` $sleep$ *is in* $\mathcal{W}_{\mathcal{P}}(\emptyset)$*, because neither* `not` $leisure$ *nor* `not` $work$ *is in* $\mathcal{W}_{\mathcal{P}}(\emptyset)$*. The greatest unfounded set stays the same. So we have* $\mathcal{W}^{\omega}_{\mathcal{P}}(\emptyset) = \{sunny, \texttt{not}\ sleep, \texttt{not}\ weekend\}$.

We have finished the first ***repeat*** *...* ***until*** *loop, and now determine*

$$
\begin{aligned}
&\mathit{PT}_{\mathcal{P}}(\{sunny, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \\
&\quad = \{\{work, \texttt{not}\ sleep\}, \{leisure, \texttt{not}\ sleep\}\} \neq \emptyset
\end{aligned}
$$

So Compute_Preferred($\{sunny, \texttt{not}\ sleep, \texttt{not}\ weekend\}$*,*$\emptyset$*,*$+\infty$*) is called. There is again the test whether the possibly-true conjunctions are empty, which fails again, so we come to choose a possibly-true conjunction.*

First, choose $\{work, \texttt{not}\ sleep\}$*, so the new interpretation is* $\{sunny,\ work,$ `not` $sleep,$ `not` $weekend\}$.

We now compute the fixpoint of $\overline{\mathcal{T}}_{\mathcal{P}}$ *for it:*

$$
\begin{aligned}
&\overline{\mathcal{T}}_{\mathcal{P}}(\{sunny, work, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \\
&\quad \{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\} \\
&\overline{\mathcal{T}}_{\mathcal{P}}(\{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \\
&\quad \{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\}
\end{aligned}
$$

We have reached a fixpoint. The consistency test and the strong constraint check are successful (after all, there are no strong constraints in our program), and compute_objective($\mathcal{P}$*,*$\{$*sunny,work,earn_money,*`not` *sleep,* `not` *weekend*$\}$*) returns 9, since* $(2, 1, sunny, work)$ *is violated. This is of course less than* $+\infty$ *(we have not computed any model yet), so we may proceed calling*

$$\mathit{Compute\_Preferred}(\{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\}, \emptyset, +\infty)$$

Note that if $(2, 1, sunny, work)$ *was a strong constraint, the computation of this branch would have been stopped here.*

In the recursive step,

$$PT_{\mathcal{P}}(\{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \emptyset$$

holds, so we have to check unfounded-freeness, which succeeds (we do not go into detail here). The strong constraint check succeeds trivially and

$$compute\_objective(\mathcal{P}, \{sunny, work, earn\_money, \texttt{not}\ sleep, \texttt{not}\ weekend\} \cup \{\texttt{not}\ leisure, \texttt{not}\ amusement\}) = 11$$

since $(2, 1, sunny, work)$ *and* $(1, 2, \texttt{not}\ amusement)$ *are violated. Clearly* $11 < +\infty$ *holds, so* $M := \{sunny, work, earn\_money\}$ *and* $minCost := 11$ *are set. Note that these are variable parameters, so the changes also affect the variables in the procedures which called this instance.*

The procedure terminates, and we get back one level into the for loop. Remember that our interpretation before the choice of the possibly-true conjunction was $\{sunny, \texttt{not}\ sleep, \texttt{not}\ weekend\}$*. We now choose the other possibly-true conjunction,* $\{leisure, \texttt{not}\ sleep\}$*, yielding the new interpretation* $\{sunny, leisure, \texttt{not}\ sleep, \texttt{not}\ weekend\}$*.*

Again, we compute the fixpoint of $\overline{T}_{\mathcal{P}}$ *for this interpretation:*

$$\begin{aligned}
&\overline{T}_{\mathcal{P}}(\{sunny, leisure, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \\
&\quad \{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\} \\
&\overline{T}_{\mathcal{P}}(\{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = \\
&\quad \{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\}
\end{aligned}$$

A fixpoint is reached. Now the consistency and strong constraint checks succeed again, and

$$compute\_objective(\mathcal{P}, \{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\}) = 27$$

since $(2, 3, \texttt{not}\ weekend, amusement)$ *is violated. But since the value of* $minCost$ *is 11 (it was a* **var** *parameter of the recursive call), the condition*

$$compute\_objective(\mathcal{P}, \{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\}) < minCost$$

is not satisfied, so if a stable model exists, which contains $\{sunny, leisure, amusement, \texttt{not}\ sleep, \texttt{not}\ weekend\}$ *(indeed there is one, as can be verified), it cannot be among the preferred ones. The computation of this branch is therefore not continued, we get back to the* **for** *loop, but there are no possibly-true conjunctions left, so the procedure terminates and we are again in the main function, output the preferred model and terminate.*

The answer to the question of what the agent will do is therefore that he will decide to go to work. However, if he later looks on the calendar and realises that $weekend.$ *holds, he will come to a different conclusion.* ♦

4.7.5 Complexity of the Algorithm

We follow the considerations in [LRS97] concerning data complexity, i.e., the complexity of the computation w.r.t. the size of the database (essentially $|HB(\mathcal{P})|$).

By Proposition 5.12 in [LRS97] $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)(\mathcal{P})$ can be computed in polynomial time. Also the computation of a fixpoint for $\overline{T}_{\mathcal{P}}$ and the evaluation of $PT_{\mathcal{P}}$ can be done in polynomial time. The functions *strong_violated* and *compute_objective* are also clearly computable in polynomial time.

Now, the procedure *Compute_Preferred* generates all computations for $\mathcal{P}$. As described above, this generation process can be seen as a tree, in which $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is the root node and each node has as many successors as possibly-true conjunctions exist. Every path in this tree represents a computation up to its limit. Each path is linear in $|HB(\mathcal{P})|$, and so the limit of a computation is reached in polynomial time.

The computationally expensive part is the check for unfounded-freeness, performed at every limit of a computation, which is reached. This function executes in at most single exponential time and polynomial space, as argued in [LRS97]. As a consequence, each computation of $\mathcal{P}$ is accomplished in single exponential time and polynomial space.

Now, the number of computations is also single exponential, so in the worst case the algorithm executes in single exponential time. Also since the procedure stack contains at most a polynomial number of entries, each comprised of an interpretation, a model, and a number, the execution of the algorithm is done using at most polynomial space.

Note that if we kept all models which minimise the objective function up to some step in the computation, the number of models would be exponential in the worst case (if all possible models are preferred), and the algorithm would use exponential space!

However, for several important classes of programs, the algorithm always terminates in polynomial time, as shown in [LRS97].

4.7.6 Computing all Preferred Stable Models

The algorithm which we presented computes one of the preferred models, but in general there may be several preferred models. In certain environments it may be sufficient to compute one preferred model, but in others it would be important to compute all.

As described in Section 4.7.5, the naïve approach of keeping all models which are so far preferred, raises the complexity of the algorithm considerably.

However, there is a way to get around this. Instead of doing **output** "M is one of the preferred stable models of $\mathcal{P}$." after the computation of one preferred model, we could respawn a similar computation, now with the optimal objective function value already known. A procedure which accomplishes this is shown in Figure 4.7 on the following page.

Compute_Preferred1 is very similar to *Compute_Preferred*. The differences are that *minCost* need not be a **var** parameter, since it is fixed, that we need only test for equality if a limit of a computation is reached, and that we have to test the lower bound of the objective function for $\leq$ *minCost* rather than $<$ *minCost* during a computation because we want to compute all models having

Procedure *Compute_Preferred1*(V_n:SetOfLit; *minCost*: Integer);
var X, V'_n, V'_{n+1}: SetOfLit;
  if $PT_{\mathcal{P}}(V_n) = \emptyset$ (* V_n^+ is a model of $\mathcal{P}$ *)
    then if (**not** *strong_violated*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$)))
      $\wedge$ *unfounded_free*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$))
      $\wedge$ *compute_objective*($\mathcal{P}$, $V_n \cup$ `not` ($HB(\mathcal{P}) - V_n$)) = *minCost*;
      then output "M is a preferred model.";
    end_if
  else for $X \in PT_{\mathcal{P}}(V_n)$ **do**
    $V'_{n+1} := V_n \cup X$; (* Choose a possibly-true conjunction *)
    repeat (* Compute a fixpoint of $\overline{T}_{\mathcal{P}}$ *)
      $V'_n := V'_{n+1}$;
      $V'_{n+1} := \overline{T}_{LP}(V'_n)$;
    until $V'_{n+1} = V'_n$ or $V'_{n+1} \cap$ `not` $(V'_{n+1}) \neq \emptyset$;
    if $V'_{n+1} \cap$ `not` $(V'_{n+1}) = \emptyset$ (* V'_{n+1} is consistent *)
      $\wedge$ (**not** *strong_violated*($\mathcal{P}$,V'_{n+1}))
      $\wedge$ *compute_objective*($\mathcal{P}$,V'_{n+1}) $\leq$ *minCost*
      then $Compute\_Preferred1(V'_{n+1}, minCost)$;
  end_for
end_procedure

Figure 4.7: A procedure which computes all preferred models, once the minimal cost value is known

this objective function value. Also we output preferred models and not only currently best models.

The complexity of the algorithm stays within polynomial space and single exponential time in this case since we just double the computation time in the worst case, whereas the space requirement does not increase.

Chapter 5

Architecture/System Description

The algorithm we described is going to be integrated into an existing experimental system, developed at Institut für Informationssysteme, Abteilung für Datenbanken und Expertensysteme at Technische Universität Wien, in the course of an ongoing project sponsored by FWF.

The system, called `dlv`, is based on the algorithm described in [LRS97], with several additional optimisations [ELM+97a, ELM+97b, ELM+97c, ELM+98, CEF+97].

5.1 Interface

The interface to the system is basically command-line oriented. In addition a Graphical User Interface (GUI) has been implemented, with which all relevant functions of the various frontends (see below) can be controlled.

5.2 Frontends

5.2.1 Native Extended Datalog

This is the heart of the system. It can parse programs of $\Pi_{\text{DATALOG}^{\neg,not,\vee,s}}$ and compute their consistent answer sets, resp. stable models and is used by all other frontends, as indicated by Figure 5.1 on the next page. Most other frontends implement a translation from some language to $\text{DATALOG}^{\neg,not,\vee,s}$. If explicitly negated atoms occur, they are transformed as described in Section 2.6.6.

This part of the system has already been updated to parse programs of $\Pi_{\text{DATALOG}^{\neg,not,\vee,w}}$. So in the future, further frontends which enable the user to specify optimisation problems and translate these problems into a suitable $\text{DATALOG}^{\neg,not,\vee,w}$ program are feasible.

5.2.2 Diagnosis

This frontend currently supports several versions of *LPAP*s. It is an ideal candidate to be extended to handle those types of abduction which are described

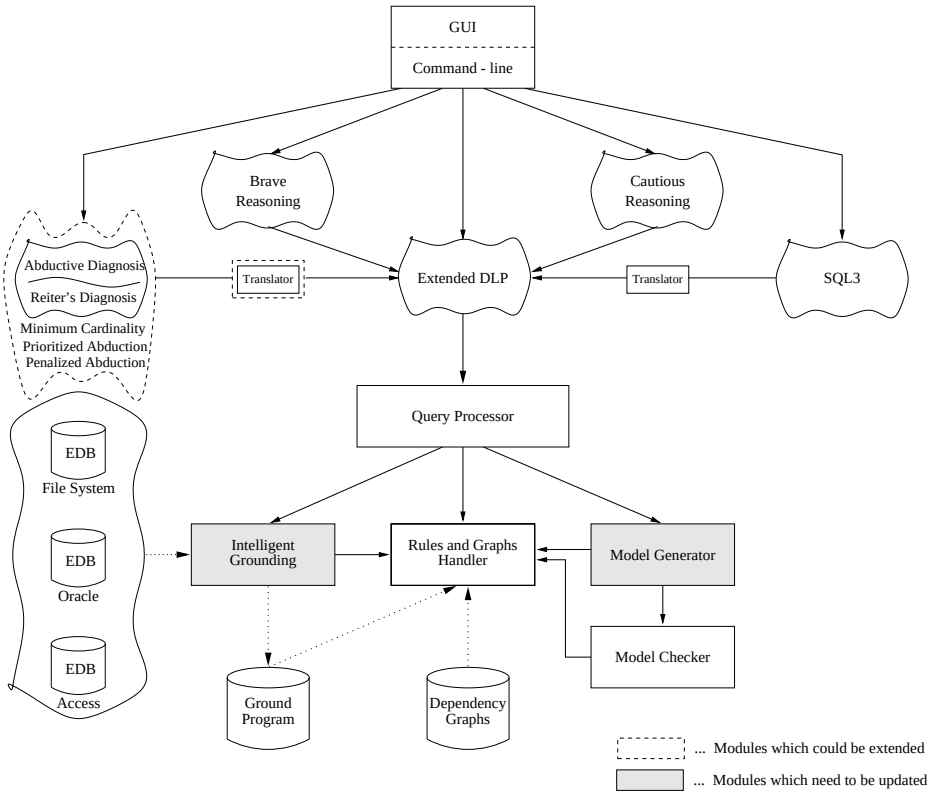


Figure 5.1: Structure of the `dlv` system

in Section 3.1.

5.2.3 SQL3

This frontend translates a subset of SQL3 statements into DATALOG$^{not,\vee}$, most notably those constructs which enable the user to express recursive queries. However, the SQL3 standard is not official yet, and our system is probably the first one which can actually handle this type of queries.

Currently we do not see any potential gain for this frontend by the addition of weak constraints.

5.2.4 Brave and Cautious Reasoning

This is a way to actually pose queries to datalog programs. In the case of the native datalog frontend, models are computed – here we ask whether some specific literal, or conjunction, in general, is true in at least one model (*Brave Reasoning*), or in all models (*Cautious Reasoning*).

The extension of these reasoning concepts to preferred models is straightforward and has not been marked explicitly in Figure 5.1 on the preceding page.

5.3 Query Processor, Grounding, and Handling of Rules

The Query Processor is the module which coordinates the computations and states of the various low-level modules. It is going to be updated to be able to deal with DATALOG$^{\neg,not,\vee,w}$ programs. This update is very straightforward and thus not indicated in Figure 5.1 on the page before.

The system has an Intelligent Grounding Module, which reduces the size of the program to be considered by the other modules considerably during the grounding phase. Actually, this module is able to compute the stable models of some easy classes of programs.

Of course, also weak constraints have to be grounded. We will even extend the language of DATALOG$^{\neg,not,\vee,w}$ by allowing the specification of layers and weights to be variable in the programs rather than fixed numbers as in this thesis, which must be handled accordingly by the grounding phase. In this light, the Intelligent Grounding will need a more sophisticated update.

The Rule Handler has to be adapted to incorporate the handling of Weak Constraints, but again this change is immediate.

5.4 Model Generator and Checker

These two modules correspond to the algorithm presented in [LRS97] and in this thesis. By *Model Generator* we refer to the generation of the computations (as in Definition 4.5.1), whereas by *Model Checker* we mean the check for unfounded-freeness.

As we have seen in Chapter 4, only the Model Generator is affected by the extension to weak constraints. The model checks do not have to be altered.

Chapter 6

Research Issues

An interesting question is on which classes of programs the algorithm described in this thesis terminates in polynomial time. It is known (cf. [LRS97]) that the class of head-cycle-free programs is one of these. Another class which exhibits this property is the class of choice programs. These are programs, where a special goal enforces some functional dependencies, thus effectively making a choice. In [SZ90] a translation from such programs to DATALOG under the Stable Model Semantics has been presented. Actually, the unstratified program for the Minimum Spanning Tree problem in Figure 3.16 on page 70 can be viewed as a choice program which has been transformed to DATALOG using the approach cited above.

Disjunction-free programs with even unstratified negation (that is, the dependency graph, in which positive and negative dependencies are represented, contains only cycles with an even number of negative edges) are known to be solvable in polynomial time, however we cannot guarantee that the presented algorithm is also within this time bound for these programs.

Concerning the extension by weak constraints defined in this thesis, we are particularly interested in programs for which one preferred model can be found in polynomial time by our algorithm.

Other issues include finding optimisations for the computation. In particular an approach similar to the partitioning into components could be promising. We might try to partition the weak constraints of a program according to the priority layers of their literals in order to be able to compute a preferred model sooner.

Also different search strategies should be evaluated, as there are lots of approaches in the literature most of which are very problem-specific. The challenge here is to generalise these concepts from these domains to a versatile system like `dlv` without losing their advantages.

List of Figures

Bibliography

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Bar96] Victor A. Bardadym. Computer-aided school and university timetabling: The new wave. In Burke and Ross [BR96], pages 22–45.

[BD95] Stefan Brass and Jürgen Dix. Characterizations of the disjunctive stable semantics by partial evaluation. In V. W. Marek, A. Nerode, and M. Truszczyński, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Third International Conference*, number 928 in LNAI. Springer, June 1995.

[BD97] Stefan Brass and Jürgen Dix. Characterizations of the disjunctive stable semantics by partial evaluation. *Journal of Logic Programming*, 32(3):207–228, 1997. Extended Abstract appeared as [BD95].

[BF91a] Nicole Bidoit and Christine Froidevaux. General logical databases and programs: Default logic semantics and stratification. *Information and Computation*, 91:15–54, 1991.

[BF91b] Nicole Bidoit and Christine Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78:85–112, 1991.

[BG94] C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[BLR97a] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, Grado, Italy, June 1997.

[BLR97b] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR '97)*, Dagstuhl, Germany, July 1997.

[BLR98] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. 1998. Submitted to IEEE Transactions on Knowledge and Data Engineering.

[Bök82] Dieter Bökemann. *Theorie der Raumplanung*. Oldenbourg Verlag, München, 1982.

[BR96] Edmund Burke and Peter Ross, editors. *Practice and Theory of Automated Timetabling, First International Conference 1995*, number 1153 in LNCS. Springer, 1996.

[CDM98] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Metaheuristics for high school timetabling. *Computational Optimization and Applications*, 9(3):275–298, 1998.

[CEF+97] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The `dlv` System: Model Generator and Application Frontends. In *Proceedings of the 12th Workshop on Logic Programming (WLP '97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer, 1990.

[Chr75] Nicos Christofides. *Graph Theory An Algorithmic Approach*. Academic Press, Inc., 1975.

[CK96] Tim B. Cooper and Jeffrey H. Kingston. The complexity of timetable construction problems. In Burke and Ross [BR96], pages 283–295.

[Dix95] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proc. of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[EG95] Thomas Eiter and Georg Gottlob. The Complexity of Logic-Based Abduction. *Journal of the ACM*, 42(1):3–42, January 1995.

[EGL97] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction From Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.

[ELM+97a] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '97)*, number 1265 in Lecture Notes in AI (LNAI), Berlin, 1997. Springer.

[ELM+97b] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The Architecture of a Disjunctive Deductive Database System. In *Proceedings Joint Conference on Declarative*

Programming (APP IA-GULP-PRODE '97), pages 141–151, June 1997.

[ELM+97c] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. Projektbericht: Ein Nichtmonotones Disjunktives Datenbanksystem. *ÖGAI Journal (J. of the Austrian Society for AI)*, 16(2):6–11, 1997. In German. English Title: Project Report: A Nonmonotonic Disjunctive Deductive Database System.

[ELM+98] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, 1998. Forthcoming.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[KKT93] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 1993.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1984.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, Berlin, 1987. second edition.

[LMR92] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[LRS97] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, June 1997.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pfe96] Gerald Pfeifer. Disjunctive Datalog — An Implementation by Resolution. Master's thesis, Institut für Informationssysteme, Technische Universität Wien, 1996.

[Prz90] Teodor C. Przymusinski. Well-founded Semantics Coincides with Three-valued Stable Semantics. *Fundamenta Informaticae*, 13:445–464, 1990.

[Prz91] Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[Prz95] T. Przymusinski. Static Semantics for Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.

[Ros90] K.A. Ross. The Well Founded Semantics for Disjunctive Logic Programs. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 385–402. Elsevier Science Publishers B. V., 1990.

[Sch95] A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567 1995, Computer Science/Department of Software Technology, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988. second edition.

[Smu78] Raymond Smullyan. *What Is the Name of This Book?* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1978.

[SZ90] Domenico Saccà and Carlo Zaniolo. Stable models and nondeterminism in logic programs with negation, 1990.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1989.

[vRS91] A. van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.