Answer Set Programming: A Primer[⋆]

Thomas Eiter[1], Giovambattista Ianni[2], and Thomas Krennwallner[1]

[1] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,tkren}@kr.tuwien.ac.at
[2] Dipartimento di Matematica, Universitá della Calabria, I-87036 Rende (CS), Italy
ianni@mat.unical.it

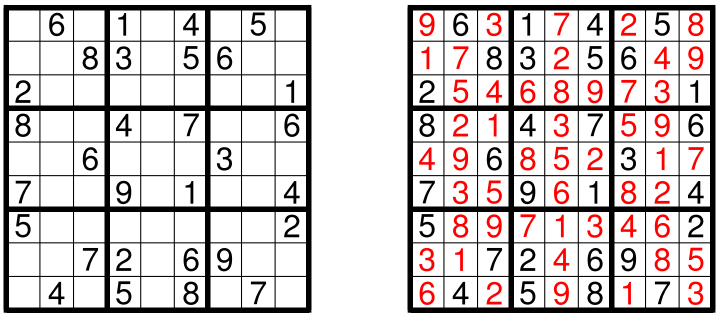
Abstract. Answer Set Programming (ASP) is a declarative problem solving paradigm, rooted in Logic Programming and Nonmonotonic Reasoning, which has been gaining increasing attention during the last years. This article is a gentle introduction to the subject; it starts with motivation and follows the historical development of the challenge of defining a semantics for logic programs with negation. It looks into positive programs over stratified programs to arbitrary programs, and then proceeds to extensions with two kinds of negation (named weak and strong negation), and disjunction in rule heads. The second part then considers the ASP paradigm itself, and describes the basic idea. It shows some programming techniques and briefly overviews Answer Set solvers. The third part is devoted to ASP in the context of the Semantic Web, presenting some formalisms and mentioning some applications in this area. The article concludes with issues of current and future ASP research.

1 Introduction

Over the the last years, *Answer Set Programming* (ASP) [1–5] has emerged as a declarative problem solving paradigm that has its roots in Logic Programming and Nonmonotonic Reasoning. This particular way of programming, in a language which is sometimes called AnsProlog (or simply A-Prolog) [6, 7], is well-suited for modeling and (automatically) solving problems which involve common sense reasoning: it has been fruitfully applied to a range of applications (for more details, see Section 6). A number of extensions of the ASP core language, which goes back to the seminal paper by Gelfond and Lifschitz [8], have been developed (resulting in an AnsProlog* language family). These extensions aim at increasing the expressiveness of the formalisms and/or providing convenient constructs for application-specific problem representation; see, e.g., [9] for an account of such extensions.

The basic idea of ASP is to describe problem specifications by means of a nonmonotonic logic program: solutions to instances of such a problem will be represented

[⋆] This is an updated version of the paper (with corrections) that appeared in the Proceedings of the 5th International Reasoning Web Summer School 2009, volume 5689 of LNCS, Springer. This work has been supported by the Austrian Science Fund (FWF) project P20840 & P20841, the EC ICT Integrated Project Ontorule (FP7 231875), and the Italian National Project Interlink II04CG8AGG.



	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Fig. 1. Sudoku puzzle (left) and solution (right)

by the intended models of the program (the so-called *answer sets*, or *stable models*) at hand. Rules and constraints, which *describe* the problem and its possible solutions rather than a concrete algorithm, are basic elements of such programs.

Such a problem encoding can be then fed into an answer set (AS) solver, which computes some or multiple answer set(s) of the program, from which the solutions of the problem can easily be read off.

As a simple motivating example, consider the popular Sudoku game.[3]

Example 1 (Sudoku). In its original version, a Sudoku consists of a tableau that has 81 cells arranged in a grid, which is divided into nine sub-tableaux (the blocks or regions) of equal size having nine fields each. The initial game setup has some of the entries filled with numbers between 1 and 9 (see Figure 1, left, for an example).

The question is now whether the tableau can be completed in a way such that each row and each column shows every digits from 1 to 9 exactly once, and moreover that also each block has this property. An example for a completed Sudoku grid is on the right in Figure 1, which is the unique solution to the initial puzzle on the left.[4]

In general, the problem of solving Sudoku tables automatically appears to be non-trivial: in principle, one can devise a brute force algorithm that considers all possible assignments and checks whether the solution constraint is satisfied. For a versatile programmer, it is not difficult to write a program in her favorite programming language, be it Java, C++, or some other language, to compute and print a solution to instances of this problem.

In this traditional, time-consuming approach, a human programmer receives an informal specification of the problem at hand, such as the Sudoku above, and manually converts it into imperative code that is able to solve instances of the problem. However, one might conceive to tackle this issue from a completely different perspective.

For instance, one can think of having access to appropriate means for *directly describing* the problem at hand in a declarative specification. This specification, if properly polished from ambiguities of natural language and expressed in a proper syntax,

[3] This game has nowadays worldwide popularity, and world and national championships are held in big tournaments each year across Europe.

[4] To date, many variants of Sudoku emerged, like, e.g., color-Sudoku, Samurai-Sudoku, etc.

would be not much different in its meaning from the formulation of Sudoku of our example. Also, such a specification could be automatically *executed*, in the sense that some computational engine *takes this specification as input*, together with a problem instance, and then *produces a solution as output*. In such a vision, the human programmer would switch her focus from *how to solve a problem* to *how to state a problem*, which is a much easier and faster task. [5]

The Prolog language, and its extensions conceived for handling constraints, can be seen at a first glance as tools for such "declarative problem solving." Prolog is indeed well-suited for this particular case.

There are however aspects which make the suitability of Prolog (with respect to AnsProlog) less apparent. Among such aspects, there is the fact that many common problems require preference handling (that is, the possibility to describe which solutions are preferred to others with respect to some "quality" criterion), and to properly deal with incomplete information (that is, the ability to properly complete missing information with default assumptions, or with assumptions of falsity, or with using some notion of undefinedness). The next example shows the impact of such aspects.

Example 2 (Social Dinner Example). Imagine the organizers of this course planning a fancy dinner for the course participants. To make the event a great success, the organizers decide to ask the attendees to declare their personal wine *preferences*. Soon, the organizers become aware of the fact that there is no wine, which satisfies all of the participant preferences. Thus, they aim at automatically finding the *cheapest* selection of bottles such that any attendee can have her preferred wine at the dinner. This solution should take into account that people usually like wine from their home country, but may not like to drink it abroad.

The organizers quickly realize that several, different specification tools are needed to accomplish this task : in this example, it is more difficult to model the scenario appropriately, and in particular to adequately represent and handle the emerging preferences, priorities, and defaults in absence of complete information, along with conflicts that emerge from them.

This situation motivates a general-purpose approach for modeling and solving also many other problems, which take among others the following aspects into account:

- Possibility of integrating diverse domains;
- Spatial and temporal reasoning (here, the notorious *Frame Problem* is challenging);
- Possibility of modeling constraints;
- Reasoning with incomplete information; and
- Possibility of modeling preferences and priority.

The ASP paradigm has been proposed as a possible solution about ten years ago, as the underlying non-monotonic logic programs are well-positioned to cover these aspects. In the following, we shall briefly look at the roots of ASP and at the relationship of ASP to Prolog, before we turn to the technical preliminaries.

[5] A specification of the Sudoku problem expressed in AnsProlog is reported in Appendix A.

1.1 Roots of ASP

ASP is strongly rooted in the area of Knowledge Representation and Reasoning, and therein in logic programming. However, rather than to foster a general problem solving paradigm, the roots of ASP are in formalisms that aimed at particular representation and reasoning tasks, such as

- modeling an agent's belief sets,
- commonsense reasoning,
- defeasible inferences, and
- preferences and priority.

To this end, many logic-based formalisms for knowledge representation have been developed. As an inherent feature, these formalisms are *nonmonotonic*, that is, they have the property that a growing stock of beliefs may invalidate part of the conclusions that were previously drawn in lack of complete knowledge.

The formalisms, which address above objectives, were motivated by the vision of John McCarthy and other pioneers in AI: logic is an ideal tool for representing and processing knowledge. Oversimplified, the idea can be explained as follows:

- declare knowledge about a "world" of interest by logical sentences;
- more precisely, one should use predicate logic for knowledge representation;
- derive new (implicit) knowledge by an automated inference procedure.

For example, the simple knowledge base

$$K = \{human(socrates), \forall x(human(x) \Rightarrow mortal(x))\}$$

might informally express the fact that Socrates is human and the rules that all humans are mortal in predicate logic; from this knowledge base, we can derive the fact $mortal(socrates)$ using deductive inference procedures, using different methods; *logical calculi* allow us to derive inferences in a purely syntactic way by manipulating formulas according to *inference rules*. In our example, we can infer $mortal(socrates)$ e.g. from the rules of Modus Ponens: $\frac{\phi,\ \phi \Rightarrow \psi}{\psi}$, and Specialisation: $\frac{\forall x(\phi(x)),\ \text{individual } c}{\phi(c)}$.

Loosely speaking, with such a calculus the derivation of new knowledge boils down to simply a search for a proof in terms of inference rule applications from a set of starting axioms. However, a big problem is that, for predicate logic in general, the existence of such a proof is undecidable (as shown in the 1930s by Church) and thus the dream of a "calculus ratiocinator" (or a "thinking machine") in the sense of Leibniz, can not be materialized in general. The insight was that knowledge processing needs control (which inference rule(s) should be applied?) and that often knowledge can be formulated in terms of rules and facts.

1.2 Prolog

After Robinson's breakthrough with the *Resolution principle* in automated theorem proving, in the early 1970s logic programming has been developed as a new knowledge based problem solving paradigm.

Prolog ("Programming in Logic") emerged as a general purpose programming language, whose guiding principle has been popularized by Kowalski's [10] slogan:

ALGORITHM = LOGIC + CONTROL

where the LOGIC on the right hand side stands for the problem specific knowledge, and the CONTROL for the "processing" of that knowledge in a suitable inference procedure.

Computing with Prolog programs is done using a predicate language, featuring the following:

– Terms are used to access objects, where constants stand for individuals (e.g., joe) and variables (e.g., X) for unknown individuals, and function symbols (like in $father(joe)$) are available.
– Terms are used to model basic data structures, like records, e.g $name(joe, doe)$.
– Instead of iteration, there is extensive use of recursion.
– In connection with this, the list constructor $[\cdot|\cdot]$ can be used, which also allows to define higher-order objects (like sets).
– Solutions are obtained via queries (goals) that are posed to the program, where formal proofs provide answers. They build on
  - SLD-resolution, a special variant of the resolution calculus, and
  - unification, as the basic mechanism to manipulate data structures.

The following is a simple Prolog program, familiar from most beginner courses in Prolog, for appending two lists and for reverting a list, respectively.

$$append([\,], X, X). \tag{1}$$
$$append([X|Y], Z, [X|T]) \leftarrow append(Y, Z, T). \tag{2}$$

$$reverse([\,], [\,]). \tag{3}$$
$$reverse([X|Y], Z) \leftarrow append(U, [X], Z), reverse(Y, U). \tag{4}$$

The above program recursively defines the predicates $append(X, Y, Z)$ and $reverse(X, Y)$, where the latter is defined in terms of the former. By posing a query against the program, we then can reverse lists. E.g., to reverse the list $[a, b, c]$, we can pose the query $?\!-\, reverse([a, b, c], X)$. A proof of the query yields a substitution: $X = [c, b, a]$, which then gives an answer. One can also pose queries that allow to reason backwards from the output to the input (which is not possible in imperative programming). E.g., if we pose $? - reverse([a|X], [b, a])$. the answer substitution $X = b$ tells us that the "input" for the output $[b, a]$ must consist of $[a, b]$.

In principal, above way of programming is a major step forward to our goal of writing programs in a declarative way, but an important point is that it may make a difference how and in which order the clauses of a Prolog programs are given. Although logically equivalent in terms of predicate calculus, if we replace rule (4) above by

$$reverse([X|Y], Z) \leftarrow reverse(Y, U), append(U, [X], Z). \tag{5}$$

and then ask $? - reverse([a|X], [b, c, d, b])$, the evaluation does not terminate (or is stopped because resources are exhausted, with no result). Similar behavior may be found if rules in a program are moved around. This is not a bug of Prolog but intrinsic in its highly efficient inference algorithm (which is sound but incomplete). Operators like the cut (which allow to prune the search space further, at the risk of losing solutions if done improperly), allow the fine control of the evaluation algorithm.

This example raises the legitimate question whether programming in Prolog is truly declarative. In fact, if one keeps in mind the goal of having specifications in which a problem is *declared*, without knowledge on how this declaration will be processed, it is desirable, as far as termination and finding of a solution is concerned, that

- the order of program rules does not matter, and that
- the order of subgoals in a rule body does not matter.

This calls for "pure" declarative programming, in which we (possibly) trade the efficiency of problem solving for strict declarativity of the formalism. The major exponent of this "pure" declarative programming paradigm is the stable model semantics of logic programs, which will be introduced in the sections below.

The stable model semantics is often confused with ASP. Indeed the semantics of the latter has been specified in terms of the former in the seminal paper [11].

The success of ASP is based on the easy usage of ASP as a modeling language, and on the variety of sophisticated algorithms and techniques for evaluating A-Prolog programs, which originated from research on computational complexity of reasoning tasks for such programs. The complexity of ASP reasoning is well understood, and a detailed picture of it and its major extensions can be found in [12]. Advanced AS solvers such as Smodels, DLV, GnT, Cmodels, Clasp, or ASSAT (see [13]), are able to deal with large problem instances; demonstration efforts of the potential of ASP are made at the AS solver competition [14] which takes place at the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR) since 2007.

1.3 Structure of the Article

The rest of this article is divided into three parts as follows. The first part introduces the stable models semantics of normal logic programs and the answer set semantics of extended logic programs, as well as of extensions thereof. Concepts and notions are given following a historical timeline, which incidentally coincides with the development of increasingly expressive specification languages based on rules. We first recall the least model semantics of Horn logic programming (Section 2) and then turn to the issue of negation in logic programs (Section 3). Then, we consider stratified logic programs, for which the perfect model semantics is the canonical semantics (Section 3.1). We then present the stable model semantics of normal logic programs (Section 4) which coincides with the perfect model semantics on stratified programs (and thus generalizes it). After that, we proceed with some extensions in Section 5; in particular, with constraints, with strong negation—where we arrive at the notion of answer sets—and with disjunctive rule heads.

The second part then considers the ASP paradigm itself. It describes the general idea and shows some ASP programming techniques (Section 6). Furthermore, it overviews AS solvers and their general architecture and implementation principles (Section 7); as an example, we briefly present the AS solver DLV.

The third part is devoted to ASP in the context of the Semantic Web, presenting some formalisms and mentioning some applications in this area (Section 8). The article concludes with issues of current and future ASP research.[6]

2 Horn Logic Programming

We will consider logic programs built from simple constituent blocks, which correspond syntactically to the language of predicate calculus. We will have *constants*, which represent individuals of the domain of discourse, like sarah, $\mathit{chicago}$, and 2. They will be represented with lowercase starting letter, or with natural numbers. Variables, like X, City, Name, denote an individual variable, and are written with uppercase starting letter. Also, one might form *functional terms* combining constants, functions symbols and variables such as in $\mathit{next}(a, Y)$, where next is a binary function symbol.

In some sense, variables and constants can be seen as subjects and objects participating to the scenario we are modeling, which can be tied together through *predicates*, like $\mathit{hasName}$ and link. Predicates relate with variables and constants through *atoms*, like $\mathit{link}(\mathit{chicago}, \mathit{paris})$ or $\mathit{hasName}(C, \mathit{sarah})$. Note that the former atom has no variables in it (it is *ground*), while the latter is *nonground*. Functional terms are syntactically equivalent to atoms, yet they have different meaning. A (ground) atom is connected to its truth value and acts as a propositional variable: for instance, $\mathit{hub}(\mathit{rome})$ might be true or false in the sense that rome might be a hub or not; on the other hand, $\mathit{father}(\mathit{gb})$, when seen as a functional term, denotes an individual of our domain of discourse ("the father of gb"), for which truth or falsity makes no sense in general.

On top of these simple notions we use the idea of *rules*. Rules are grouped in sets that we will call (*logic*) *programs*.

We will start with a class of logic programs featuring the simplest form of a rule.

2.1 Positive Logic Programs

Definition 1 (Positive Logic Program). *A* positive logic program P *is a finite set of clauses (rules) in the form*

$$a \leftarrow b_1, \ldots, b_m \ , \qquad (6)$$

where a, b_1, ..., b_m are atoms of a first-order language L. We call a the head *of the rule, while $b_1, \ldots, b_m$ represents the rule's* body. *A* fact *is a rule with empty body such as $a \leftarrow$, denoted for short as a.*

To give an intuition of the meaning of a rule, a reader familiar with imperative programming languages might interpret this construct as an abstraction of the **if** ... **then** ... construct common in traditional programming languages, to which, as it has been illustrated, Modus Ponens might apply. For a reader familiar with first order logic, rules can be seen as material implications restricted to Horn clauses, where $A \leftarrow B$ is read as $B \supset A$ or $B \rightarrow A$.

[6] The accompanying slides are available at `http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-lecture.zip`.

For instance, the rule

$$connected(cagliari) \leftarrow hub(rome), link(rome, cagliari)$$

might be "procedurally" read as "if Rome is a hub, and there is a link between Rome and Cagliari, then Cagliari is a connected airport," or, when seen as a first-order Horn clause in predicate logic, the same rule can be interpreted as "in any possible scenario in which Rome is a hub and there is a link between Rome and Cagliari, it is the case that Cagliari is connected."

However, we will observe later that rules in declarative logic programming do not strictly correspond to the procedural scheme of imperative languages, nor to material implication. Nevertheless, they are *declarative* constructs, and we make this more clear later in this section.

The above example rule is ground, but logic programs might contain nonground rules like

$$connected(X) \leftarrow hub(Y), link(Y, X) \ \ ,$$

which can be read as the universally quantified clause $\forall X, Y\ hub(Y) \wedge link(Y, X) \supset connected(X)$. Importantly, one must distinguish between the imperative and logical reading of clauses: a variable X in imperative programming associates a single value to it and stands for a named storage cell, whereas X reads as "any X having a certain property" in the logical interpretation of clauses.

We can also think of a logic program as a description of a scenario, in which certain assertions, either specific and related to certain individuals (that is, ground), or general (that is, nonground, or partially ground), must hold.

The following definitions clarify this intuition.

Definition 2 (Herbrand Universe, Base, Interpretation). *Given a logic program P, the* Herbrand universe *of P, $HU(P)$, is the set of all terms which can be formed from constants and functions symbols in P (resp. the vocabulary of L, if explicitly known).*

The Herbrand base *of P, $HB(P)$, is the set of all ground atoms which can be formed from predicates occurring in P and the terms in $HU(P)$. A (Herbrand)* interpretation *is an interpretation I over $HU(P)$, that is, I as subset of $HB(P)$.*

An interpretation can be seen as a set denoting which ground atoms are true in a given scenario.

Example 3. Assume the following program P_1 is given:

$$\begin{aligned}
h(0,0).\\
t(a,b,r).\\
p(0,0,b).\\
p(f(X),Y,Z) \leftarrow p(X,Y,Z'), h(X,Y), t(Z,Z',r).\\
h(f(X),f(Y)) \leftarrow p(X,Y,Z'), h(X,Y), t(Z,Z',r).
\end{aligned}$$

The unique function symbol appearing in P_1 is f, and the constant symbols in P_1 are r, a, b, and 0. Thus, $HU(P_1) = \{0, a, b, r, f(0), f(f(0)), \ldots, f(a), f(f(a)), \ldots\}$, which represents the (infinite) set of individuals possibly involved in P_1.

The Herbrand base is $HB(P_1) = \{p(0,0,0), p(a,a,a), \ldots, h(0,0), \ldots, t(0,0,0), t(a,a,a), \ldots\}$, and represents the set of all possible ground assertions which might hold.

Some possible Herbrand interpretations are

- $I_1 = \emptyset$,
- $I_2 = HB(P_1)$,
- $I_3 = \{h(0,0), t(a,b,r), p(0,0,b)\}$,

and so on. An interesting question is which scenarios (interpretations) are compatible with P_1. For instance, the interpretation $\{h(0,0), t(a,b,r)\}$ is contradicting P_1, which follows from the simple expectation that, in virtue of the last fact in P_1, also $p(0,0,b)$ should be considered true.

Definition 3. *A* ground instance *of a clause C of the form* (6) *is any clause C' obtained from C by applying a substitution*

$$\theta\colon\ Var(C) \rightarrow HU(P)$$

to the variables in C, denoted as $Var(C)$. For any clause C, we denote by $grnd(C)$ the set of all possible ground instances of C, and for any program P we let $grnd(P) = \bigcup_{C \in P} grnd(C)$ (called the grounding of *P).*

Intuitively, $grnd(C)$ allows for the materialization of the universal quantification of variables appearing in C. Roughly speaking, C is a shortcut denoting a set of clauses $grnd(C)$. The range of each variable appearing in C is given by the set of terms appearing in the Herbrand universe.

Example 4. Consider the following program P_2:

$$\begin{aligned} p(f(X),Y,Z) \leftarrow p(X,Y,Z'), h(X,Y), t(Z,Z',r). \\ h(0,0). \end{aligned}$$

The ground instances of the first rule in P_2 are

$$\begin{aligned} p(f(0),0,0) &\leftarrow p(0,0,0), h(0,0), t(0,0,r). \\ &\vdots \\ p(f(0),r,0) &\leftarrow p(0,r,0), h(0,r), t(0,0,r). \\ &\vdots \\ p(f(r),r,r) &\leftarrow p(r,r,r), h(r,r), t(r,r,r). \end{aligned}$$

7

Definition 4. *Let I be an interpretation. Then I is a* model *of*

[7] Note that in practice most of the ground rules appearing in $grnd(C)$ for given C might have no actual impact when computing the *least model* of C as defined next.

– *a ground (variable-free) clause $C = a \leftarrow b_1, \ldots, b_m$, denoted $I \models C$, if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $a \in I$;*
– *a clause C, denoted $I \models C$, if $I \models C'$ for every $C' \in grnd(C)$;*
– *a program P, denoted $I \models P$, if $I \models C$ for every clause $C \in P$.*

Intuitively, a model of P is an interpretation which is compatible with assertions appearing in P.

Example 5. Reconsider the program P_2 in Example 4. Note that $I_1 = \emptyset$ is not a model of P_2 (the fact $h(0,0)$ is not true in I_1), while $I_2 = HB(P_2)$ is a model; indeed, for every program P it clearly holds that $HB(P)$ is a model of P. However, $I_3 = \{h(0,0), t(0,0,r), p(0,0,0)\}$ is not a model of P_2, since the first rule would require $p(f(0),0,0) \in I_3$.

2.2 Minimal Model Semantics

In general, there are multiple "compatible" interpretations of a program P, that is, there can be multiple interpretations, which are models of P. Some of them are however trivial, e.g., think of I_2 in the previous example w.r.t. P_2, or they convey information which is not encoded in P_2. For instance, $I_4 = I_3 \cup \{p(f(0),0,0), h(r,r)\}$ is a model of P_2. There is however no evidence that $h(r,r)$ should be true according to P_2: indeed we might remove it from I_4, obtaining a smaller model $I_5 = I_3 \cup \{p(f(0),0,0)\}$.

On the other hand, we cannot remove $p(f(0),0,0)$ from I_5 since the first rule of the program would not be satisfied. In other words, $p(f(0),0,0)$ is an atom which has to be *necessarily* true in the scenario described by P_2, while this is not the case for $h(r,r)$.

One might ask at this point whether there exists a particular *canonical model* for a program which contains only the atoms which are necessarily true according to P. This notion of "necessity" is commonly called *foundedness*.

Example 6. Consider the small program P_3

$$a \leftarrow b. \quad b \leftarrow c. \quad c.$$

The truth of atom a in the model $I = \{a,b,c\}$ is "founded." Intuitively, c must appear in any model of P_3, which implies that also b and then a are necessarily true.

Given the program P_4

$$a \leftarrow b. \quad b \leftarrow a. \quad c.$$

we obtain that the truth of atom a in model $I = \{a,b,c\}$ is not founded. In other words, there is no necessity of a appearing in a model. Indeed, $I' = \{c\}$ is also a model.

The above intuition can be translated into a formal semantics, which prefers models having as few true facts as is possible.

Definition 5. *A model I of a program P is* minimal*, if there exists no model J of P such that $J \subset I$.*

Theorem 1. *Every positive logic program P has a single minimal model (called the* least model*), denoted $LM(P)$.*

This is entailed by the following property:

Proposition 1. *If I and J are models of P, then also $I \cap J$ is a model of P.*

Example 7. For $P_3 = \{a \leftarrow b. \quad b \leftarrow c. \quad c.\}$, we have $LM(P_3) = \{a, b, c\}$. For $P_4 = \{a \leftarrow b. \quad b \leftarrow a. \quad c.\}$, we get the least model $LM(P_4) = \{c\}$

For program P_1 above, we have

$$LM(P_1) = \{h(0,0), t(a,b,r), p(0,0,b), p(f(0),0,a), h(f(0),f(0))\} \ .$$

Computation of the Least Model. A natural question is, how we can compute the least model $LM(P)$ of a program P.

By means of the *immediate consequence operator*, one can obtain $LM(P)$ through an iterative process. Let $T_P \colon 2^{HB(P)} \to 2^{HB(P)}$ be an operator defined as

$$T_P(I) = \left\{ a \;\middle|\; \begin{array}{l} \text{there exists some } a \leftarrow b_1, \ldots, b_m \\ \text{in } grnd(P) \text{ such that } \{b_1, \ldots, b_m\} \subseteq I \end{array} \right\} \ .$$

We define $T_P^0 = \emptyset$, and $T_P^{i+1} = T_P(T_P^i)$ for $i \geq 0$.

Theorem 2. *T_P has a least fixpoint, lfp(T_P), and the sequence $\langle T_P^i \rangle$, $i \geq 0$, converges to it, i.e., lfp$(T_P) = LM(P)$.*

The above result can be proved by means of the fixpoint theorems of Knaster-Tarski and of Kleene given in Appendix B. The second part of the theorem is easily shown by observing that *lfp*(T_P) is a model of P and no smaller model exists.

Example 8. The immediate consequence operator captures the idea that if all the atoms in a rule r body are founded, then also the head of r must be founded.

For instance, for $P_3 = \{a \leftarrow b. \quad b \leftarrow c. \quad c.\}$, we have

$$T_{P_3}^0 = \{\}, \quad T_{P_3}^1 = \{c\}, \quad T_{P_3}^2 = \{c, b\}, \quad T_{P_3}^3 = \{c, b, a\}, \quad T_{P_3}^4 = T_{P_3}^3 \ .$$

Hence, *lfp*$(T_{P_3}) = \{c, b, a\}$. For $P_4 = \{a \leftarrow b. \quad b \leftarrow a. \quad c.\}$, we have

$$T_{P_4}^0 = \{\}, \quad T_{P_4}^1 = \{c\}, \quad T_{P_4}^2 = T_{P_4}^1 \ .$$

Hence *lfp*$(T_{P_4}) = \{c\}$.

For program P_1 above, we have

$$\begin{aligned} T_{P_1}^0 &= \emptyset, \\ T_{P_1}^1 &= \{h(0,0), t(a,b,r), p(0,0,b)\} \\ T_{P_1}^2 &= \{h(0,0), t(a,b,r), p(0,0,b), p(f(0),0,a), h(f(0),f(0))\} \\ T_{P_1}^3 &= T_{P_1}^2. \end{aligned}$$

3 Negation in Logic Programs

Positive logic programs allow for declarative modeling of a variety of problems. However, it turns out that many situations require a construct which model the intuitive notion of negation. Negation is a natural linguistic concept and happens to be extensively required when natural problems have to be modeled declaratively. For instance, given the rule

$$\mathit{connected}(X) \leftarrow \mathit{hub}(Y), \mathit{link}(Y, X) \ \ ,$$

which defines airports connected to at least one hub airport, one might think of defining airports which are *not* connected to any hub. This can be modeled intuitively by put the not modifier in front of atoms, and considering the rule

$$\mathit{badlyConnected}(X) \leftarrow \mathrm{not}\ \mathit{connected}(X) \ \ .$$

We will define *normal logic programs* as a set of clauses having the form

$$a \leftarrow b_1, \ldots, b_m, \mathrm{not}\ c_1, \ldots, \mathrm{not}\ c_n \qquad (n, m \geq 0) \tag{7}$$

where a and all b_i, c_j are atoms in a first-order language L. Note that rule bodies now include expressions which we call *(default) negated literals* $\mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_l$, which consist of atoms c_i preceded by the negation modifier not. Accordingly, the atoms $b_1, \ldots, b_k$ are called *positive literals*.

Intuitively, a ground literal corresponds to a propositional variable as it was the case for atoms: a negated literal has a truth value which is opposite to its corresponding positive literal. For instance, if $\mathit{hub}(\mathit{rome})$ is true, then $\mathrm{not}\ \mathit{hub}(\mathit{rome})$ is false.

Once negated literals are syntactically defined, one can think of a proper formal meaning for rules in which they appear. The Prolog semantics has been pragmatically and operationally extended from SLD to SLDNF in terms of *Negation as failure*: here, one considers as false a negated literal $\mathrm{not}\ a(\cdot)$, if the truth of its corresponding positive literal cannot be (finitely) proved through SLD resolution.

It is important to observe that negation in classical logic is different from negation in logic programming (cf. surveys [15, 16] and [17, 18] for more discussion).

Example 9. Consider the program P_5:

$$\begin{aligned} &\mathit{man}(\mathit{dilbert}). \\ \mathit{single}(X) &\leftarrow \mathit{man}(X), \mathrm{not}\ \mathit{husband}(X). \\ \mathit{husband}(X) &\leftarrow \mathit{fail}. \quad \text{\% fail = "false" in Prolog} \end{aligned}$$

Under Prolog semantics, if we ask the query

$$?-\mathit{single}(X).$$

we obtain as an answer

$$X = \mathit{dilbert} \ \ .$$

Intuitively, the answer is motivated by the fact that $\mathit{husband}(\mathit{dilbert})$ cannot be proved from P_5. For proving $\mathit{single}(\mathit{dilbert})$ using forward chaining, one can use the first rule of the program, in which it must be first shown that $man(dilbert)$ holds, and then that there is no proof for $husband(dilbert)$; indeed, there is no evidence that $husband(dilbert)$ is true in P_5.

Note however that this operational approach fails to give a satisfactory answer for programs like P_6

$$\begin{aligned} & man(dilbert). \\ single(X) &\leftarrow man(X), \text{not } husband(X). \\ husband(X) &\leftarrow man(X), \text{not } single(X). \end{aligned}$$

where $single(dilbert)$ and $husband(dilbert)$ are mutually dependent using negation. An SLD resolution algorithm would loop forever when trying to answer the query $single(X)$.

Approaches which give meaning to logic programs via a model theoretic definition (that is, providing an appropriate notion for a "best" model) are able to treat recursive definitions for positive programs properly, for which a unique minimal model exists. However, P_6 has two minimal Herbrand models

$$\begin{aligned} M_1 &= \{man(dilbert), single(dilbert)\}, \text{ and} \\ M_2 &= \{man(dilbert), husband(dilbert)\} \ . \end{aligned}$$

Both M_1 and M_2 satisfy P_6 and constitute a minimal set of necessarily true facts which are compatible with P_6. One thus may guess that introducing negation in logic programs induces a major problem regarding the meaning of normal logic programs.

The debate about the proper semantics for attributing meaning to negation in logic programs has been long lasting,[8] and provoked what we could call the *Great Logic Programming Schism*. Indeed, there are two philosophically very different approaches:

1. To keep the idea of defining a single model for a program, possibly including also problematic classes of programs with negation. This can be achieved by properly defining which *single* model should be selected among all classical models of a program. This line of research produced the notion of *perfect model* [21] which has been agreed being satisfactory for the class of so-called "stratified programs." For general normal problems, the most popular semantics is perhaps the one based on the *well-founded model* [22].
2. To identify a collection of *multiple* preferred models. This line of research abandons the "dogmatic" requirement of a single model and accepts the possibility of having multiple scenarios compatible with a given program. Note that, in general, for such multiple models approaches, we have a single model for positive and stratified programs which corresponds to the least and perfect model, respectively.

Answer Set Programming and its underlying *stable model semantics* is based on the latter methodology.

[8] The interested reader might refer to [15, 19, 20] for surveys about the matter.

3.1 Stratified Negation

As a first class of programs with negation we will consider *stratified programs* [23]. Stratified programs have the property that one can find an ordering for the evaluation of the rules in the program, such that the value of negative literals can be predetermined.

Intuitively, for evaluating the body of a rule containing $\mathrm{not}\, r(\boldsymbol{t})$, the value of the negative literal $r(\boldsymbol{t})$ should be known. This mimics the negation-as-failure approach as follows:

1. First evaluate $r(\boldsymbol{t})$;
2. if $r(\boldsymbol{t})$ is false, then $\mathrm{not}\ r(\boldsymbol{t})$ is true;
3. if $r(\boldsymbol{t})$ is true, then $\mathrm{not}\ r(\boldsymbol{t})$ is false and the rule is not applicable.

Example 10. We can evaluate the single rule program

$$boring(chess) \leftarrow \mathrm{not}\ interesting(chess)$$

according to this recipe: as $interesting(chess)$ clearly evaluates to false, the negated literal $\mathrm{not}\ interesting(chess)$ evaluates to true; hence, also $boring(chess)$ evaluates to true. This results in the Herbrand model $H = \{boring(chess)\}$ of P, which is the intuitive meaning of P.

Note however that this implicitly introduces a particular order of evaluation for rules and make specifications *procedural* more than declarative.

Dependency Graph. The above method makes only sense if there is no cyclic negation in programs. Otherwise, it is not possible to find an "evaluation ordering" for a program. The notion of dependency graph of programs captures this intuition.

Definition 6 (Dependency graph). *The dependency graph of a program P, $dep(P) = \langle V, E \rangle$, consists of*

- *a set of nodes V, which is defined as the set of all predicates p occurring in P, and*
- *a set of arcs E, which contains arcs of form $p \to q$ if and only if an atom with predicate name p is in the head of a rule $r \in P$ and the body of r contains a literal with predicate name q. If this literal is under negation, the edge will be marked with $\star$ ($p \to^\star q$).*

Example 11. Consider the following program P_7

$$\begin{aligned} man(dilbert).& \\ husband(X) \leftarrow man(X),\ married(X).& \\ single(X) \leftarrow man(X), \mathrm{not}\, husband(X).& \end{aligned}$$

and its dependency graph $dep(P_7)$ shown in Figure 2. The order of evaluation for negated predicates is built according to the following policy: If there is a path in $dep(P_7)$ from a predicate $p = p_0 \to p_1 \to p_2 \to \cdots \to p_{n-1} \to p_n$ to a predicate $q = p_n$, such that some $p_i \to p_{i+1}$ is marked with $\star$, then q must be evaluated prior to p. In this example we have a path $single \to^\star husband \to married$, thus both $husband$ and $married$ must be evaluated before $single$.

$$\begin{tikzcd}
husband \arrow[r] \arrow[dr] & married \\
single \arrow[u, "\star"] \arrow[r] & man
\end{tikzcd}$$

Fig. 2. $dep(P_7)$

Stratification. We formalize the notion of stratification as follows. Let $pred(R)$ denote the set of predicate names occurring in a set of rules R.

Definition 7 (Stratification). *A* stratification *of a set of rules P is a partitioning $\Sigma = \{S_i \mid i \in \{1,\dots,n\}\}$ of $pred(P)$ into n nonempty and pairwise disjoint sets of predicate names such that*

(a) if $p \in S_i$, $q \in S_j$, and $p \to q$ is in $dep(P)$ then $i \geq j$; and
(b) if $p \in S_i$, $q \in S_j$, and $p \to^\star q$ is in $dep(P)$ then $i > j$.

The sets $S_1,\dots,S_n$ are called the strata *of P w.r.t. Σ. A program P is called* stratified, *if it has some stratification Σ.*

Note that there are programs which are not stratified, such as P_6 above. The stratification Σ specifies an *evaluation order* for the predicates in a logic program. Here *evaluation* of a predicate p means to compute the set of true atoms that have p as predicate name. This sequential evaluation can be done by computing a series of *iterative least models*.

Definition 8. *Let P a logic program with a stratification $\Sigma = \{S_1,\dots,S_k\}$ of length $k \geq 1$. We define P_{S_i} as the subset of the rules of P which have a head atom whose predicate belongs to S_i, and $HB^\star(P_{S_i}) = \bigcup_{j \leq i}\{p(\mathbf{t}) \in HB(P) \mid p \in S_j\}$. We define the* iterative least models *$M_i \subseteq HB(P)$ with $i \in \{1,\dots,k\}$ by:*

(i) M_1 is the least model of P_{S_1};
(ii) if $i > 1$, then M_i is the least subset M of $HB(P)$ such that (a) M is a model of P_{S_i}, and (b) $M \cap HB^\star(P_{S_{i-1}}) = M_{i-1} \cap HB^\star(P_{S_{i-1}})$.

We denote by $M_{P,\Sigma}$ the iterative least model M_k.

Example 12. Consider again the program P_7:

$$\begin{aligned} &man(dilbert).\\ husband(X) \leftarrow\ & man(X),\ married(X).\\ single(X) \leftarrow\ & man(X),\ \text{not}\ husband(X). \end{aligned}$$

According to the dependency graph $dep(P_7)$, a stratification Σ for P_7 is

$$S_1 = \{man, married\},\ S_2 = \{husband\},\ S_3 = \{single\}\ .$$

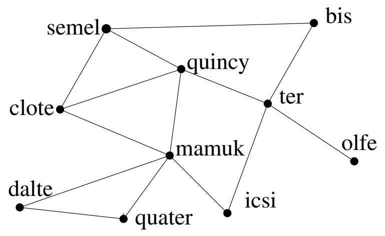


Fig. 3. An example railroad network

We obtain $M_1 = LM(P_{S_1}) = \{man(dilbert)\}$ from the evaluation of $P_{S_1} = \{man(dilbert)\}$. When evaluating M_2 we obtain

$$P_{S_2} = \{husband(X) \leftarrow man(X), married(X)\} \ .$$

Note that $HB^\star(P_{S_1}) = \{man(dilbert), married(dilbert)\}$. It is easy to see that $M_2 = \{man(dilbert)\}$ is a model for P_{S_2}, and that $M_2 \cap HB^\star(P_{S_1}) = M_1 \cap HB^\star(P_{S_1})$; also, M_2 is the least model having these properties.

For the evaluation of M_3, note that

$$P_{S_3} = \{single(X) \leftarrow man(X), \text{ not } husband(X)\} \ .$$

Thus one finds that $M_3 = \{single(dilbert)\} \cup M_2$ is the least model of P_{S_3} such that $M_3 \cap HB^\star(P_{S_2}) = M_2 \cap HB^\star(P_{S_2})$.

It is worth noting that stratifications are not unique. For instance, one can compute the iterative least models using an alternative stratification Σ', in which $S_1 = \{man, married, husband\}$ and $S_2 = \{single\}$.

In both cases the iterative least model obtained at the last iteration is the same. An important result tells us that, provided a stratification exists, other stratifications produce the same final model.

Theorem 3 ([23]). *Let P be a stratified program. Then for every stratifications Σ and Σ' of P, it holds that $M_{P,\Sigma} = M_{P,\Sigma'}$.*

Hence, we can drop the dependency of $M_{P,\Sigma}$ on a given stratification Σ and define $M_P = M_{P,\Sigma}$ (for a Σ of choice) as the canonical model for P, which is referred to as *perfect model* [21].[9]

Example 13 (Railroad network). Take, as an example, the railroad network given in Figure 3. The goal is to determine whether safe connections between locations are possible. Given two railroad stations a and b, a *cutpoint station* c for a and b is such that if connections to c fail, there is no alternative connection between a and b. We will say that the connection between a and b is safe if there are no cutpoints between a and b. In Figure 3, ter is a cutpoint for $olfe$ and $semel$, while $quincy$ is not.

The above problem can be modeled as follows. First, we introduce the set of predicates:

[9] In fact, Przymusinski and Apt et al. developed their semantics independently, but the proposals coincide on stratified programs, and the name *perfect model* for M_P is customary.

$$
\begin{array}{rll}
\mathit{linked}(A,B) \leftarrow & \mathit{link}(A,B). & (R_1)\\
\mathit{linked}(A,B) \leftarrow & \mathit{link}(B,A). & (R_2)\\
\mathit{connected}(A,B) \leftarrow & \mathit{linked}(A,B). & (R_3)\\
\mathit{connected}(A,B) \leftarrow & \mathit{connected}(A,C), \mathit{linked}(C,B). & (R_4)\\
\mathit{cutpoint}(X,A,B) \leftarrow & \mathit{connected}(A,B), \mathit{station}(X), & (R_5)\\
 & \text{not } \mathit{circumvent}(X,A,B). & \\
\mathit{circumvent}(X,A,B) \leftarrow & \mathit{linked}(A,B), X \neq A, \mathit{station}(X), X \neq B. & (R_6)\\
\mathit{circumvent}(X,A,B) \leftarrow & \mathit{circumvent}(X,A,C), \mathit{circumvent}(X,C,B). & (R_7)\\
\mathit{has\_icut\_point}(A,B) \leftarrow & \mathit{cutpoint}(X,A,B), X \neq A, X \neq B. & (R_8)\\
\mathit{safely\_connected}(A,B) \leftarrow & \mathit{connected}(A,B), & (R_9)\\
 & \text{not } \mathit{has\_icut\_point}(A,B). & \\
\mathit{station}(X) \leftarrow & \mathit{linked}(X,Y). & (R_{10})
\end{array}
$$

Fig. 4. Railroad program P_r

- $\mathit{station}(a)$: a is a railway station;
- $\mathit{link}(a,b)$: there is a direct connection from station a to b;
- $\mathit{linked}(a,b)$: the symmetric closure of link; that is, $\mathit{linked}(a,b)$ and $\mathit{linked}(b,a)$ hold whenever $\mathit{link}(a,b)$ holds;
- $\mathit{connected}(a,b)$: there is path linking a to b, either direct or through intermediate stations;
- $\mathit{cutpoint}(x,a,b)$: each existing path from a to b goes through station x;
- $\mathit{circumvent}(x,a,b)$: when going from a to b one can avoid x; that is, there is a path between a and b not passing from x;
- $\mathit{has\_icut\_point}(a,b)$: there is at least one cutpoint between a and b;
- $\mathit{safely\_connected}(a,b)$: a and b are connected with no cutpoint.

We will assume that atoms of form $\mathit{link}(a,b)$ are given as set of facts describing the railroad network at hand. Other predicates are defined according to the program P_r shown in Figure 4.[10] Informally, R_1 and R_2 define linked as the symmetric closure of link, and $\mathit{connected}$ is defined by means of rules R_3 and R_4. Roughly speaking, R_3 expresses that a and b are $\mathit{connected}$ if there is a direct link among them, while R_4 expresses that a and b are $\mathit{connected}$ if there is a node c, which a is connected to, and c has a link to b. Negation is exploited in R_5 for defining $\mathit{cutpoint}$s: x is a cutpoint for all the paths from a to b if a and b are $\mathit{connected}$ and x is a $\mathit{station}$ for which $\mathit{circumvent}(x,a,b)$ does not hold.

Now, let us analyze how to define the notion of *circumvention*. There are two ways for circumventing a station x when going from a to b: either there exists a direct link from a to b (rule R_6) or one can $\mathit{circumvent}$ x when going from a to c and then $\mathit{circumvent}$ x when going from c to b (rule R_7).

[10] The predicate $\neq$ is a "built-in" predicate, which cannot be user defined. It is thus not shown in the evaluation and the dependency graph.

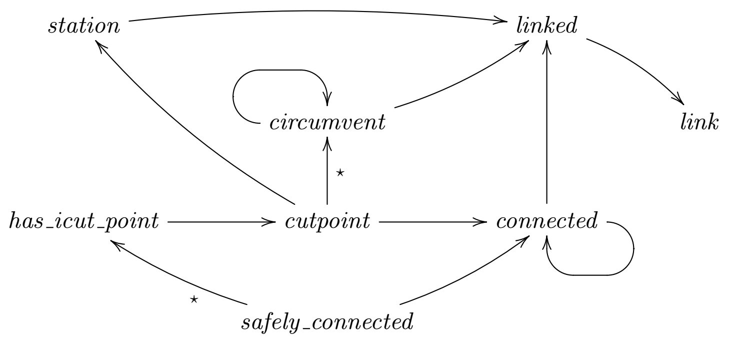


Fig. 5. Dependency graph $dep(P_r)$ of the railroad program P_r

Accordingly, the path from a to b *has a cutpoint* if there is a nontrivial (i.e., x is neither equal to a or b) cutpoint from a to b (rule R_9). Again, negation is exploited for defining when a and b are safely connected (rule R_9): couples of endpoint stations are safely connected if they are $connected$ and do not have cutpoints. Eventually, rule R_{10} defines a $station$ as those nodes which are directly $linked$ to others.

The dependency graph of P_r is shown in Figure 5. A possible stratification of P_r is $\Sigma_r = \{S_1, S_2, S_3\}$, where

– $S_1 = \{link, linked, station, circumvent, connected\}$,
– $S_2 = \{cutpoint, has\_icut\_point\}$, and
– $S_3 = \{safely\_connected\}$.

We then get the iterative least models

– $M_1 = \{\ linked(semel, bis), linked(bis, ter), linked(ter, olfe), \ldots,$
$\quad connected(semel, olfe), \ldots, circumvent(quincy, semel, bis), \ldots\}$,
– $M_2 = M_1 \cup \{\ cutpoint(ter, semel, olfe), has\_icut\_point(semel, olfe), \ldots\}$, and
– $M_3 = M_2 \cup \{\ safely\_connected(semel, bis), safely\_connected(semel, ter)\ \}$.

The iterative least model M_3 is then a perfect model for P_r. Note that M_3 does not contain $safely\_connected(semel, olfe)$.

3.2 Unstratified Negation

The notion of perfect model is however inadequate whenever a program has no stratification. This happens when two or more predicates are mutually defined over "not," like in the following program P_u:

$$\begin{aligned} man(dilbert). \\ single(X) \leftarrow man(X), \text{not } husband(X). \\ husband(X) \leftarrow man(X), \text{not } single(X). \end{aligned}$$

Note that P_u has two minimal models (which, as shown next, are *stable*):

- $M = \{man(dilbert),\ single(dilbert)\}$ and
- $N = \{man(dilbert),\ husband(dilbert)\}$;

both might be seen as "plausible" scenarios compatible with P_u.

In general, we can associate to a program P a set of *preferred* (or plausible) models $PM(P)$. In the presence of multiple plausible models, each describing a possible scenario specified by a given program, a natural question is how to interpret and how to reconcile possible discrepancies between models appearing in $PM(P)$.

One can consider this issue from two complementary points of view:

1. One point is to see P as a knowledge base, in which explicit (facts) and implicit (rules) information is stored, and wonder if a given query q (or, in general, a formula) holds. Queries can be ground (e.g., $q = man(dilbert)$ holds if q is true w.r.t. P_u according to some criterion), or nonground (e.g., for evaluating $q = man(X)$ we have to find the set of values x such that $man(x)$ holds in P_u).
   In this respect, a ground query q can be answered under *Cautious (Skeptical) Reasoning*, that is q evaluates to true if it is true in *every* model in $PM(P)$, or under *Brave (Credulous) Reasoning*, in which q is true if it is true in *some* preferred model. Similarly, answering a non-ground query q amounts to finding the set of all the ground assignments of q which hold in any preferred model (cautious reasoning) or in some preferred model (brave reasoning).
2. Cautious and brave reasoning can be seen as a form of quantification/iteration over preferred models, which however still depict a single scenario. In cautious reasoning the single scenario (the set of true facts) is described by the intersection of all the models, while in brave reasoning one considers their union, this way discarding the richer information given in $PM(P)$.
   However, each model in $PM(P)$ brings peculiar information: it can be seen as the representation of a possible world compatible with P, or, in other words, as a *solution* to the problem instance encoded by P. *Model generation* (that is, the computation of the set $PM(P)$) in this respect is—more than query answering—of valuable importance.

Example 14. The preferred models M and N of P_u represent "possible worlds" compatible with P_u. The ground atom $man(dilbert)$ is a cautious and brave consequence of P_u. But, neither $single(dilbert)$ nor $husband(dilbert)$ are cautious consequences, whereas both are brave consequences of P_u (the first holds in M while the second holds in N).

4 Stable Semantics

Many definitions for $PM(P)$ have been conceived in the past, cf. [15, 24]. We will concentrate from this point on the—largely considered the most prominent one—notion of preferred model based on *stable models*.

4.1 Normal Logic Programs – Syntax

A logic program P based on the stable model semantics has the same syntactic building blocks as stratified programs: importantly, it is not necessary that P has a stratification, as we do not rely on the notion of perfect model for computing its semantics. Also, we keep the the notions of Herbrand universe $HU(P)$, Herbrand base $HB(P)$, and interpretation as for not-free ("positive") logic programs.

4.2 Stable Model Semantics

First, we will define the stable model semantics for a variable-free (ground) program.

The intuition behind stable model semantics is to treat negated atoms in a special way. Intuitively, such atoms are a source of "contradiction" or "unstability."

Example 15. In P_u from above, one can consider $M' = \{man(dilbert)\}$ as possible, preferred model. Assuming facts in M' as true, note however that the two rules of P_u would enforce to assume that besides $man(dilbert)$ also $single(dilbert)$ and $husband(dilbert)$ are true. On the other hand, if one considers $M'' = \{man(dilbert)$, $single(dilbert)$, $husband(dilbert)\}$ as the set of true facts, it turns out that the two rules of P_u have now their bodies false, and do not give evidence of truth for $single(dilbert)$ and $husband(dilbert)$.

"Stability" can thus be seen as follows: if an interpretation M of P is not—in the sense formalized below—self-contradicting, then it is *stable*.

Definition 9. *The* Gelfond-Lifschitz reduct *[8] (short GL-reduct or simply reduct) of a program P w.r.t. an interpretation M, denoted P^M, is a program obtained by*

1. *removing rules with* $\mathrm{not}\ a$ *in the body for each $a \in M$; and*
2. *removing literals* $\mathrm{not}\ a$ *from all other rules.*

Intuitively, given an interpretation M, the conditions 1 and 2 above enforce truth values for negative literals. If $a \in M$, then a rule's body with the negative literal $\mathrm{not}\ a$ cannot become true. On the other hand, if $a \notin M$, the $\mathrm{not}\ a$ can be assumed true and removed from any body where it occurs.

In other words, M can be seen as an *assumption* about which negated literals are true and what are false; the program P^M incorporates these assumptions. Note that P^M is a positive program, and thus has a least model $LM(P^M)$. If P^M does not "contradict" M, one should expect that $LM(P^M) = M$, that is, M can be reconstructed from scratch applying the rules of P^M. If this happens to be the case, then M can be regarded as being "stable."

Definition 10. *An interpretation M of P is a* stable model *of P, if*

$$M = LM(P^M).$$

Note that $P^M = P$ for any "not"-free program P. Thus, $LM(P)$ (which is equal to $LM(P^M)$) is its single stable model.

Example 16. If we take P_u again in consideration

$$man(dilbert). \qquad (f_1)$$

$$single(dilbert) \leftarrow man(dilbert), \mathrm{not}\ husband(dilbert). \qquad (r_1)$$

$$husband(dilbert) \leftarrow man(dilbert), \mathrm{not}\ single(dilbert). \qquad (r_2)$$

we may have the following "candidate" interpretations:

- $M_1 = \{man(dilbert), single(dilbert)\}$,
- $M_2 = \{man(dilbert), husband(dilbert)\}$,
- $M_3 = \{man(dilbert), single(dilbert), husband(dilbert)\}$
- $M_4 = \{man(dilbert)\}$,

One can verify that only M_1 and M_2 qualify themselves as stable models.

- if we consider M_1 we get that the reduct $P_u^{M_1}$ is

$$\begin{aligned} man(dilbert). \\ single(dilbert) \leftarrow man(dilbert). \end{aligned}$$

  Note that $husband(dilbert) \notin M_1$, thus not $husband(dilbert)$ is removed from r_1. On the other hand r_2 is deleted from P_u since $single(dilbert) \in M_1$: indeed, under the assumption made in M_1, the literal not $husband(dilbert)$ is false and will prevent r_2 to trigger and make its head true.
  The least model of $P_u^{M_1}$ is $\{man(dilbert), single(dilbert)\}$ which coincides with M_1.
  Symmetrically, we can verify that M_2 is stable as well.
- On the other hand, M_3 and M_4 are not stable. If we take $M_3 = \{man(dilbert)$, $single(dilbert)$, $husband(dilbert)\}$ in consideration, we find that $P_u^{M_3}$ consists only of $man(dilbert)$. Both r_1 and r_2 are indeed deleted. Thus, $LM(P_u^{M_3}) = \{man(dilbert)\} \neq M_3$. This means that the assumptions made in M_3 are not "stable" with respect to negated literals in P_u.
  If we take $M_4 = \{man(dilbert)\}$, we observe that $P_u^{M_4}$ consists of

$$\begin{aligned} man(dilbert). \\ single(dilbert) \leftarrow man(dilbert). \\ husband(dilbert) \leftarrow man(dilbert). \end{aligned}$$

  given that both not $husband(dilbert)$ and not $single(dilbert)$ are removed from r_1 and r_2 respectively. Therefore, $LM(P_u^{M_4}) = \{man(dilbert)$, $single(dilbert)$, $husband(dilbert)\} \neq M_4$.

Notably, there are situations in which "stability" is impossible and no meaning can be assigned to a program.

Example 17. The program P_i

$$p \leftarrow \text{not } p. \tag{8}$$

has no stable models. Consider any interpretation M for P_i such that $p \notin M$. Thus, $\text{not}\, p$ is true and the body of (8) is satisfied, which means that p should be true as well in order for M being a model for P_i. But this is in direct contradiction to $p \notin M$. Now, if we take an interpretation M' such that $p \in M'$, we get that $\text{not}\, p$ is false and our rule (8) is satisfied, hence M' is a model for P_i. But it is not a stable model, as the reduct $P_i^{M'} = \emptyset$, and we have that $LM(P_i^{M'}) = \emptyset$, which is different from M'.

If we take an arbitrary program P, and add the rule (8) (with p being a new propositional atom), we get that P has no stable model.

Example 18. Consider the program P_s:

$$s \leftarrow \text{not } q. \qquad (r_1)$$
$$q \leftarrow \text{not } s. \qquad (r_2)$$
$$p \leftarrow q, \text{not } s. \qquad (r_3)$$
$$f \leftarrow s, \text{not } f. \qquad (r_4)$$

P_s has a single stable model $M_1 = \{p, q\}$, while $M_2 = \{s\}$ is not stable.

– Indeed, for $M_1 = \{p, q\}$ we have that in $P_s^{M_1}$ the rules r_1 is deleted, while r_2, r_3 and r_4 are modified, obtaining:

$$q.$$
$$p \leftarrow q.$$
$$f \leftarrow s$$

For which $LM(P_i^{M_1}) = \{p, q\} = M_1$.

– For $M_2 = \{s\}$, we get $P_s^{M_2}$ by deleting r_2 and r_3 from P_s and updating r_1 and r_4:

$$s.$$
$$f \leftarrow s.$$

We get $LM(P_s^{M_2}) = \{s, f\} \neq M_2$. Note that $M_3 = \{s, f\}$ is not stable as well. Indeed, one can observe that rule r_4 prevents the existence of a stable model containing s.

Programs with Variables. As for the case of positive and stratified programs, it is immediate to lift the notion of stable model from propositional programs to non-ground ones. Intuitively, this step amounts to considering non-ground rules (containing variables) as shorthands for all their possible ground instances, obtained using a domain of choice for the terms which can be constructed. This latter domain is usually the Herbrand universe of the program at hand. The stable semantics of non-ground programs is thus obtained by means of a reduction to the variable-free case.

Definition 11. *Given a program P, an interpretation M of P is a* stable model *of P, if M is a stable model of $grnd(P)$.*

Example 19. Consider the following variant of P_u which we will call $P_{u'}$:

$$man(dilbert). \qquad (r_1)$$
$$woman(alice). \qquad (r_2)$$
$$single(X) \leftarrow man(X), \text{not } husband(X). \qquad (r_3)$$
$$husband(X) \leftarrow man(X), \text{not } single(X). \qquad (r_4)$$

We have that, for instance,

$$grnd(r_3) = \{\ single(dilbert) \leftarrow man(dilbert), \text{not } husband(dilbert).\\ single(alice) \leftarrow man(alice), \text{not } husband(alice).\quad \};$$

$$grnd(P_{u'}) = \{\ man(dilbert).\\ woman(alice).\\ single(dilbert) \leftarrow man(dilbert), \text{not } husband(dilbert).\\ single(alice) \leftarrow man(alice), \text{not } husband(alice).\\ husband(dilbert) \leftarrow man(dilbert), \text{not } single(dilbert).\\ husband(alice) \leftarrow man(alice), \text{not } single(alice).\quad \}.$$

The program $grnd(P_{u'})$, and thus $P_{u'}$, has the following stable models:

- $M_1 = \{man(dilbert), woman(alice), single(dilbert)\}$
- $M_2 = \{man(dilbert), woman(alice), husband(dilbert)\}$

4.3 Semantic Properties of Stable Models

The success of stable models as semantics for normal logic programs (with arbitrary usage of negation) relies on two important aspects: first, stable models have a strong theoretical basis, and enjoy many properties which reflect natural intuitions. Second, as it will be seen in Section 6 they pave the way to a innovative problem modeling methodology.

We survey here some important (most of which desirable) theoretical properties of stable models. The reader can refer to [25–27] for other insights, alternative definitions and properties of stable models.

We first consider the relationship between stable models and classical models of a logic program, i.e., when negation as failure is interpreted as classical negation.

To this end, the notion of (classical) Herbrand model is easily lifted to clauses with negated literals in their bodies.

Definition 12. *Let I be an interpretation. Then I is a* model *of*

- *a ground clause $C : a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$, denoted $I \models C$, if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $\{a, c_1, \ldots, c_n\} \cap I \neq \emptyset$.*
- *a clause C, denoted $I \models C$, if $I \models C'$ for every $C' \in grnd(C)$;*
- *a program P, denoted $I \models P$, if $I \models C$ for every clause C in P.*

Intuitively, the above definition lifts Definition 4 by taking in consideration negated literals: an interpretation I is, again, "compatible" with a clause C either if it contains the head of C, or if the body of C is false. A body can be false either if some positive b_i is not in I, or if some c_i is in I. One expects that if the body of C is true, then also its head must be true: indeed, if $b_1, \ldots, b_m \in I$ and $c_1, \ldots, c_n \notin I$, I can be model of C only if it contains a.

The above definition complies with the notion of Herbrand model satisfying the clause $a \vee \text{not } b_1 \vee \ldots \vee \text{not } b_m \vee c_1 \vee \ldots \vee c_n$, where not is interpreted as classical negation. Now the following property holds:

Theorem 4. *1. Every stable model M of P is a model of P.*
2. A stable model M does not contain any model M' of P properly ($M' \not\subset M$), i.e., is a minimal model of P (w.r.t. $\subseteq$).

The above properties guarantee that stable models of a program with negation enjoy two of the desirable properties holding for least models of positive programs: first, a stable model M of P is "compatible" with all the rules of P, that is, it does not contradict P. Also, M contains a minimal amount of facts which one must admit to be true for gaining the "compatibility" with the scenario described by P, and no unnecessary and/or redundant information.

Corollary 1. *Stable models are incomparable w.r.t. $\subseteq$, i.e., if M_1 and M_2 are different stable models of P, then $M_1 \not\subseteq M_2$ and $M_2 \not\subseteq M_1$.*

Also, stable models gracefully generalize the semantics for positive programs (the least model of a positive program P is clearly the unique stable model of P), and for stratified semantics: indeed, the perfect model of a stratified program is also its unique stable model.

Theorem 5. *If a program P is stratified, then P has a single stable model, which coincides with the perfect model.*

Note, for instance, that the railroad program P_r is stratified. Its single stable model coincides with the perfect model. It is indeed worth noting that there is only one stable configuration for a stratified program although it can have multiple minimal models.

Example 20. If one considers the program P_m

$$\begin{aligned} p(a). \\ r(X) \leftarrow p(X), \text{not}\, q(X). \end{aligned}$$

we get two minimal models $M_1 = \{p(a), r(a)\}$ and $M_2 = \{p(a), q(a)\}$ for P_m. Note that while M_1 is stable, M_2 is not stable, as the reduct $grnd(P_m)^{M_2} = \{p(a)\}$, and $LM(grnd(P_m)^{M_2}) = \{p(a)\} \neq M_2$.

What makes M_2 different from M_1 is the fact that there is neither rule nor fact in P_m justifying the presence of $q(a)$ in a model.

Indeed one can see stable models as models in which all atoms $a \in M$ are somehow "supported" by evidence: in a sense, a stable model "supports", or "gives evidence" of the truth of each $a \in M$.

Theorem 6. *Given a program P and an interpretation I, let*

$$T_P(I) = \left\{ a \;\middle|\; \begin{array}{l} \textit{there is some } r = a \leftarrow b_1, \ldots, b_m, c_1, \ldots, \text{not}\, c_n \in grnd(P) \\ \textit{such that } \{b_1, \ldots, b_m\} \subseteq I, \{c_1, \ldots c_m\} \cap I = \emptyset \end{array} \right\} .$$

If I is a stable model of P, then $T_P(I) = I$.

Example 21. Note that $q(a)$ in example 20 is *unsupported* in M_2, indeed $q(a) \notin T_P(M_2)$.

Nonetheless, it must be noted that there are models which are minimal fixed points of T_P, but are however not stable:

Example 22. Consider the short program P_s:

$$\begin{aligned} a &\leftarrow \text{not}\, b. \\ b &\leftarrow c. \\ c &\leftarrow b. \end{aligned}$$

Note that $M_1 = \{a\}$ and $M_2 = \{b, c\}$ are both minimal and such that $T_{P_s}(M_1) = M_1$ and $T_{P_s}(M_2) = M_2$, respectively. In particular, b and c are—in a sense—self-supported. Consider the reducts $P_s^{M_1} = \{a \leftarrow;\ b \leftarrow c;\ c \leftarrow b\}$ and $P_s^{M_2} = \{b \leftarrow c;\ c \leftarrow b\}$. We have that $LM(P_s^{M_1}) = \{a\} = M_1$ and $LM(P_s^{M_2}) = \emptyset \neq M_2$, thus M_1 is a stable model, whereas M_2 is just a minimal model, but not a stable one.

Self-supported atoms are in general not desirable, since they can lead to paradoxical scenarios in which true facts are not supported by evidence; a and b from the previous example are indeed *unfounded* w.r.t M_2 in the sense specified below.

Definition 13 ([22]). *Given a program P, a set $U \subseteq HB_P$ is an* unfounded set *of P relative to an interpretation I, if for every $a \in U$ and every $r \in ground(P)$ with $H(r) = a$, either*

1. *There is some atom b appearing as positive literal in the body of r which is such that either $b \notin I$ or $b \in U$, or*
2. *There is some atom b appearing as negative literal in the body of r such that $b \in I$.*

For normal programs there exists the greatest unfounded set of P relative to I, denoted by $U_P(I)$.

Intuitively, if I is compatible with P, then all atoms in $U_P(I)$ can be safely switched to false and the resulting interpretation is still compatible with P. Assuming I as a set of true facts, there is no rule in P that can justify an atom $a \in U$ becoming true.

An interpretation I is called *unfounded-free*, if $I \cap U = \emptyset$ for each unfounded set U of P rel. to I. In other words, I is *unfounded-free* iff $I \cap U_P(I) = \{\}$. [11]

The notion of unfounded set extends the notion of "non-supportedness" by implicitly forbidding support of an atom by an atom which is unfounded. For gaining "foundedness" by M an atom $a \in M$ necessitates support by a rule whose body is made true by founded atoms only (not belonging to the unfounded set at hand).

Theorem 7 (implicit in [28]). *Given a program P, a model M of P is stable iff M is* unfounded-free.

[11] Note that, for more general classes of programs than normal programs (e.g., disjunctive program as later defined in Section 5.3), $U_P(I)$ is undefined. More generally, we can then say that I is unfounded-free, if there is no (non-empty) subset of I which is an unfounded set.

Example 23. If we take P_u again in consideration

$$man(dilbert). \qquad (f_1)$$
$$single(dilbert) \leftarrow man(dilbert), \text{not } husband(dilbert). \qquad (r_1)$$
$$husband(dilbert) \leftarrow man(dilbert), \text{not } single(dilbert). \qquad (r_2)$$

And the four following "candidate" interpretations:

- $M_1 = \{man(dilbert), single(dilbert)\}$,
- $M_2 = \{man(dilbert), husband(dilbert)\}$,
- $M_3 = \{man(dilbert), single(dilbert), husband(dilbert)\}$
- $M_4 = \{man(dilbert)\}$,

One can observe that M_3 has the greatest unfounded set $U_{P_u}(M_3) = \{single(dilbert), husband(dilbert)\}$: assuming M_3 as a set of "true" facts, there is indeed no rule which could make atoms in $U_{P_u}(M_3)$ true. M_3 is thus not unfounded-free. Note that M_4 is not a model at all, since r_1 and r_2 are not satisfied.

Example 24. Note that the minimal model $M_2 = \{b, c\}$ of P_s is not unfounded free: indeed $U_{P_s}(M_2) = \{b, c\}$.

Reasoning from stable models. Since a logic program P might have no, one, or multiple stable models, the question is how inference from P should be defined. With respect to a particular stable model M, a ground atom a is considered to be true (denoted $M \models a$), if $a \in M$, and false, if $a \notin M$. This is usually extended to inference from all stable models of P in two dual modes, as mentioned already in Section 3.2:

Brave Reasoning An atom a is a *brave* (or *credulous*) consequence of P, denoted $P \models_b a$, if $M \models a$ for some stable model of P;
Cautious Reasoning An atom a is a *cautious* (or *skeptical*) consequence of P, denoted $P \models_c a$, if $M \models a$ for every stable model of P.

These notions can be extended to propositional combinations of ground atoms in the natural way (where $M \models \neg a$ iff $a \notin M$), and similarly to (combinations of) closed formulas.

Both $\models_b$ and $\models_c$ are *nonmonotonic*, as adding further rules to P might invalidate a conclusion.

Example 25. If we reconsider the program P_m in Example 20, then both $P_m \models_b r(a)$ and $P_m \models_c r(a)$, as $r(a)$ is true in the unique stable model of P_m. However, for $P'_m = P_m \cup \{q(a)\}$, neither $P'_m \models_b r(a)$ nor $P'_m \models_c r(a)$ holds, as $r(a)$ is false in the single stable model $\{p(a), q(a)\}$ of P'_m.

From this example, one might believe that the nonmonotonic behavior of inference is due to the fact that we added some fact ($q(a)$) that was missing before, but that this would not happen if the fact were already a consequence; that is, that inference satisfies *cautious monotonicity*:

- If $P \models_x a$ and $P \models_x b$, then $P \cup \{a\} \models_x b$.

where $x \in \{b, c\}$. This property is obviously fulfilled for classical inference $\models$ in place of $\models_x$. However, it does not hold for cautious reasoning under stable semantics.

Proposition 2. *In general, $P \models_c a$ and $P \models_c b$ does not imply that $P \cup \{a\} \models_c b$.*

In fact, the property fails even if P has a single stable model. For example, consider the program $P = \{b \leftarrow \text{not } c; c \leftarrow \text{not } b; a \leftarrow \text{not } a; a \leftarrow b\}$. This program has the single stable model $M = \{a, b\}$, and thus $P \models_c a$ and $P \models_c b$. However, the program $P \cup \{a\}$ has another stable model, viz. $N = \{a, c\}$, and thus $P \cup \{a\} \not\models_c b$. The property is, however, true for brave reasoning.

Similarly then, also the stronger property of *cumulativity* fails:

- If $P \models_x a$, then $P \models_x b$ iff $P \cup \{a\} \models_x b$.

That is, by adding consequences as "lemmas," we might change the set of conclusions that can be drawn (which is not the case for classical inference $\models$). In fact, this property also fails for brave reasoning, as shown by the above examples (e.g., $P \cup \{a\} \models_b c$ while $P \not\models_b c$).

In conclusion, care is needed when arguing about how rules in a program compute truth values for atoms under stable semantics. As long as atoms do not depend on negation through cycles, i.e., in the stratified part of a program, adding atoms that are computed true as facts does not change the semantics. Fortunately, this can be generalized to settings where a program can be split into an "lower' and an "upper" part where the former informally provides input to the latter in a modular way [29]. In other cases, one has to carefully examine the effects of adding atoms—in an unfounded way—as facts. More about properties of consequences from stable models can be found e.g. in [27].

4.4 Computational Properties

There are many computational tasks related to logic programs under stable model semantics: one might want to check if a given program P is consistent (that is, it admits at least one stable model), or to compute one, or all, of its models. Also it can be of interest to determine truth of a given query Q under brave or cautious reasoning. We briefly focus here on the problem CONS of deciding whether a given input program P has some stable model, that is, deciding the consistency of P under stable model semantics. The computational complexity of CONS has direct impact on other related problems, thus giving an indication of the complexity of other related problems. For instance, evidence of consistency can be given by computing one stable model.

It turns out that assessing consistency of a ground program P is in general NP-complete.

Theorem 8 ([30]). *The problem* CONS *of deciding whether a given ground program P has some stable model is* NP*-complete.*[12]

[12] Recall that NP is the class of problems solvable in polynomial time on a non-deterministic Turing machine [31].

Intuitively, this result can be justified by thinking of a simple nondeterministic algorithm for checking the existence of a stable model for P. For showing that CONS is in NP one can: (i) guess a candidate stable model M; (ii) check in polynomial time if M is stable (e.g. by verifying $U_P(M) = \{\}$). Also, one can show that it is possible to build a program P_ϕ, having a stable model iff a given propositional formula ϕ in CNF is true (where P_ϕ is of size at most polynomially higher than the size of ϕ).

However, computational complexity might change depending on allowed extensions (disjunction, presence of function symbols, etc.):

– For "not"-free programs and stratified programs, CONS can be solved in polynomial time (in fact, solvable in linear time);
– For programs with variables but not function symbols, CONS has exponentially higher complexity (NEXP-complete);
– For non-ground, arbitrary programs (allowing functional terms), CONS is undecidable. There are however known syntactic conditions on the usage of function symbols which retain complexity in 2-EXP [32, 33] resp. 2-NEXP [34, 35].[13]

It is important to note the dramatic change in complexity when P is non-ground. This should not be surprising if one considers that, usually, $grnd(P)$ is exponentially bigger than P.

Example 26. Given the rule r_g

$$r(X_1, \ldots, X_k) \leftarrow h(a, b), c_1(X_1), \ldots, c_k(X_k)$$

one can easily observe that $|grnd(r_g)| = O(2^k)$.

In particular one can observe that the size of a grounded program can be exponentially bigger than its original non-ground counterpart if k is allowed to vary, that is, if programs can have arbitrarily long rules, and arbitrarily large arities. This might not be the case if a bound on such parameters is given (see e.g. [36]). Also, one might wonder why the introduction of function symbols makes CONS undecidable. One can easily see that, in this setting, it is possible to have stable models of infinite size:

Example 27. Consider the program P_f:

$$\begin{aligned} p(a). \\ p(f(X)) \leftarrow p(X). \end{aligned}$$

We can observe that $grnd(P_f) = \{p(a), p(f(a)) \leftarrow p(a), p(f(f(a))) \leftarrow p(f(a)), \ldots\}$ is infinite, as well as its unique stable model $M_{inf} = \{p(a), p(f(a)), p(f(f(a)), \ldots\}$.

It is thus not surprising that for non-ground programs, admitting functions symbols, CONS and other related reasoning problems become as difficult as deciding the termination of a Turing machine on a given input.[14]

[13] A decision problem is in 2EXP (2NEXP) time, if it can be solved by a (non-)deterministic Turing Machine in time $O(2^{2^{p(n)}})$, where $p(\cdot)$ is a polynomial and n is the size of the input instance.

[14] The reader can find in [12] a thorough collection of results regarding computational complexity of logic programming under various semantics including the stable models semantics.

5 Extensions

In the above sections, we have dealt with the motivation and history of answer set programming, and described syntax and semantics of gradually increasing expressive program classes. In particular, we looked into the class of normal logic programs under stable models semantics and showed their alluring semantic properties. But, so far, we did not touch upon the full area of answer set programming. In this section, we will approach the main topic of this chapter and show more syntactic extensions of normal logic programs and define their semantics, which, eventually, brings us to the answer set semantics.

We now turn our attention to three particular extensions of normal logic programs that lead to the notion of Answer Set Programming: (i) (integrity) constraints (rules with empty head) like

$$\leftarrow \mathit{edge}(X,Y), \mathit{red}(X), \mathit{red}(Y) \ , \tag{9}$$

which forces that adjacent nodes in a graph are not allowed to have the colour red; (ii) strong (or "classical") negation in atoms, e.g., $-\mathit{single}(\mathit{dilbert})$ (Dilbert is *known* not to be a single); and (iii) disjunctive rules, i.e., allowing for disjunctions in rule heads like in

$$\mathit{female}(X) \vee \mathit{male}(X) \leftarrow \mathit{person}(X) \ ,$$

which intuitively means that persons are either female or male. For many crucial knowledge representation tasks, these extensions are not only desirable, but also necessary for succinct encodings of problems. Programs that permit strong negation are also called *Extended Logic Programs* (ELP). If ELPs additionally allow for disjunctive rules, we obtain the class of disjunctive ELPs, which are also called *Disjunctive Logic Programs* (DLP).

Next, we will look into these important extensions in more detail and then provide syntax and semantics of ELPs and DLPs.

5.1 Constraints

Integrity constraints check admissibility of models, possibly using auxiliary predicates defined by normal stratified rules. For instance, the constraint rule (9) can be equally well expressed as the "killing clause"

$$\mathit{falsity} \leftarrow \text{not } \mathit{falsity}, \mathit{edge}(X,Y), \mathit{red}(X), \mathit{red}(Y) \ , \tag{10}$$

where $\mathit{falsity}$ is a fresh propositional atom. Now, if there is an interpretation I for a program with the constraint (9) such that I contains $edge(a,b)$, $red(a)$, and $red(b)$, but $falsity \notin I$, then (10) is applicable and forces $falsity$ to be true. But then, I cannot be a model for our program, as $falsity$ is false in I. This means that (10) "kills" all models that do not satisfy the constraint (9).

5.2 Strong Negation

In Section 3, we have defined normal logic programs, i.e., logic programs that allow for weak negation in rule bodies. The intuitive meaning of $\text{not } a$ is that "a cannot be proved (derived) using rules," and that a is false by default (or *believed* to be false). But this is different from (provably) *knowing* that a is false, which is expressed by $\neg a$; in ASP, one also writes $-a$ for this.

Example 28 (by John McCarthy). Consider an agent A with the following task: "At a railroad crossing, cross the rails if no train approaches." We may encode this scenario using one of the following two rules:

$$\mathit{walk} \leftarrow \mathit{at}(A,L), \mathit{crossing}(L), \text{not } \mathit{train\_approaches}(L). \tag{11}$$

$$\mathit{walk} \leftarrow \mathit{at}(A,L), \mathit{crossing}(L), -\mathit{train\_approaches}(L). \tag{12}$$

In the following, let us assume that A is at some crossing L.

If we take (11) as encoding for the railroad-crossing task, and A cannot infer from her beliefs that $\mathit{train\_approaches}(L)$ is true, then A will conclude to walk even though A cannot be sure that there is no approaching train: her beliefs might not represent the state of the world completely. Now, if we take (12) as the encoding, A will only walk if she can prove that there is no approaching train.

In (11), an update to A's knowledge can lead to revised conclusions; if we add $\mathit{train\_approaches}(L)$, then A will refuse to walk. This is the typical behavior of non-monotonic rules like (11), but may not be desired in critical situations like crossing a railroad, as an approaching train, which has not been perceived by A yet, might cause devastating effects on the agent. From this point of view, the rule (12) employing strong negation is preferable.

There are several ways to express negative knowledge using strongly negated atoms. One way is to explicitly state them as facts in a knowledge base. For instance, the fact $-\mathit{broken}(\mathit{battery})$ expresses that a battery is definitely not broken. If this knowledge base concludes in a different rule that $\mathit{broken}(\mathit{battery})$ holds, then we face inconsistency, and this causes to vanish all models of that particular knowledge base.

Another useful application for strong negation (in combination with weak negation) is to express *default rules*. For example, we can express that "a bird flies by default" with the rule $\mathit{flies}(X) \leftarrow \mathit{bird}(X), \text{not } -\mathit{flies}(X)$.

Extended Logic Programs. Adding strong negation to normal logic programs leads to the so called *extended logic programs*.

Definition 14. *An* extended logic program (ELP) *is a finite set of rules*

$$a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \qquad (n, m \geq 0) \tag{13}$$

where a and all b_i, c_j are atoms or strongly negated atoms in a first-order language L.

The semantics of ELPs can be defined in different ways, either genuinely by considering sets of ground literals rather than sets of atoms as basis, as done in [11], or by a simple reduction to normal logic programs that compiles strong negation away; we follow here for simplicity the latter. To this end, we

– view negative literals "$-p(\boldsymbol{X})$" as atoms with fresh predicate symbols $-p$, for each atom $p(\boldsymbol{X})$;
– add the clause

$$\mathit{falsity} \leftarrow \text{not } \mathit{falsity}, p(\boldsymbol{X}), -p(\boldsymbol{X}) \tag{14}$$

to P (this prevents that $p(\boldsymbol{X})$ and $-p(\boldsymbol{X})$ are true at the same time); and
– select the stable models of the resulting program P'. These are called *answer sets* of P.

Note that extended logic programs have similar properties as normal logic programs under stable models semantics. For a ground atom a, constraint (14) prevents that both a and $-a$ are contained in answer sets. One takes a *three-valued view* on this: an atom may be true, false, or *undefined* (i.e., we *don't know* if the atom is true or false). This contrasts with the two-valued view of stable models of a normal logic program, in which an atom a is either true (if a is in the model) or false (if a is not on the model, in the spirit of Reiter's Closed World Assumption [37]).[15]

The use of strong negation may cause inconsistency, even if a program does not have weak negation. For example, take the program P

$$\begin{aligned} &\mathit{true}.\\ \mathit{trivial} \leftarrow\ &\mathit{true}.\\ a \leftarrow\ &\mathit{true}.\\ -a \leftarrow\ &\mathit{true}. \end{aligned}$$

which derives both a and $-a$. The constraint (14) prevents that P has answer sets, thus P is inconsistent. However, this inconsistency is of a different quality than the one cause by default negation (cf. Example 17).

Example 29. The next program is a knowledge base for determining if one should query the *science citation index* (*sci*) or the citeseer database:

$$\begin{aligned} \mathit{up}(S) &\leftarrow \mathit{website}(S), \text{not } -\mathit{up}(S). && (r_1)\\ -\mathit{query}(S) &\leftarrow -\mathit{up}(S). && (r_2)\\ \mathit{query}(\mathit{sci}) &\leftarrow \text{not } -\mathit{query}(\mathit{sci}), \mathit{up}(\mathit{sci}). && (r_3)\\ \mathit{query}(\mathit{citeseer}) &\leftarrow \text{not } -\mathit{query}(\mathit{citeseer}), -\mathit{up}(\mathit{sci}), \mathit{up}(\mathit{citeseer}). && (r_4)\\ \mathit{flag\_error} &\leftarrow -\mathit{up}(\mathit{sci}),\ -\mathit{up}(\mathit{citeseer}). && (r_5)\\ \mathit{website}(\mathit{sci}). &\qquad \mathit{website}(\mathit{citeseer}). \end{aligned}$$

In rule (r_1), we define that websites are up by default, and (r_2) encodes that a website known to be not up should not be queried. The rules (r_3) and (r_4) give a preference on the websites: whenever we cannot prove that sci is not usable and sci is available, then we should query the science citation index, but we should only query $\mathit{citeseer}$ if it is available for querying and sci is down. In (r_5), we simply raise an error-flag whenever both websites are down.

[15] The answer sets of an ELP P without strong negation coincide with the stable models of P, and thus the terms are often used interchangeably (confusing two- vs three-valuedness).

The single answer set of this program is

$$M = \{\mathit{website}(\mathit{sci}), \mathit{website}(\mathit{citeseer}), \mathit{up}(\mathit{sci}), \mathit{up}(\mathit{citeseer}), \mathit{query}(\mathit{sci})\} \ ,$$

whose intuitive meaning is that we should query the science citation index, even though $\mathit{citeseer}$ is up and running.

If we add to our knowledge base the rule

$$-\mathit{query}(S) \leftarrow \text{not } \mathit{query}(S), -\mathit{reliable}(S) \qquad (r_6)$$

and the facts that sci is down and $\mathit{citeseer}$ is unreliable,

$$-\mathit{up}(\mathit{sci}) \text{ and } -\mathit{reliable}(\mathit{citeseer}) \ ,$$

we can witness a different behavior. Intuitively, (r_6) creates a nondeterminism in our program, as for websites S that are known to be unreliable we can infer $-\mathit{query}(S)$, provided that we cannot prove $\mathit{query}(S)$. But rules (r_3) and (r_4) gives us similar knowledge, except with unlike signs: we can infer a positive fact $\mathit{query}(S)$ given that we cannot prove $-\mathit{query}(S)$. To resolve this conflicting views, we obtain for our knowledge base under answer set semantics exactly two answer sets, with each intuitively describe the corresponding alternative view on our site selection problem:

- $M_1 = \{\mathit{website}(\mathit{sci}), \mathit{website}(\mathit{citeseer}), -\mathit{up}(\mathit{sci}), \mathit{up}(\mathit{citeseer}), -\mathit{reliable}(\mathit{citeseer}), -\mathit{query}(\mathit{sci}), \mathit{query}(\mathit{citeseer})\}$, and
- $M_2 = \{\mathit{website}(\mathit{sci}), \mathit{website}(\mathit{citeseer}), -\mathit{up}(\mathit{sci}), \mathit{up}(\mathit{citeseer}), -\mathit{reliable}(\mathit{citeseer}), -\mathit{query}(\mathit{sci}), -\mathit{query}(\mathit{citeseer})\}$,

i.e., in M_1 we have chosen to query $\mathit{citeseer}$ and in M_2 we conclude that we should not query $\mathit{citeseer}$, thus querying no website at all.

Relationship to Reiter's Default Logic. It has been noted already in [11] that ELPs are closely related to Reiter's famous Default Logic [38]. For an ELP clause C of form (13), consider the corresponding default

$$d(C) = \frac{b_1 \wedge \cdots \wedge \ b_m : \ \neg.c_1, \ \ldots, \ \neg.c_n}{a} \ ,$$

where $\neg.c_i$ is the opposite of c_i (i.e., $\neg.a = \neg a$ and $\neg.\neg a = a$).

Theorem 9. *Let P be an extended logic program and let $T = (\emptyset, \{d(C) \mid C \in P\})$ be the corresponding default theory. Then, M is an answer set of P if and only if $E = Cn(M)$ is a consistent default extension of T.*

Thus, extended logic programs under answer set semantics can be regarded as a fragment of default logic.

5.3 Disjunction

The next extension to normal logic programs is disjunctions in rule heads. The use of disjunction is natural to express indefinite knowledge. For instance, the rule

$$\mathit{female}(X) \vee \mathit{male}(X) \leftarrow \mathit{person}(X)$$

expresses that all persons are either female or male. Another example is the disjunctive fact

$$\mathit{broken}(\mathit{left\_hand}, \mathit{tom}) \vee \mathit{broken}(\mathit{right\_hand}, \mathit{tom}) \leftarrow \ ,$$

which expresses that tom has a broken arm, but it is unknown whether the left or the right hand is broken.

Disjunctive information is a natural extension for expressing a "guess" and to create non-determinism in logic programs, like in the rule

$$\mathit{ok}(C) \vee -\mathit{ok}(C) \leftarrow \mathit{component}(C) \ ,$$

which states that a component may be in a working condition or not working at all, and in each of the alternative states of the component, we might have to take different actions for solving our problem.

The semantics of disjunctive rules is such that we conclude one of the alternatives to be true (the minimality principle).

In the next example, we look at different disjunctive programs, and the models they admit.

Example 30. Disjunction is *minimal*, i.e., from a rule, we usually infer only a single atom "at a time." The single rule program

$$a \vee b \vee c \leftarrow \tag{15}$$

has three minimal models: $\{a\}$, $\{b\}$, and $\{c\}$. There exist no smaller models for (15), since $\emptyset$ is not a model. The interpretation $I = \{a, b\}$ for instance is a model of this program, but both $\{a\}$ and $\{b\}$ are smaller than I and satisfy (15), hence I is not a minimal model.

If we take a closer look into the minimal models of disjunctive programs, we observe that they are actually *subset minimal*. Take, for instance, the program

$$a \vee b \leftarrow \tag{16}$$

$$a \vee c \leftarrow \tag{17}$$

This program has two minimal models: $\{a\}$ and $\{b, c\}$. The interpretation $J = \{a, b, c\}$ is a model for both (16) and (17), hence it is a model of the program. But J is a proper superset of $\{b, c\}$, thus J is not a minimal model. Note that $\{a\}$ is the only singleton minimal model, as both $\{b\}$ and $\{c\}$ do not satisfy the program.

In a similar vein, the program

$$a \vee b \leftarrow \tag{18}$$

$$a \leftarrow b \tag{19}$$

has the single minimal model $\{a\}$, as the model $\{a, b\}$ is not minimal with respect to set inclusion. The interpretation $\{b\}$ is not a model; it satisfies rule (18), but it is not a model for (19).

Note that disjunction should not be understood as *exclusive*. Take program

$$a \vee b \leftarrow \tag{20}$$
$$b \vee c \leftarrow \tag{21}$$
$$a \vee c \leftarrow \tag{22}$$

which has three minimal models $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$. Each of the three minimal models are not contained in the other, but the intersection of any two of the minimal models is nonempty.

Let us next consider the use for disjunctive rules vs. unstratified negation. Going back to our Dilbert scenario, the program

$$man(dilbert). \tag{23}$$
$$single(X) \leftarrow man(X), \text{not } husband(X). \tag{24}$$
$$husband(X) \leftarrow man(X), \text{not } single(X). \tag{25}$$

which expresses that a man is either a single or a husband, is equivalent to the disjunctive program

$$man(dilbert). \tag{26}$$
$$single(X) \vee husband(X) \leftarrow man(X). \tag{27}$$

Here, the use of disjunction is more intuitive. In fact, one can see the rule (27) resulting from (24) resp. (25) by "shifting" the negated literal $\text{not } husband(X)$ (resp. $\text{not } single(X)$) to the head (classically, the clauses are equivalent). While such shifting works in this example, as well as under certain syntactic conditions (like headcycle-freeness) [39], we note that in general, disjunctive rule heads are not syntactic sugar for unstratified negation; this is also evidenced by complexity results provided in Section 5.3.

Extended Logic Programs with Disjunctions. The extension of ELPs with disjunctive rule heads leads to the class of extended disjunctive logic programs in [11].

Definition 15. *A* extended disjunctive logic program *(EDLP) is a finite set of rules*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \qquad (k, m, n \geq 0)$$

where all a_i, b_j, and c_l are atoms or strongly negated atoms.

The semantics for an EDLP can be defined similarly as for an extended logic program, with the only difference being that instead of choosing a stable model M of P (i.e., M is the least model of the reduct P^M), we define an *answer set* M of an EDLP P as a *minimal model* M of the reduct P^M, since multiple minimal models of P^M might exist.

Example 31. Consider the program P:

$$man(dilbert).$$
$$single(X) \vee husband(X) \leftarrow man(X).$$

There are two answer sets for P:

- $M_1 = \{man(dilbert), single(dilbert)\}$, and
- $M_2 = \{man(dilbert), husband(dilbert)\}$.

Please note that P is "not"-free (positive), hence the reduct $grnd(P)^M = grnd(P)$ for every interpretation M.

It is worth mentioning here that answer sets of EDLPs can also be nicely defined in equilibrium logic [40, 41], which is a non-monotonic version of the logic of here and there, an intermediate logic between classical logic and intuitionistic logic. This logic is well-suited to capture not only EDLPs, but also other extensions of normal logic programs.

Semantic Properties of Disjunctive ELPs. The extensions of normal logic programs to ELPs and DLPs considered in this section inherit most of the alluring properties of stable models, which have been shown in Section 4.3.

We now define Herbrand interpretations to EDLPs. Since an extended logic program may contain atoms under classical negation, an interpretation for EDLPs may also contain strongly negated ground atoms, i.e., literals of form a or $-a$. But this means that an interpretation can be inconsistent if it contains both a and $-a$. In [11], the inconsistent answer set has been defined as the interpretation which contains all possible atoms and their strongly negated counterparts. For our purposes, we deal only with consistent interpretations and thus disregard the inconsistent answer set. We define models as follows.

Definition 16. *A interpretation I is a* model *of*

- *a ground clause C : $a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \mathrm{not}\ c_1, \ldots, \mathrm{not}\ c_n$, denoted $I \models C$, if either $\{b_1, \ldots, b_m\} \nsubseteq I$ or $\{a_1, \ldots, a_k, c_1, \ldots, c_n\} \cap I \neq \emptyset$;*
- *a clause C, denoted $I \models C$, if $I \models C'$ for every $C' \in grnd(C)$;*
- *a program P, denoted $I \models P$, if $I \models C$ for every clause C in P.*

The above definition takes all of our extensions into account: (i) constraints do not have head literals, hence $k = 0$ and only the body part (the b_i, c_j) is taken into account; (ii) rules with strong negation are considered by viewing all a_i, b_j, c_l in I as classical literals; as well as (iii) disjunctive rules (where $n > 1$), with the meaning that if the rule body is satisfied, at least some literal a_i, $1 \leq i \leq n$, must be true. In a sense, such interpretations represent three-valued states: a ground atom a is regarded *true* in I, if $a \in I$, while a is regarded *false* in I, if $-a \in I$; of neither a nor $-a$ is contained in I, then a is *unknown* in I.

Similar to stable models of normal logic programs, a (disjunctive) ELP P may have no, one, or multiple answer sets, which are models of P, and, in fact, minimal models of P.

Theorem 10. *Let P be a (disjunctive) ELP and M be an answer set of P.*

1. *M is a model of P.*
2. *M is a minimal model of P.*

Hence, just like least models of positive programs and stable models of normal programs, an answer set satisfies all rules of an EDLP. Moreover, an answer set is a minimal model of the program, which intuitively means that it contains only the absolutely necessary bare minimum of facts in order to satisfy a program.

Corollary 2. *If M_1 and M_2 are two different answer sets of P then $M_1 \not\subseteq M_2$ and $M_2 \not\subseteq M_1$.*

Similarly to stable models for normal logic programs, one can define unfounded sets for answer sets of of EDLPs to address the problem of self-supported literals. Leone et al. [28] did this for programs without strong negation.

Definition 17 ([28]). *Given an EDLP P without strong negation and an interpretation I, a set $U \subseteq HB_P$ is an* unfounded set *of P relative to an interpretation I, if for every $a \in U$ and every $r \in ground(P)$ such that a appears in the head of r, at least one of the conditions hold:*

1. *There is a literal b appearing in the positive body of r such that either $b \notin I$ or $b \in U$;*
2. *There is a literal b appearing in the negative body of r such that $b \in I$; or*
3. *There is a literal b appearing in the head of r such that $b \notin U$ and $b \in I$.*

Unlike for normal logic programs, we cannot guarantee the existence of a greatest unfounded set for disjunctive programs relative to an interpretation. But there exist interpretations for an EDLP, where the existence of a greatest unfounded set is guaranteed: the unfounded-free interpretations. We call an interpretation I for an EDLP P *unfounded-free*, if $I \cap U = \emptyset$ for each unfounded set U for P w.r.t I.

Theorem 11 ([28]). *Given an EDLP program P without strong negation, an interpretation M of P is an answer set iff M is* unfounded-free.

The `DLV` system heavily relies on unfounded sets as its underlying principle to build the answer sets of EDLPs (see Section 7.2 for more details).

Note that the notion of unfounded set is easily extended to answer sets of EDLPs with strong negation, by adding the respective constraints; [42] defines unfounded sets directly in Equilibrium Logic.

A recent development in the ASP area is a syntactic counterpart of unfounded sets: *loop formulas* [43–45]. These formulas have been conceived as a way to transform logic programs under stable and answer set semantics to propositional theories, and let standard SAT solvers perform the task of computing the stable models of these extended theories. In a nutshell, this translation uses Clark's completion [46] for logic programs to create a propositional theory and augment this theory by additional loop formulas, which guarantee that this theory admits only stable models. Note that in general there can be exponentially many loop formulas for a given EDLP [47].

We end this section by looking into reasoning with answer sets, which is defined just as reasoning with stable models: a classical literal a is a (i) *brave (credulous) consequence* of program P, $P \models_b a$, iff $M \models_b a$ for some answer set M of P; and (ii) *cautious (skeptical) consequence* of a program P, $P \models_c a$, iff $M \models_c a$ for all answer sets M of P. The behaviour with respect to properties like cautious monotonicity and cumulativity is then similar as in the disjunction-free case.

Computational Properties of Disjunctive ELPs. Similar to normal logic programs under stable model semantics, EDLPs under answer set semantics have many interesting computational tasks, and a particular one is testing whether a program P is consistent, i.e., whether P has some answer set. Here, we restrict our attention to the consistency problem, give complexity results for various classes of EDLPs, and briefly sketch proofs or give ideas how such a proof can look like. Let us start with the general case.

Theorem 12 ([48]). *Deciding whether a given ground disjunctive program P has some answer set is Σ_2^p-complete in general.*

Recall that $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$ is the class of problems decidable in polynomial time on a nondeterministic Turing machine with an oracle for solving problems in NP [49].

The membership of consistency of disjunctive ELPs can be shown by the following argument: we first guess an answer set M for a program P, and verifying whether M is a minimal model of P^M is in co-NP (note that P^M can be computed in polynomial time), thus decidable with one call to an NP-oracle.

The intuition for the hardness part is as follows: we have to create a reduction from validity of a quantified Boolean formula of the form $\exists X \forall Y E(X, Y)$ to an EDLP P, where $E(X, Y)$ is in disjunctive normal form and X and Y are the (lists of) variables occurring in E. For a detailed proof, we refer the reader to [48].

But there exist subclasses of EDLPs with lower computational complexity. For instance, testing whether a strictly positive disjunctive ELPs has an answer set is easy, since each positive program has a model and the Gelfond-Lifschitz reduct does not change the given program.

For a ground DLP P with constraints, but without "not" in the rule bodies, deciding whether P has some answer set is NP-complete. Hardness can be shown by a reduction from SAT: given a propositional CNF-formula ϕ, we transform ϕ into a positive disjunctive program P with constraints by adding rules $a \vee \bar{a} \leftarrow$ for each atom a in ϕ and adding the "negation" C' of each clause C in ϕ as a rule of form $u \leftarrow C'$ to P, where u is a fresh symbol (e.g., for a clause $a \vee b \vee \neg c$ we add the rule $u \leftarrow \bar{a}, \bar{b}, c$), and finally add the constraint $\leftarrow u$; then, ϕ is satisfiable iff P has an answer set. Membership can be shown by guessing an interpretation for the positive part of the program, and checking if the interpretation is a minimal model of the positive part and is compliant with the constraints in polynomial time.

Other classes of EDLPs exist which obtain lower complexity, for instance, deciding consistency of headcycle-free EDLPs [39] is also NP-complete.

Similarly to normal logic program, we obtain an exponential blowup for nonground EDLPs compared to the propositional case. In particular, verifying whether a nonground

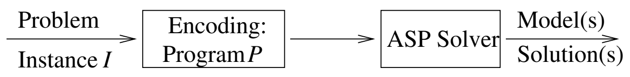


Fig. 6. Encoding of problems in ASP

EDLP has some answer set is $\mathrm{NEXP}^{\mathrm{NP}}$-complete, i.e., complete for the class of problems that run in exponential time on a nondeterministic Turing machine and have access to an NP-oracle (see also [12]). If function symbols are allowed, the complexity does not increase through disjunction in general (cf. [12]); syntactic restrictions are known under which the complexity of deciding consistency stays is 2-EXP-complete [32] and 3-EXP-complete [33], respectively.

6 Answer Set Programming Paradigm

In this section, we now turn to the Answer Set Programming (ASP) paradigm, which emerged from the nonmonotonic Logic Programming area at the end of the 1990s. There were, as already mentioned, several texts in which this paradigm was proposed, [50, 3–5]; after the LPNMR 1999 conference, a special issue of the AI Journal was edited [2] covering the subject, a dedicated ASP workshop series started in 2001 [1]. The textbook by Baral [6] was then a further step to disseminate this approach.

Let us first start with more motivation by outlining the general idea behind answer set programming: given an instance of a (search) problem I and its corresponding representation in form of a logic program P, we may perceive the *models of* P as *solutions for* I. That is, in ASP, we view problem solving tasks as computing the models of their matching encoded programs. This view gives rise to a general strategy for implementing any kind of problem solving task, shown graphically in Figure 6:

1. we *encode* our problem instance I as a (nonmonotonic) logic program P, such that solutions of I are represented by models of P; and then
2. *compute* some model M of P, by using an AS solver of our choice; and finally
3. *extract* a solution for I from M.

We may vary this strategy by allowing to compute more than one solution, which intuitively corresponds to obtaining multiple or even all solutions for our problem instance I.

This method has been successfully applied to numerous problems in a range of areas; an incomplete list is

– diagnosis
– information integration
– constraint satisfaction
– reasoning about actions (including planning)
– routing, and scheduling
– phylogeny construction
– security analysis

– configuration
– computer-aided verification
– health care
– biomedicine and biology
– Semantic Web
– knowledge management
– text mining and classification
– question answering

The survey [51] is a source for specific applications, some of which can be viewed in an online showcase collection.[16]

We illustrate the ASP approach on the problem of computing legal 3-colorings of a graph.

Example 32. Let $G = (V, E)$ be a graph with nodes $V = \{a, b, c, d\}$ and edges $E = \{(a, b), (b, c), (c, a), (a, d)\}$, which constitutes our problem instance I. We can encode the legal three colorings of G into answer sets of a logic program P as follows. For each node n, we have atoms b_n, r_n, and g_n which informally mean that node n is colored blue, red, and green, respectively. Then we set up the following rules. For each node $n \in V$,

$$\begin{aligned} &b_n \leftarrow \text{not } r_n, \text{not } g_n. \\ &r_n \leftarrow \text{not } b_n, \text{not } g_n. \\ &g_n \leftarrow \text{not } r_n, \text{not } b_n. \end{aligned}$$

and for each edge (n, n') in E, the constraints

$$\begin{aligned} &\leftarrow b_n, b_{n'}. \\ &\leftarrow r_n, r_{n'}. \\ &\leftarrow g_n, g_{n'}. \end{aligned}$$

Then, the answer sets of P encode 1-1 the legal 3-colorings of G. Informally, the rules for $n \in V$ assign one of the three colors to n, and the constraints for (n, n') check that adjacent nodes do not have the same color. Equally well, we can replace the three rules for $n \in V$ by the single (and perhaps more intuitive) rule

$$b_n \vee r_n \vee g_n \leftarrow .$$

This problem solving strategy is closely related to similar approaches like SAT-solving, where the problem instance is encoded onto the (classical) models of a propositional formula of clause set. It is because of this that some authors refer to Answer Set Programming as the more general paradigm in which a problem is encoded into the models of a logical theory, and consider the usage of nonmonotonic logic programs as theory as a particular instance of this paradigm. We prefer here, however, to reserve the term ASP for the setting with nonmonotonic logic programs under the answer set semantics itself.

[16] `http://www.kr.tuwien.ac.at/research/projects/WASP/showcase.html`

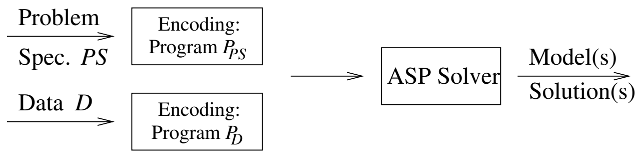


Fig. 7. Uniform problem encoding in ASP

Compared to SAT solving, ASP provides features that are not available there. For example, the transitive closure of a given graph G (and its complement) is expressible within an answer set, which is cumbersome in classical propositional logic. Here, one can exploit negation as failure fruitfully. Furthermore, ASP offers many constructs besides negation as failure, and, importantly, allows also problem descriptions with predicates and variables. This can be utilized for generic problem solving where the specification of solutions (the "logic" PS) is separated from the concrete instance of the problem at hand (the "data" D, usually given as facts); see Figure 7.

Example 33. In the graph 3-coloring problem, assuming that $G = (V, E)$ is stored using facts $node(n)$ for each $n \in V$ and $edge(n, n')$ for each $(n, n') \in E$, which gives the data D, the generic specification of solutions PS can be given by the following rules:

$$b(X) \leftarrow node(X), \text{not } r(X), \text{not } g(X).$$
$$r(X) \leftarrow node(X), \text{not } b(X), \text{not } g(X).$$
$$g(X) \leftarrow node(X), \text{not } r(X), \text{not } b(X).$$

and the constraints

$$\leftarrow b(X), b(Y), edge(X, Y).$$
$$\leftarrow r(X), r(Y), edge(X, Y).$$
$$\leftarrow g(X), g(Y), edge(X, Y).$$

Similarly as above, we can use the single disjunctive rule

$$b(X) \vee r(X) \vee g(X) \leftarrow node(X).$$

instead of the three unstratified rules defining $b(X)$, $r(X)$, and $g(X)$. Then, the answer sets of $PS \cup D$ correspond to the legal 3-colorings of G.

The efficient evaluation of ASP programs requires the integration of techniques from the areas of Knowledge Representation, Database, and Search, as language constructs and features need to be handled (possibly by compiling them away), (larger) input volumes of data need to be processed, and nondeterminism as it occurs with unstratified negation has to handled with search.

In the rest of this section, we will briefly discuss some declarative programming techniques that are used in ASP. There is a variety of "design patterns" which depend on the type of problem to be solved and the language elements that are used or needed. We discuss here the use of four techniques: (i) double negation as a technique to compute maximal elements in a set, which can done using stratified negation; (ii) a general guess and check methodology which uses unstratified negation or disjunction to generate and prune solutions candidates; (iii) an advanced technique called *saturation*, which can be used in disjunctive logic programming to test properties of various subsets of a set within an answer set, and is essential to solve "hard" problems there (cf. [48, 52]); (iv) in combination with this, iteration over a set to test whether a property holds for all elements without the use of negation.

6.1 Use of Double Negation

The first technique which we look at is the use of double negation. In classical logic $\neg\neg A \equiv A$, i.e., double negation can be canceled. This can be similarly exploited in ASP to define a predicate $p(X)$ in terms of its complement $-p(x)$, and is particularly attractive if $-p(x)$ can be defined easily. For example, one can avoid counting and arithmetic for determining the maximum in a (finite) set of numbers.

Example 34. Suppose the data about employees of a company and their salaries are stored as facts $empl(N, S)$ in the data D, where N is the name and S the salary of an employee. Then the maximum salary, $s^* = \max\{s \mid empl(e, s) \in D\}$, is determined by the following simple ASP program:

$\%$ salary S is *not* maximal
$-max(S) \leftarrow empl(N, S),\ empl(N_1, S_1), S < S_1.$
$\%$ double negation
$max(S) \leftarrow empl(N, S), \mathrm{not}\ -max(S).$

Example 35. For a little more involved example where this technique can be used successfully, consider the problem of computing the greatest common divisor (gcd) of two natural numbers $n, m > 0$; recall that the gcd of n and m is the largest integer d^* such that d^* divides both n and m. This problem is a standard example in logic programming and elegant solutions for it can be found in textbooks, which basically implement Euclid's recursive algorithm for it by rules:

$\%$ base case
$gcd(X, X, X) \leftarrow int(X), X > 1.$
$\%$ subtract smaller from larger number
$gcd(D, X, Y) \leftarrow X < Y, gcd(D, X, Y_1), Y = Y_1 + X.$
$gcd(D, X, Y) \leftarrow X > Y, gcd(D, X_1, Y), X = X_1 + Y.$

Here, $int(X)$ is a (built in) predicate for natural numbers ≥ 0. While Euclid's algorithm is ingenious, the average programmer will approach the problem by trying to cast the definition into rules. Here, double negation can be used again to single out the maximal common divisor d^* in the predicate $gcd(X, Y, Z)$ given that the common divisors are computed in a predicate $cd(X, Y, Z)$, which in turn can be done easily using a predicate $divisor(X, Y)$ that is defined using simple (built-in) arithmetic. A respective program is the following:

$\%$ Declare when D divides a number N.
$divisor(D, N) \leftarrow int(D), int(N), int(M), N = D * M.$
$\%$ Declare common divisors

$cd(T, N_1, N_2) \leftarrow divisor(T, N_1), divisor(T, N_2).$
% Single out non-maximal common divisors T
$-gcd(T, N_1, N_2) \leftarrow cd(T, N_1, N_2), cd(T_1, N_1, N_2), T < T_1.$
% Apply double negation: take non non-maximal divisor
$gcd(T, N_1, N_2) \leftarrow cd(T, N_1, N_2), not\ -gcd(T, N_1, N_2).$

For a similar encoding in Prolog, one has to be careful to define and use $int(X)$ properly in the rules, otherwise the program might not terminate.

Note that the above programs are both stratified and thus have a single answer set over any input data (provided that some answer set exists). We will next consider programs that are geared toward multiple answer sets for capturing problems with multiple solutions.

6.2 The "Guess and Check" Methodology

An important element of ASP is to employ a "Guess and Check" methodology, which is sometimes also called Generate-and-Test [3]. The idea is here to proceed as follows:

1. use nondeterminism that comes with unstratified negation, or equally well with disjunction in rule heads, to create candidate solutions to a problem (program part $\mathcal{G}$), and
2. to check with further rules and/or constraints, whether a solution candidate is proper (program part $\mathcal{C}$). This part may also involve auxiliary predicates, if needed.

From another perspective, the part $\mathcal{G}$ defines the search space, and the part $\mathcal{C}$ prunes illegal branches. A detailed discussion of this paradigm is given in [53, 52], and in [53] it is extended by a further component to compute optimal solutions (we will deal with this in Section 7.2 below). We will just briefly illustrate the methodology on a few examples.

Example 36. As a first example, we revisit the 3-colorability problem in Example 33.

$$g(X) \vee r(X) \vee b(X) \leftarrow node(X) \;\Big\}\; \textbf{Guess}$$

$$\left.\begin{array}{l} \leftarrow b(X), b(Y), edge(X, Y) \\ \leftarrow r(X), r(Y), edge(X, Y) \\ \leftarrow g(X), g(Y), edge(X, Y) \end{array}\right\} \textbf{Check}$$

The first disjunctive rule constitutes the guessing part $\mathcal{G}$, which generates all possible assignments of one colors to the nodes of the graph, while the three constraints constitute the checking part $\mathcal{C}$.

The next example shows a checking part which uses auxiliary predicates.

Example 37. Recall that for a directed graph $G = (V, E)$, a path $n_0 \rightarrow n_1 \rightarrow \cdots \rightarrow n_k$ in G from a start node $n_0 \in V$ is called a *Hamiltonian path*, if all nodes n_i are distinct and each node in V occurs in the path, i.e., $V = \{n_0, \ldots, n_k\}$. Assume that, as above, the graph G is stored using the predicates $node(X)$ and $edge(X, Y)$, and that a predicate $start(X)$ stores the unique node n_0. Consider the following program:

$$\left.\mathit{inPath}(X,Y) \vee \mathit{outPath}(X,Y) \;\leftarrow\; \mathit{edge}(X,Y).\right\}\textbf{Guess}$$

$$\left.\begin{array}{l}\leftarrow \mathit{inPath}(X,Y),\ \mathit{inPath}(X,Y_1),\ Y \neq Y_1.\\ \leftarrow \mathit{inPath}(X,Y),\ \mathit{inPath}(X_1,Y),\ X \neq X_1.\\ \leftarrow \mathit{node}(X),\ \text{not}\ \mathit{reached}(X).\end{array}\right\}\textbf{Check}$$

$$\left.\begin{array}{l}\mathit{reached}(X) \leftarrow \mathit{start}(X).\\ \mathit{reached}(X) \leftarrow \mathit{reached}(Y),\ \mathit{inPath}(Y,X).\end{array}\right\}\textbf{Auxiliary Predicate}$$

The guessing part $\mathcal{G}$ simple states for each edge of the graph whether it belongs to the path or not. The checking part $\mathcal{C}$ tests whether inPath really constitutes a path in G in which each node occurs only once (which is ensured if there is at most one edge from/to each node), and that all nodes are on the path. For this, the auxiliary predicate $\mathit{reached}(X)$ is used, which expresses that the node X is reached from the starting node. The latter is expressed with two simple recursive rules.

Note that deciding the existence of a Hamiltonian path is, like 3-colorability, NP-complete; a similar SAT encoding would be, due to the reachability check, more cumbersome.

As a final example, we consider a scenario where the checking part is interfering with the guessing part, which shows that the two parts may not always be cleanly separated. In fact, this happens for the elementary task of choosing an element from a set.

Example 38. Suppose departments of a company are stored in a predicate $\mathit{dept}(X)$, and the task is to choose a single department; in general, there will be multiple choices (or none, if dept would be empty). The following program is a simple (yet little elegant) solution to the problem:

$$\left.\mathit{sel}(D) \vee -\mathit{sel}(D) \leftarrow \mathit{dept}(D)\right\}\textbf{Guess}$$

$$\left.\begin{array}{rl} & \leftarrow \mathit{sel}(D_1), \mathit{sel}(D_2), D_1 \neq D_2\\ \mathit{some\_sel} & \leftarrow \mathit{sel}(D)\\ & \leftarrow \mathit{dept}(D), \text{not}\ \mathit{some\_sel}\end{array}\right\}\textbf{Check}$$

Here, the checking part tests that not more than one department has been chosen, and that at least one is chosen if there are departments (hence, exactly one is chosen).

A more elegant solution is to let the checking part interfere with the guessing part, and to exploit the minimality property of answer sets.

$$\begin{array}{ll}\mathit{sel}(D) \leftarrow \mathit{dept}(D), \text{not}\ -\mathit{sel}(D) & \}\ \textbf{Guess}\\ -\mathit{sel}(D_1) \leftarrow \mathit{dept}(D_1), \mathit{sel}(D_2), D_1 \neq D_2 & \}\ \textbf{Check}\end{array}$$

The guessing rules informally states that, by default, a department D is chosen. The checking rule says that if some department is chosen, then all others can not be chosen; this is fed back to the guessing part. In combination, since not all departments have to be excluded from selection, exactly one will be chosen in an answer set (provided some departments exist, otherwise $\emptyset$ is the single answer set). In other words, the only stable configurations of the above program are those in which one and only one atom of type $\mathit{sel}(v)$ is present.

Note that we could equally well replace the checking rule with the rule

$$-sel(D_1) \vee -sel(D_2) \leftarrow dept(D_1), dept(D_2), D_1 \neq D_2.$$

Informally, this rule says that if we have two different departments, then at least one of them can not be selected.

As a final example for choice, we consider a simple course scheduling scenario.

Example 39. Suppose there is a computer science department cs at a university u. We have information about members and courses of cs, as well as preferred courses of members, both encoded as facts F:

$$\begin{array}{rrr}
member(sam, cs). & course(java, cs). & course(ai, cs). \\
member(bob, cs). & course(c, cs). & course(logic, cs). \\
member(tom, cs). & & \\
likes(sam, java). & likes(sam, c). & \\
likes(bob, java). & likes(bob, ai). & \\
likes(tom, ai). & likes(tom, logic). &
\end{array}$$

Our task is now to assign each member of the department some courses, such that (i) each member should have at least one course, (ii) nobody should have more than two courses, and (iii) assign only courses that the course leader likes. We can use the following program P to encode this problem:

$$\begin{array}{rl}
teaches(X,Y) \leftarrow & member(X, cs), course(Y, cs), likes(X, Y), \\
& \text{not } -teaches(X, Y). \\
-teaches(X,Y) \leftarrow & member(X, cs), course(Y, cs), teaches(X_1, Y), X_1 \neq X. \\
some\_course(X) \leftarrow & member(X, cs), teaches(X, Y). \\
\leftarrow & member(X, cs), \text{not } some\_course(X). \\
\leftarrow & teaches(X, Y_1), teaches(X, Y_2), teaches(X, Y_3), \\
& Y_1 \neq Y_2, Y_1 \neq Y_3, Y_2 \neq Y_3.
\end{array}$$

Informally, the first rule says that a CS faculty member gets a CS course she likes assigned by default. The second rule states that a CS faculty member does not get a CS course assigned if somebody else teaches it. The third and fourth rules make sure that each CS faculty gets at least one course assigned. The final rule excludes any assignment where one person is assigned three (or more) courses.

We obtain the following three answer sets of $P \cup F$:

– $\{teaches(sam, c), teaches(bob, java), teaches(bob, ai), teaches(tom, logic), \ldots\}$
– $\{teaches(sam, java), teaches(sam, c), teaches(bob, ai), teaches(tom, logic), \ldots\}$
– $\{teaches(sam, c), teaches(bob, java), teaches(tom, ai), teaches(tom, logic), \ldots\}$

6.3 Saturation Technique

A more advanced technique that is used in disjunctive ASP is the so called *saturation technique*, which is used to check whether *all* possible guesses satisfy a certain property, like *not* being a solution to a problem. Testing such a property, like whether all assignments of three colors to nodes do *not* legally color a graph G, may be co-NP-hard, and thus can not be evidently encoded in a normal logic program such that the program has some answer set precisely if G is *not* 3-colorable; in fact, the program in Example 33 has *no* answer set if G is *not* 3-colorable.

It is, however, possible to express the property of non-3-colorability by a *unique answer set candidate* for a program, such that the candidate is the only answer set if the graph is not 3-colorable, and is not an answer set otherwise. More abstractly, to test a property we design a program P and an answer set candidate M_{sat} such that M_{sat} is the single answer set of P if the property holds, and P has other answer sets (excluding M_{sat}) otherwise. The construction is such that any answer set of P is a subset of M_{sat}, and whenever the property is found to hold, any candidate answer set is "saturated" to M_{sat}. Intuitively, the property is tested *within* the answer set.

Example 40. For testing non-3-colorability, the constraints in the checking part of the program in Example 33 can be replaced, thus obtaining the program P_{non_col}:

$$\begin{array}{l} b(X) \vee r(X) \vee g(X) \leftarrow node(X). \\ non\_col \leftarrow r(X), r(Y), edge(X,Y). \\ non\_col \leftarrow g(X), g(Y), edge(X,Y). \\ non\_col \leftarrow b(X), b(Y), edge(X,Y). \\ \chi(X) \leftarrow non\_col, node(X). \end{array}$$

where $\chi \in \{r, g, b\}$. Informally, this change has the following effect: Whenever an assignment of colors to the nodes is bad, the assignment is rejected by "saturating" the candidate model at hand, selecting all ground facts $r(n)$, $g(n)$, and $b(n)$ for any node n. Importantly, the saturation is the same for all bad assignments. Thus, if *all* assignments of colors are bad, there will a be single answer set M_{sat} of the program, which contains, besides the graph description, non_col and $r(n)$, $g(n)$, and $b(n)$ for any node n. On the other hand, any good assignment of colors will lead to an answer set M such that $M \subset M_{sat}$, which means that M_{sat} is not an answer set of the program, and that P_{non_col} has many answer sets, smaller than M_{sat} corresponding to valid 3-colorings. Thus, M_{sat} is the single answer set of the program just if the graph is not 3-colorable. Note also that $P_{non\_col} \models_c non\_col$ iff the graph at hand is not 3-colorable.

We can abstract from the previous example a general design rule: if we desire to check that a property *Pr* holds *for all guesses* defining a search space, we can establish a guess and saturation check paradigm as follows:

- Define the search space of guesses through a subprogram P_{guess}, using disjunctive rules.
- Define a subprogram P_{check}, which checks *Pr* for a guess M_g.
- If *Pr* holds for M_g, an appropriate set of saturation rules P_{sat} generates the special candidate answer set M_{sat}.
- If *Pr* does not hold for M_g, an answer set results which is a *strict subset* of M_{sat} (thus preventing that M_{sat} is an answer set).

It is thus crucial that the program P_{check}, which formalizes *Pr*, and P_{sat} do not generate incomparable candidate answer sets. Incomparability might be easily introduced, besides subprograms with negation as failure, by improper use of disjunction, or by ill-designed (positive) saturation rules; we will see an example in the next subsection.

In combination with further guessing rules, which assign values to atoms that are not involved in saturation, it is possible to express problems that have complexity beyond NP, like the strategic companies problem [53, 52], or quantified Boolean formulas of the form $\exists X \forall Y E(X,Y)$, which are Σ_2^p-complete.

6.4 Iteration over a Set

As last technique, we consider testing a property for all elements of a set without the use of negation. This may be needed in some contexts, for instance in combination with the saturation technique, or when the use of negation could lead to undesired behavior (e.g., in case of cyclic negation).

Example 41. Suppose we want to test whether in a directed graph $G = (V, E)$, all nodes are reachable from a designated start node $n_0 \in V$. Using the representation of G and n_0 as in Example 37, we could use the following rules and double negation:

$$\begin{array}{l}
\mathit{all\_reached} \leftarrow \text{not } -\mathit{all\_reached}.\\
-\mathit{all\_reached} \leftarrow \mathit{node}(X), \text{not } \mathit{reached}(X).\\
\mathit{reached}(X) \leftarrow \mathit{start}(X).\\
\mathit{reached}(X) \leftarrow \mathit{reached}(Y),\ \mathit{edge}(Y,X).
\end{array}$$

Here, $\mathit{all\_reached}$ is true in the resulting answer set, exactly if all nodes are reachable from n_0.

Now suppose that we want to test in an answer set whether reachability holds for each graph G' that results from G by removing between all nodes n and n', that are mutually connected, one of the edges $n \to n'$, $n' \to n$ at random. The edges of G' can be generated using the rules

$$\begin{array}{l}
\mathit{edge}_1(X,Y) \vee \mathit{edge}_1(Y,X) \leftarrow \mathit{edge}(X,Y), \mathit{edge}(Y,X).\\
\mathit{edge}_1(X,Y) \leftarrow \mathit{edge}(X,Y), \text{not } \mathit{edge}(Y,X).
\end{array}$$

Let us replace edge in the rules for $\mathit{reached}$ with edge_1 and add the saturation rule

$$\mathit{edge}_1(X,Y) \leftarrow \mathit{all\_reached}, \mathit{edge}(X,Y).$$

We thus obtain the program P_g:

$$\begin{array}{l}
\mathit{all\_reached} \leftarrow \text{not } -\mathit{all\_reached}.\\
-\mathit{all\_reached} \leftarrow \mathit{node}(X), \text{not } \mathit{reached}(X).\\
\mathit{reached}(X) \leftarrow \mathit{start}(X).\\
\mathit{reached}(X) \leftarrow \mathit{reached}(Y),\ \mathit{edge}(Y,X).\\
\mathit{edge}_1(X,Y) \vee \mathit{edge}_1(Y,X) \leftarrow \mathit{edge}(X,Y), \mathit{edge}(Y,X).\\
\mathit{edge}_1(X,Y) \leftarrow \mathit{edge}(X,Y), \text{not } \mathit{edge}(Y,X).\\
\mathit{edge}_1(X,Y) \leftarrow \mathit{all\_reached}, \mathit{edge}(X,Y).
\end{array}$$

However P_g does not work as expected, as evidenced, e.g., by the simple graph $G = (\{a, b\}, \{a \to b, b \to a\})$, where for $n_0 = a$ we get that the candidate

% Guess a subgraph for testing
$edge_1(X,Y) \vee edge_1(Y,X) \leftarrow edge(X,Y), edge(Y,X).$
$edge_1(X,Y) \leftarrow edge(X,Y), \text{not } edge(Y,X).$
% Compute all reachable nodes
$reached(X) \leftarrow start(X).$
$reached(X) \leftarrow reached(Y),\ edge_1(Y,X).$
% iterate to check if all nodes are reached
$all\_reached \leftarrow last(X), all\_reached\_upto(X).$
$all\_reached\_upto(X) \leftarrow all\_reached\_upto(Y), succ(Y,X), reached(X).$
$all\_reached\_upto(X) \leftarrow first(X), reached(X).$
% Saturation rule
$edge_1(X,Y) \leftarrow all\_reached, edge(X,Y).$

Fig. 8. Program with reachability test for subgraphs in an answer set

$$\begin{aligned} M_{sat} = \{\ & all\_reached, edge_1(a,b), edge_1(b,a), reached(a), reached(b), \\ & edge(a,b), edge(b,a), node(a), node(b), start(a)\} \end{aligned}$$

is a "saturated" answer set, while the property fails for G (for a witnessing G', remove $a \to b$; we refer to this graph as $G_{-(a\to b)}$). Indeed, P has also an answer set $M_2 = \{-all\_reached, edge_1(b,a), reached(a), edge(a,b), \ldots\}$, which is not a subset of the saturation candidate, corresponding to the graph $G_{-(a\to b)}$.

This apparently non-obvious behaviour can be explained from several perspectives: M_{sat} is a proper answer set due to the fact that $G_{-(b\to a)}$ reaches all possible nodes from a, thus making $all\_reached$ true. Consequently, the saturation rule makes the extension of $edge_1$ equal to $edge$, which results in M_{sat}. On the other hand, one might expect that M_2, which corresponds to the deletion of the edge $a \to b$, should invalidate M_{sat}, and thus obtain M_2 as (the single) answer set. But M_2 *is not* contained in M_{sat}. Indeed, although M_2 is not a saturated answer set, and although the extension of $edge_1$ in M_2 is a strict subset of $edge$, it contains the "spare" atom $-all\_reached$, which does not appear in M_{sat}, making the two answer sets incomparable.

We are thus in a situation in which M_{sat} is an answer set, resulting from *some* of the guessed subgraphs G' of G in which reachability is retained, while we expected M_{sat} as an answer set if and only if reachability holds *for all* guessed subgraphs G' of G.

The problems of program P_g can be remedied by using recursive positive rules—which check whether each node is reachable—instead of double negation. This will help in establishing a "proper" containment between candidate answer sets and M_{sat}. To this end, an ordering of the nodes is taken, and associated successor predicates $\mathit{first}(X)$, $succ(Y,X)$, and $last(X)$ which express that X is the first node, the successor of Y, and the last node in this ordering, respectively. The rules for $all\_reached$ and $-all\_reached$ in P are replaced by the following rules:

$$\begin{aligned} & all\_reached \leftarrow last(X), all\_reached\_upto(X). \\ & all\_reached\_upto(X) \leftarrow all\_reached\_upto(Y), succ(Y,X), reached(X). \\ & all\_reached\_upto(X) \leftarrow \mathit{first}(X), reached(X). \end{aligned}$$

if we add then the respective facts for the ordering, the resulting program (see Figure 8) works as desired. Informally, these rules access the (positive) reachability that is computed by the rules for $reached$ with respect to varying subgraphs G' of G.

The use of an ordering and a rule scheme as above can be easily applied in other contexts. In case the set over which the iteration is made is susceptible to change itself (e.g., if in the previous example only all nodes that have no outgoing edges need to be reached), then special rules can be added that skip elements, indicated by $out(X)$:

$$\begin{array}{l} all\_reached\_upto(X) \leftarrow all\_reached\_upto(Y), succ(Y, X), out(X). \\ all\_reached\_upto(X) \leftarrow first(X), out(X). \end{array}$$

where out is computed using positive rules; the formulation for the continued example is left as a (simple) exercise.

7 Answer Set Solvers

In this section, we mention some AS solvers and briefly present the `DLV` system.

Given that deciding whether a given extended logic program has some answer set is NP-complete, it is clear that efficient computation of answer sets is not easy, and that we can not expect to have a polynomial time algorithm for this task (even under polynomial total-time, i.e., if the combined size of the input P and the output in terms of all answer sets of P is measured). In fact, the problem is yet harder if disjunction in rule heads is allowed.

A number of different, sophisticated algorithms have been developed over the past 15 years (similar as in the area of SAT solving), and to date a number of AS solvers are available; a partial list is shown in Table 1. Some of these solvers provide a number of extensions to the language described here, and have been further developed into families of solvers (e.g. the `DLV` system).

A collection of benchmark problems for AS solvers is maintained at the ASPARAGUS platform,[17] where also information about language formats and the ASP System Competition (whose first edition was at the LPNMR 2007 conference) can be found. In the next subsection, we briefly address implementation strategies of AS solvers; an excellent source on this topic is Ilkka Niemelä's ICLP'04 tutorial.[18]

7.1 Architecture of ASP Solvers

Traditional answer set solvers typically have a two level architecture:

1. **Grounding Step:** Given a program P with variables, a (subset) P' of its grounding is generated which has the same answer sets as P.
2. **Model Search**: The answer sets of the grounded (propositional) program P' are computed.

[17] `http://asparagus.cs.uni-potsdam.de/`

[18] `http://www.tcs.hut.fi/~ini/papers/niemela-iclp04-tutorial.ps.gz.`

Table 1. Some Answer Set Solvers

DLV	`http://www.dbai.tuwien.ac.at/proj/dlv/` [a]
Smodels	`http://www.tcs.hut.fi/Software/smodels/` [b]
GnT	`http://www.tcs.hut.fi/Software/gnt/`
Cmodels	`http://www.cs.utexas.edu/users/tag/cmodels/`
ASSAT	`http://assat.cs.ust.hk/`
NoMore(++)	`http://www.cs.uni-potsdam.de/~linke/nomore/`
Platypus	`http://www.cs.uni-potsdam.de/platypus/`
clasp	`http://www.cs.uni-potsdam.de/clasp/`
XASP	`http://xsb.sourceforge.net`, distributed with XSB v2.6
aspps	`http://www.cs.engr.uky.edu/ai/aspps/`
ccalc	`http://www.cs.utexas.edu/users/tag/cc/`

a + several extensions, e.g. dlvhex, dlv-db, dlt, OntoDLV
b + Smodels_*cc*

Thus, in analogy with the definition of the semantics, also the computation proceeds by a reduction to the propositional case. To facilitate finite computations (and answer sets), many systems do not support function symbols (that is, they handle the so called *Datalog* fragment of logic programming), or only in a very limited form; this is because as already mentioned, function symbols are a well-known source of undecidability, even in rather plain settings (see [12]); see Section 9 for further discussion.

The efficient realization of both steps requires the use of sophisticated algorithms and methods; some have been developed from scratch, while others have been borrowed from related areas, e.g., from SAT Solving. We next look at the two steps.

Grounding Step. Efficient grounding is at the heart of current state-of-the art systems, and different grounding procedures have been realized, including

– DLV's grounder (integrated),
– lparse (for Smodels), gringo (for clasp), which can be used separately, and
– XASP, aspps.

In order to make the ground program P' small and easy to evaluate, sophisticated techniques for "intelligent grounding" have been developed, and restrictions are imposed on the rule syntax, like

– *rule safety* (DLV): every variable in a rule must occur in some positive body literal (i.e., not prefixed with not) whose predicate is not '$=$" or any another built-in comparison predicate. This is a standard condition in the area of deductive databases.
– *domain-restriction* (Smodels): every variable in a rule must occur in a positive *domain predicate*, which are predicates not defined via negative recursion or using "choice rules" [54].

A problem with even highly efficient grounding procedures is that in the end a *grounding bottleneck* may show up: even if a given program P can be evaluated in polynomial space in principle, the (small) grounding P' produced might contain exponentially many rules; this is discussed in detail in [36]. Efficient nonground evaluation of

a:− *not b*.
b:− *not a*.
c:− *not c*, *a*.

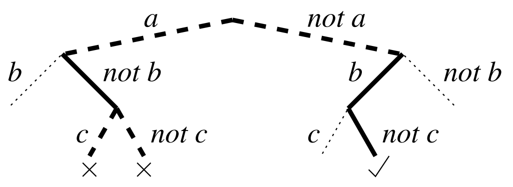


Fig. 9. Model search for a simple program (solid lines = deterministic propagation)

ASP programs intensified only more recently. Techniques for partial and lazy grounding (as used e.g. in [35, 55–57]) are proving to be helpful and thus naturally constitute an important issue for the next generation of AS solvers.

Model search. The second step is model (answer set) search for a propositional program. This is more complicated than the analog problem in SAT Solving or CSP, as it informally comprises two subtasks:

– generation of a candidate model (e.g., a classical model), and
– model checking (testing the stability condition); this problem is easily shown to be P-complete for normal programs and to be co-NP-complete for disjunctive programs, respectively.

The two tasks can be solved using different approaches:

1. One approach, which is historically the first, employs special model search algorithms. Such algorithms have been developed, e.g., for Smodels, `DLV`, NoMore, aspps, and clasp. They take inspiration from the DPLL algorithm for SAT and its variants and improvements, in which truth values are assigned to atoms, consequences that emerge propagated and, if conflicts are found, backtracking takes place. However, while a SAT solver may find any classical model, an AS solver has to find a *specific* such model which satisfies stability; this makes the task much harder. E.g., only atoms can be true that are supported by rules. The result of the model search is an answer set candidate, whose stability may still need to be checked, as it is the case for disjunctive programs in `DLV`, for instance. To this end, the characterization of stable models in terms of unfounded sets according to Theorem 11 is exploited (which can be compiled into an instance of UNSAT).
The search for a specific model led in the `DLV` system, e.g. to four truth values for an atom in the search: *t(rue), f(alse), u(ndefined)*, and *m(ust-be-true)*. Here *must-be-true* means that the atoms has to be true, but its truth still remains to be supported. Starting with all atoms undefined, *possibly true* literals are identified, whose truth value is subsequently determined with trial and error. For a simple example, consider the program in Figure 9. Initially, all atoms are undefined and not a and not b would be possibly true literals. Assume that first a is assigned false; then the right branch is explored. For b, we then can conclude from the first rule must-be-true (as its body must be false) and true from the second rule (as its body is false); hence, b is assigned true. For c, we can conclude false (as a is false). Now

all atoms are either true or false; the candidate $M = \{b\}$ is indeed an answer set and output. Coming back to the root, a is alternatively assigned must-be-true. From the first rule, we would then conclude that b is false, which in turn makes a true; further, one would conclude that $\text{not } c$ is possibly true. However, setting c to false leads to a conflict (by the third rule c then would have to be true), and also setting c to must-be-true, as then c has no support and must be false.
Important for these approaches are heuristics (which atom/rule to consider next); for more details concerning `DLV`, see [58, 59].

2. Later, another approach was to translate the logic program to SAT Solving, which has been realized e.g. in ASSAT and Cmodels. To this end, as already mentioned in Section 5.3 the so called *Clark completion* of a logic program [46] (which translates an acyclic program into an equivalent SAT instance) is extended with loop formulas [43, 44].
   Note that for SAT, model checking is easy in terms of complexity, and can be done in LOGSPACE (in fact, the problem is solvable in ALOGTIME, which is far down in LOGSPACE).[19] An attractive advantage of this approach is that it can benefit from improvements to SAT solving technology; drawbacks are that to generate all answer sets, one needs a SAT solver that can compute all models of a clause set (or one has to tune the transformation for incremental enumeration of answer sets) and that in general, the SAT instance that is constructed can have exponential size.

7.2 The **DLV** System

As an example of an AS solver, we briefly consider here the `DLV` system. [20] `DLV` is a state-of-the-art answer set solver which has been developed at the Vienna University of Technology and the University of Calabria over more than a decade, starting out with a research project on non-monotonic deductive databases in 1996; it is freely available for download.

The system has a language that is richer than the extended disjunctive logic programs considered above, and supports additional constructs (e.g., aggregates, weak constraints) some of which increase the expressivity (e.g., weak constraints allow to express optimization problems with complexity beyond Σ_2^p). `DLV` supports certain built-in predicates (e.g. bounded integer arithmetic and comparisons), and offers a range of front-ends for specific KR tasks (e.g., planning or diagnosis), as well an interface to databases. The principle reasoning tasks supported by `DLV` are 1. answer set generation (all or a given number) and 2. brave and cautious query answering, which is supported for both ground and non-ground queries.

The `DLV` system has been described in many publications, of which [53] is the most comprehensive; the article [60] is recent. As mentioned above, its reasoning engine implements the two-level architecture outlined in Section 7, using a highly optimized grounding module for the first level; the model search is by a DPLL style algorithm

[19] This also intuitively is a clue why there are so many loop formulas, made more precise in [47]: if there were few and they could be easily constructed, we could solve a P-complete (resp. co-NP-complete) problem in LOGSPACE (resp., in polynomial time).

[20] `http://www.dbai.tuwien.ac.at/proj/dlv/`

that uses the characterization of stable models by unfounded sets in Theorem 11. DLV also incorporates a lot of deductive database technology in order to handle larger data volumes (including magic sets, which have been generalized to programs with negation and disjunction). The DLV engine has been extended in many directions leading to a family of systems that support different purposes, including dlv-ex, dlvhex, OntoDLV, dlv-db, and dlt.

DLV syntax. Briefly, the core language of DLV consists of rules of the form

$$\mathtt{a_1\ v\ \cdots\ v\ a_n\ :-\ b_1, \cdots, b_k,\ not\ b_{k+1}, \cdots,\ not\ b_m.}$$

where $n + m > 1$, and all $\mathtt{a_i}$, $\mathtt{b_j}$ are atoms or strongly negated atoms (e.g. $-\mathtt{a}$); no function symbols are allowed; the syntax of terms is like in Prolog. Certain built-in predicates are supported (cf. Table 2). Note that DLV allows constraints $(n = 0)$; as mentioned in Section 7.1, DLV requires rule safety. The extended language also allows that the $\mathtt{b_j}$ are aggregate atoms, in which the values of aggregate functions over a conjunction of literals (including $\#\max$, $\#\min$, #sum, #count, and #times), can be compared to given bounds; we refer to [61] for details.

Furthermore, the DLV language has *weak constraints*, which are of the form

$$\mathtt{:\sim\ b_1, \cdots, b_k,\ not\ b_{k+1}, \cdots,\ not\ b_m.}\ [w : l] \tag{28}$$

where all $\mathtt{b_j}$ are as in rules and w (the *weight*) and l (the *level*, or *layer*) are positive integer constants or variables. For convenience, w and/or l can be omitted and are set to 1 in this case. Informally, the expression (28) is a constraint that can be violated, which incurs cost w; for an answer set, the costs of all violated (instances of) weak constraints are added up, grouped by levels of priorities l. Among all answer sets, those whose cost vector is lexicographically (ordered by priority) smallest are chosen as *optimal answer sets*; see [53] for formal details. With the help of weak constraints, the Guess and Check methodology in Section 6.2 can be extended to a Guess, Check and Optimize Methodology (an example follows shortly).

Queries are specified in DLV by expressions

$$\mathtt{b_1, \cdots, b_k,\ not\ b_{k+1}, \cdots,\ not\ b_m?}$$

where all $\mathtt{b_j}$ are atoms or strongly negated atoms; the query mode (brave or cautious) is selected using a switch (–brave resp. –cautious). Variables in queries are allowed; all bindings of variables to constants will be shown such that the resulting ground query evaluates to true; if the query is ground, in case of brave reasoning a witnessing answer set is shown.[21]

Example 42. To illustrate the use of weak constraints, we consider the well-known Traveling Salesperson problem (TSP), where cities are stored as facts $city(X)$ and direct connections as facts $conn(X, Y, C)$, where C is the cost of traveling from X to Y. Furthermore, the tour is required to start in a city designated with a fact $start(X)$.

The following DLV program computes optimal tours in its optimal answer sets:

[21] Unless magic sets are enabled, which will be the default in future DLV releases.

Table 2. DLV built-ins (Oct-11-2007 release)

Comparison Predicates (for constants and integers):

$<, >, <=, >=, ==, !=$

Arithmetic Predicates (require an upper bound #maxint for integers; see below):

$\#\mathtt{int}(\mathtt{X})$:	X is known integer ($1 \leq \mathtt{X} \leq \mathtt{N}$).
$\#\mathtt{succ}(\mathtt{X},\mathtt{Y})$:	Y is successor of X, i.e., $\mathtt{Y} = \mathtt{X} + 1$.
$+(\mathtt{X},\mathtt{Y},\mathtt{Z})$:	$\mathtt{Z} = \mathtt{X} + \mathtt{Y}$.
$*(\mathtt{X},\mathtt{Y},\mathtt{Z})$:	$\mathtt{Z} = \mathtt{X} * \mathtt{Y}$.

Facts over a fixed integer range

$\mathtt{pred}(c_1..c_2)$.	where $c_1, c_2 \geq 0$ are numeric integer constants, is short for $\mathtt{pred}(c_1)$. $\mathtt{pred}(c_1{+}1)$. $\cdots\mathtt{pred}(c_2 - 1)$. $\mathtt{pred}(c_2)$.

Built-in constants

#maxint	upper integer limit, set with -N switch, or with #maxint $= i$. for integer $i \geq 0$ in the program

---

$$\left.\begin{array}{l}\mathtt{inTour(X,Y,C)\ v\ outTour(X,Y,C)\ {:}-\ start(X), conn(X,Y,C).}\\ \mathtt{inTour(X,Y,C)\ v\ outTour(X,Y,C)\ {:}-\ reached(X), conn(X,Y,C).}\\ \mathtt{reached(X)\ {:}-\ inTour(Y,X,C).}\qquad\qquad \textbf{(aux.)}\end{array}\right\}\ \textbf{Guess}$$

$$\left.\begin{array}{l}\mathtt{{:}-\ inTour(X,Y,\_), inTour(X,Y1,\_), Y <> Y1.}\\ \mathtt{{:}-\ inTour(X,Y,\_), inTour(X1,Y,\_), X <> X1.}\\ \mathtt{{:}-\ city(X), not\ reached(X).}\end{array}\right\}\ \textbf{Check}$$

$$\left.\mathtt{{:}\sim\ inTour(X,Y,C).\ [C:1]}\right\}\ \textbf{Optimize}$$

Here, the first three rules guess a tour, which is done in an incremental manner beginning at the start city for all cities reached. Note that different from Example 37, we want a complete tour (a cycle) rather than a path and thus the start city must be reached from some other city. The weak constraint in the optimize part states that including the connection from X to Y costs C; the total cost of an answer set is the cost of a tour (all weak constraints have the same priority).

If we add the query

```
start(X), inTour(X, Y, C)?
```

to the program, then we obtain under brave reasoning all possible first legs for an optimal tour, and under cautious reasoning a mandatory first leg (if one exists).

Front-ends. Besides the answer set semantics core, DLV offers various front-ends for particular KR tasks, including

– diagnosis

– knowledge-based planning ($\mathcal{K}$ language)
– front-end to SQL3
– inheritance reasoning

The first three front-ends can be invoked using command-line switches, while the inheritance front-end is automatically enabled if the input is an inheritance program [62]. Many other front-ends, created by various authors, are available as separate packages.

Using `DLV`. The `DLV` system is primarily command-line oriented, but there is also a plain GUI and there are web interfaces available.[22]

The system reads input from files whose names are passed on the command-line. If the command-line option "`--`" has been specified, input is also read from standard input (stdin). Output is printed to standard output (stdout), one line per answer set. Detailed documentation and an online manual are available at the `DLV` homepage. [20]

8 ASP for the Semantic Web

As mentioned in Section 6, ASP has been deployed to many application areas. We focus here on the Semantic Web, where different ways to exploit ASP and ASP techniques have been considered (see [17, 18, 63] for more discussion):

– As a host language for Web/Semantic Web formalisms. For example, mappings respectively encodings of ontologies in description logics into ASP have been conceived (see [64] for references), and encoding of web query languages, e.g. SPARQL [65].
– For diverse problem solving, like Web service composition (e.g. [66, 67]), Web Service repair [68], or ontology merging [69, 70].
– For combining rules and ontologies into a unifying framework (cf. [64, 71, 73] for discussion and references).

In the context of the Semantic Web, special needs arise that have to be accommodated:

– dealing with open worlds and domains (cf. [74, 75]),
– access to (semi-)structured and poorly structured data,
– external sources and distributed computation (cf. [76]),
– heterogeneity of sources, and
– web dynamics (cf. [77]).

In the rest of this section, we review some research and development efforts which have been moving ASP languages in the direction of the Semantic Web. Among them are extensions of ASP to access ontologies in OWL, the Web Ontology Language, and extensions which allow to access heterogeneous knowledge sources on the Web. For more information on ASP for the Semantic Web, we refer to previous Reasoning Web schools [18, 17] and the tutorial [63].

[22] E.g. `http://asptut.gibbi.com/`

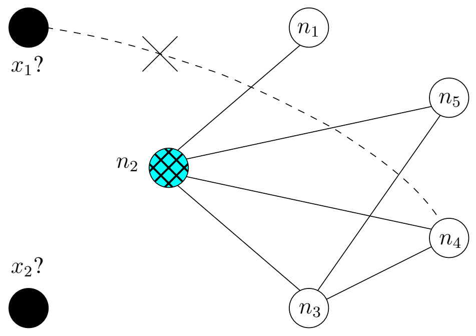


Fig. 10. Hightraffic network

8.1 DL-Programs

Description logic programs (dl-programs), which had been introduced in [78], are a form of hybrid knowledge bases combining description logics[23] and logic programs under answer set semantics. They form another contribution to the attempt in finding an appropriate formalisms for combined rules and ontologies for the Semantic Web.

Roughly speaking, dl-programs consist of a normal logic program P and a description logic knowledge base (DL-KB) L. In addition to traditional atoms, the logic program P might contain special devices, called dl-atoms. Those dl-atoms may occur in the body of a rule and involve queries to L. Moreover, dl-atoms can specify an input to L before querying it, thus in dl-programs a bidirectional data flow is possible between the description logic component and the logic program.

The way dl-programs interface DL-KBs enables the possibility of acting as a loosely coupled formalism between a knowledge base formulated in terms of a logic program and a knowledge base formulated in terms of description logic axioms. This feature brings the advantage of reusing existing logic programming and DL systems in order to build an implementation of dl-programs.

In the following, we provide the syntax of dl-programs and an overview of the semantics. An in-detail treatise is given in [64].

We will illustrate the main ideas behind the notion of dl-program with the following example:

Example 43. An existing network must be extended by new nodes (Fig. 10). The knowledge base L_N contains information about existing nodes $(n_1, \ldots, n_5)$ and their interconnections as well as a definition of "overloaded" nodes (concept $\mathit{HighTrafficNode}$),

[23] The reader is referred to [79] of this volume for a general background on description logics.

which are nodes with more than three connections:

$$\begin{aligned}
&\geq 1\ \mathit{wired} \sqsubseteq \mathit{Node};\ \ \top \sqsubseteq \forall \mathit{wired}.\mathit{Node};\ \ \mathit{wired} = \mathit{wired}^{-};\\
&\geq 4\ \mathit{wired} \sqsubseteq \mathit{HighTrafficNode};\ \ n_1 \neq n_2 \neq n_3 \neq n_4 \neq n_5;\\
&\mathit{Node}(n_1);\ \ \mathit{Node}(n_2);\ \ \mathit{Node}(n_3);\ \ \mathit{Node}(n_4);\ \ \mathit{Node}(n_5);\\
&\qquad\qquad \mathit{wired}(n_1, n_2);\ \ \mathit{wired}(n_2, n_3);\ \ \mathit{wired}(n_2, n_4);\\
&\qquad\qquad \mathit{wired}(n_2, n_5);\ \ \mathit{wired}(n_3, n_4);\ \ \mathit{wired}(n_3, n_5).
\end{aligned}$$

In L_N, only n_2 is an overloaded node, and is highlighted in Fig. 10 with a criss-cross pattern.

To evaluate possible combinations of connecting the new nodes, the following program P_N is specified:

$$\mathit{newnode}(x_1). \tag{29}$$

$$\mathit{newnode}(x_2). \tag{30}$$

$$\mathit{overloaded}(X) \leftarrow \mathrm{DL}[\mathit{wired} \uplus \mathit{connect}; \mathit{HighTrafficNode}](X). \tag{31}$$

$$\begin{aligned}\mathit{connect}(X, Y) \leftarrow\ &\mathit{newnode}(X), \mathrm{DL}[\mathit{Node}](Y),\\ &\text{not } \mathit{overloaded}(Y), \text{not } \mathit{excl}(X, Y).\end{aligned} \tag{32}$$

$$\mathit{excl}(X, Y) \leftarrow \mathit{connect}(X, Z), \mathrm{DL}[\mathit{Node}](Y), Y \neq Z. \tag{33}$$

$$\mathit{excl}(X, Y) \leftarrow \mathit{connect}(Z, Y), \mathit{newnode}(Z), \mathit{newnode}(X), Z \neq X. \tag{34}$$

$$\mathit{excl}(x_1, n_4). \tag{35}$$

Rules (29)–(30) define the new nodes to be added. Rule (31) imports knowledge about overloaded nodes in the existing network, taking new connections already into account. Rule (32) connects a new node to an existing one, provided the latter is not overloaded and the connection is not to be disallowed, which is specified by Rule (33) (there must not be more than one connection for each new node) and Rule (34) (two new nodes cannot be connected to the same existing one). Rule (35) states a specific condition: Node x_1 must not be connected with n_4.

Two different semantics have been defined for dl-programs, the (strong) answer-set semantics [78] and the well-founded semantics [80, 81]. The former extends the notion of Gelfond-Lifschitz reduct (see Section 4) incorporating the presence of dl-atoms: dl-programs can have, in general, multiple answer sets. The latter extends the well-founded semantics of [22] to dl-programs.

Example 44. As specified by the strong answer set semantics of dl-programs, the program (L_N, P_N) in Example 43 has four strong answer sets (we show only atoms with predicate *connect*): $M_1 = \{\mathit{connect}(x_1, n_1), \mathit{connect}(x_2, n_4), \ldots\}$, $M_2 = \{\mathit{connect}(x_1, n_1), \mathit{connect}(x_2, n_5), \ldots\}$, $M_3 = \{\mathit{connect}(x_1, n_5), \mathit{connect}(x_2, n_1), \ldots\}$, and $M_4 = \{\mathit{connect}(x_1, n_5), \mathit{connect}(x_2, n_4), \ldots\}$. Note that the ground dl-atom

$$\mathrm{DL}[\mathit{wired} \uplus \mathit{connect}; \mathit{HighTrafficNode}](n_2)$$

from rule (31) is true in any partial interpretation of P_N. According to the proposed well-founded semantics for dl-programs in [80], the unique well-founded model of (L_N, P_N) contains thus $\mathit{overloaded}(n_2)$.

Features and Properties of DL-Programs. The strong answer set semantics of dl-programs is nonmonotonic, and generalizes the stable semantics of ordinary logic programs. In particular, satisfiable positive dl-programs (programs without default negation and $\cap$ operator) have a least model semantics, and satisfiable stratified dl-programs have a unique minimal model which is iteratively described by a finite sequence of least models.

Applications. The bidirectional flow of knowledge between a description logic base and a logic program component enables a variety of possibilities. A major application for dl-programs is nonmonotonic reasoning on top of monotonic systems. It is for instance possible to take a dl-knowledge base L and coupling it with a properly designed logic program in order to extend L with *defaults* [38] and *closed world assumption* (CWA) [37]. Both reasoning applications can be implemented in dl-programs to support nonmonotonic reasoning for description logics, as detailed in [64].

8.2 HEX-Programs

HEX-programs [82] are declarative nonmonotonic logic programs with support for external knowledge and higher-order disjunctive rules, under answer set semantics. In spirit of dl-programs, they allow for a loose coupling between general external knowledge sources and declarative logic programs through the notion of external atoms, which take input from the logic program and exchange inferences with the external source. In addition, meta-reasoning tasks may be accomplished by means of higher-order atoms. HEX-programs are evaluated under a generalized answer-set semantics, thus are in principle capable of capturing many proposed extensions in answer-set programming.

Syntax of HEX-Programs. Let $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, and *external predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ are prefixed with the "&" symbol. We note that constant names serve both as individual and predicate names.

Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. A *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \ldots, Y_n)$, where $Y_0, \ldots, Y_n$ are terms; $n \geq 0$ is the *arity* of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \ldots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant.

For example, $(x, \mathit{rdf}{:}\mathit{type}, c)$, $\mathit{node}(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms.

An *external atom* is of the form

$$\&g[Y_1, \ldots, Y_n](X_1, \ldots, X_m) \ , \tag{36}$$

where $Y_1, \ldots, Y_n$ and $X_1, \ldots, X_m$ are two lists of terms (called *input* and *output* lists, respectively), and $\&g \in \mathcal{G}$ is an external predicate name. We assume that $\&g$ has fixed lengths $\mathit{in}(\&g) = n$ and $\mathit{out}(\&g) = m$ for input and output lists, respectively. Intuitively, an external atom provides a way for deciding the truth value of an output tuple depending on the extension of a set of input predicates: in this respect, an external predicate $\&g$ is equipped with a function $f_{\&g}$ evaluating to true for proper input values.

$$\mathit{subRelation}(\mathit{brotherOf}, \mathit{relativeOf}). \tag{38}$$
$$\mathit{brotherOf}(\mathit{john}, \mathit{al}). \tag{39}$$
$$\mathit{relativeOf}(\mathit{john}, \mathit{joe}). \tag{40}$$
$$\mathit{brotherOf}(\mathit{al}, \mathit{mick}). \tag{41}$$
$$\mathit{invites}(\mathit{john}, X) \vee \mathit{skip}(X) \leftarrow X \neq \mathit{john}, \&\mathit{reach}[\mathit{relativeOf}, \mathit{john}](X). \tag{42}$$
$$R(X,Y) \leftarrow \mathit{subRelation}(P,R), P(X,Y). \tag{43}$$
$$\mathit{someInvited} \leftarrow \mathit{invites}(\mathit{john}, X). \tag{44}$$
$$\leftarrow \text{not } \mathit{someInvited}. \tag{45}$$
$$\leftarrow \&\mathit{degs}[\mathit{invites}](\mathit{Min}, \mathit{Max}), \mathit{Max} > 2. \tag{46}$$

Fig. 11. Example HEX program

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_m, \text{not } \beta_{m+1}, \ldots, \text{not } \beta_n \ , \tag{37}$$

where $m, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are atoms, and $\beta_1, \ldots, \beta_n$ are either atoms or external atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_m\}$ and $B^-(r) = \{\beta_{m+1}, \ldots, \beta_n\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms. A HEX-*program* is a finite set P of rules. It is *ordinary*, if all rules are ordinary.

We next give an illustrative example.

Example 45 ([83]). Consider the HEX-program P in Figure 11. Informally, this program randomly selects a certain number of John's relatives for invitation. The first line states that $\mathit{brotherOf}$ is a subrelation of $\mathit{relativeOf}$, and the next three lines give concrete facts. The disjunctive rule (42) chooses relatives, employing the external predicate $\&\mathit{reach}$. This latter predicate takes in input a binary relation e and a node name n, returning the nodes reachable from n when traversing the graph described by e (see the following Example 47). Rule (43) axiomatizes subrelation inclusion exploiting higher-order atoms; that is, for those couples of binary predicates p, r for which it holds $\mathit{subRelation}(p, r)$, it must be that $r(x, y)$ holds whenever $p(x, y)$ is true.

The constraints (45) and (46) ensure that the number of invitees is between 1 and 2, using (for illustration) an external predicate $\&\mathit{degs}$ from a graph library. Such a predicate has a valuation function $f_{\&\mathit{degs}}$ where $f_{\&\mathit{degs}}(I, e, \mathit{min}, \mathit{max})$ is true iff min and max are, respectively, the minimum and maximum vertex degree of the graph induced by the edges contained in the extension of predicate e in interpretation I.

Semantics of HEX-Programs. In the sequel, let P be a HEX-program. The *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{C}$. The grounding of a rule r, $\mathit{grnd}(r)$, is defined accordingly, and the grounding of program

P is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$, and $\mathcal{G}$ are implicitly given by P.

Example 46 ([83]). Given $\mathcal{C} = \{edge, arc, a, b\}$, ground instances of $E(X, b)$ are for instance $edge(a, b)$, $arc(a, b)$, $a(edge, b)$, and $arc(arc, b)$. Ground instances of $\&reach[edge, N](X)$ are all possible combinations where N and X are replaced by elements from $\mathcal{C}$; some examples are $\&reach[edge, edge](a)$, $\&reach[edge, arc](b)$, and $\&reach[edge, edge](edge)$.

An *interpretation relative to P* is any subset $I \subseteq HB_P$ containing only atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$.

With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n+m+1)$-ary Boolean function $f_{\&g}$ assigning each tuple $(I, y_1 \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, if and only if $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$.

Example 47 ([83]). Let us associate with the external atom $\&reach$ a function $f_{\&reach}$ such that $f_{\&reach}(I, E, A, B) = 1$ iff B is reachable in the graph E from A. Let $I = \{e(b, c), e(c, d)\}$. Then, I is a model of $\&reach[e, b](d)$ since $f_{\&reach}(I, e, b, d) = 1$.

Note that in contrast to the semantics of higher-order atoms, which in essence reduces to first-order logic as customary (cf. [84]), the semantics of external atoms is in spirit of second order logic since it involves predicate extensions.

Considering example 45, as John's relatives are determined to be Al, Joe, and Mick, P has six answer sets, each of which contains one or two of the facts $invites(john, al)$, $invites(john, joe)$, and $invites(john, mick)$.

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ iff $I \models H(r)$ whenever $I \models B(r)$. We say that I is a *model* of a HEX-program P, denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model.

Given a HEX-program P, the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I, is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an *answer set of P* iff I is a minimal model of fP^I.

In principle, the truth value of an external atom depends on its input and output lists and on the entire model of the program. In practice, however, we can identify certain types of input terms that allow to restrict the input interpretation to specific relations. The Boolean function associated with the external atom $\&reach[edge, a](X)$ for instance will only consider the extension of the predicate $edge$ and the constant value a for computing its result, and simply ignore everything else of the given input interpretation.

Features and Properties of HEX-Programs. As mentioned above, HEX-programs are a generalization of dl-programs, consisting indeed in a form of coupling of rules with arbitrary external computation sources, within a declarative logic-based setting. The higher-order features are similar to those of HiLog [85], i.e., the semantics of this high-order extension is still within first-order logic.

The semantics of HEX-programs conservatively extends ordinary answer-set programs, and it is easily extendable to support weak constraints [86]. External predicates can define other ASP features like aggregate functions [61]. Computational complexity of the language depends on external functions. The former is however not affected if external functions evaluate in polynomial time.

The dlvhex prototype,[24] an implementation of HEX-programs, is based on a flexible and modular architecture. The evaluation of the external atoms is realized by plugins, which are loaded at run-time. The pool of available external predicates can be easily customized by third-party developers.

Applications. HEX-programs have been put to use in many applications in different contexts. Hoehndorf et al. [70] showed how to combine multiple biomedical upper ontologies by extending the first-order semantics of terminological knowledge with default logic. The corresponding prototype implementation of such kind of system is given by mapping the default rules to HEX-program. Fuzzy extensions of answer-set programs and their relationship to HEX-programs are given in [87, 88]. The former maps fuzzy answer set programs to HEX-programs, whereas the latter defines a fuzzy semantics for HEX-programs and gives a translation to standard HEX-programs. In [89], the planning language $\mathcal{K}^c$ has been introduced which features external function calls in spirit of HEX-programs.

8.3 Other linguistic extension of ASP in the direction of Semantic Web

We briefly survey here other notable works aiming at integrating the stable model semantics with Semantic Web related formalisms, and remind the reader to other discussions of related work such as [18, 71].

Research efforts can be categorized in the two main groups of *translational approaches* and *integration approaches*. The latter can be further classified in *loose, tight* or *full* integration.

As for integration approaches, $\mathcal{DL}+log$ [90] is the latest in a chain of extensions of the DL $\mathcal{ALC}$ with rules such as $\mathcal{AL}$-log, r- and r^+-hybrid knowledge bases. As a tight semantics approach, $\mathcal{DL}+log$ gives meaning to combined knowledge bases in terms of unique model structures, which aim at satisfying both the description logic base at hand and the logic program. *Hybrid MKNF knowledge bases* [91] build on Lifschitz's bimodal *Logic of Minimal Knowledge and Negation as Failure (MKNF)* [93], and aim at a seamless (which is sometimes referred as *full* integration) integration of classic and nonmonotonic semantics beyond tight integration approaches. Besides dl-programs and hex-programs, it is worth mentioning other loose coupling languages in the direction of probabilistic [94] and fuzzy hybrid systems [95] under stable semantics; see [73] for an overview. An extension of RDF(S) with stable models has been proposed in [96].

One might also consider the idea of translating Semantic Web ontologies to semantically equivalent ASP logic programs. This task is quite challenging, given the profound semantic differences between the two formalisms. Nonetheless, some success has been reached, and translation from several flavors of description logics to ASP are known, cf.

[24] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/`

[6, 97, 98, 32]. Notably, the availability of function symbols (or, in any case, of infinite domains), solves some of the semantic difficulties [32, 99].

8.4 Other Semantic Web enabled systems based on ASP

OntoDLV [100] is a system for ontologies specification and reasoning under answer set semantics. OntoDLV implements a logic-based ontology representation language, called OntoDLP (where DLP stands for Disjunctive Logic Programs), which is an extension of (disjunctive) ASP with all the main ontology constructs including classes, inheritance, relations, and axioms. OntoDLP is strongly typed, and includes also complex type constructors, like lists and sets. OntoDLV supports some interoperability mechanism with OWL, allowing the user to retrieve information from external OWL Ontologies and to exploit this data in OntoDLP ontologies and queries. OntoDLV facilitates the development of complex applications in a user-friendly visual environment; it is endowed with a robust persistency-layer for saving information transparently on a DBMS, and it seamlessly integrates the `DLV` system [53] exploiting the power of a stable and efficient AS solver. Indeed, OntoDLV is already used for the development of real-world applications including agent-based systems, information extraction and text classification frameworks.

GiaBATA [101] is a system for storing, aggregating, and querying Semantic Web data, based on declarative logic programming technology, namely on the dlvhex system, which allows to implement a fully SPARQL compliant semantics, and on `dlv-db`, which extends the `DLV` system with persistent storage capabilities. Compared with off-the-shelf RDF stores and SPARQL engines[25], GiaBATA offers more flexible support for rule-based RDFS and other higher entailment regimes by enabling custom reasoning via rules, and the possibility to choose the reference ontology on a per query basis. Due to the declarative approach, GiaBATA gains the possibility of applying well-known logic-level optimization features of logic programming (LP) and deductive database systems. The architecture of GiaBATA allows for extensions of SPARQL by non-standard features such as aggregates, custom built-ins, or arbitrary rulesets. The resulting system provides a flexible toolbox that embeds Semantic Web data and ontologies in a fully declarative LP environment.

9 Conclusion

Answer Set Programming is a booming paradigm for declarative problem solving, which emerged from Logic Programming and Nonmonotonic Reasoning and has been deployed to a range of application areas. A number of answer set solvers are available which provide a variety of constructs and the features for problem modeling, helping the user to find formalizations of problems in a more natural and understandable manner. As for the Semantic Web, extensions of the basic ASP languages and formalisms have been

[25] For details of RDF and its query language SPARQL, the reader may refer to [102] in this volume.

developed, aiming at different goals. For example, to provide a formalism for combining rules and ontologies (e.g., dl-programs, $\mathcal{DL}+log$[90] and MKNF knowledge bases [91], conceptual logic programs [103], hybrid and guarded hybrid knowledge bases [104, 105], and open answer set programming [99]; see [64, 18] and references therein), or more general formalisms for accessing and interfacing data on the (Semantic) Web like HEX-programs, and systems like dlvhex, OntoDLV, and GiaBATA.

The interest in Answer Set Programming and significance of the underlying stable model semantics could be experienced at last year's edition of the International Logic Programming conference, which dedicated a (well-attended) special session to discuss the influence of stable model semantics on the field of logic programming. It witnessed ASP to be a vibrant area of research in which despite the advances and developments in the last years still a number of research challenges exist.

While the theory of ASP is well-developed and applications are expanding, the deployment of ASP to an industrial scale needs further efforts (cf. Nicola Leone's talk at LPNMR 2007).[26] Next generation answer set solvers must be developed which provide better support for the needs in practice.

Among these needs are complex data structures including lists, sets, records etc.; underneath, this calls for function symbols (recall that in Prolog, lists are special syntax function symbols) and a move beyond the Datalog fragment of logic programming. As mentioned in Section 7, function symbols have been largely banned because the quickly lead to undecidability. Only more recently, work on decidable classes of and prototype implementations of stable models semantics with function symbols has been carried out, including [34, 106–108, 35, 32, 33], and function symbols also increasingly attract attention as a modeling construct.

The class of ω-restricted programs [34] has been implemented on top of Smodels, and the recently presented class of finitely-ground programs in the DLV-Complex system on top of DLV [109, 35], which aims at providing functions symbols in a decidable setting, giving support to lists and sets along with libraries for their manipulations. However, both system, models are always finite, which hinders modeling infinite processes and objects; classes like finitary programs [106], finitely recursive programs [107], FDNC-programs [32], and BD-programs [33] do not have this restriction, but lack implementations to date.

Related to this issue is incremental model building, which is of particular interest for applications in reasoning about actions and change: a model may describe the evolution of the world, which happens from one epoch to the next. Here, it is desired to build the model according to the evolution, step by step; this is, for instance, relevant for planning. Recent work [55] aims in this direction, giving a formal framework for incremental model building.

Another issue of relevance for practice is modularity in ASP, and to provide means for code reuse. While modularity has been recognized as an important aspect more than a decade ago [110], it has only found more recently increasing attention, cf. [111–115]; the use of macros [116] and templates [117] aims in this direction. Specifically for the Semantic Web context, a modular formalism with multiple nonmonotonic logic programs [74] and the MWeb framework [76] have been conceived. The formalism in [74] allows to

[26] http://lpnmr2007.googlepages.com/nicola-lpnmr07.pdf

interlink web-accessible logic programs, i.e., logic programs may refer to logic programs that may refer to remote knowledge bases distributed on the Web. `MWeb` attempts to enhance the Semantic Web with the notions of scope and context for modular web rule bases, and pays attention to support knowledge hiding and the safe use of strong and weak negation, as well as to different reasoning modes.

However, most of these approaches reduce a system of modules (polynomially) into a single global program, or impose constraints regarding the use of recursion, resulting in limited expressiveness. The recent approach in [111], which improves [110], has no such constraints but the high expressiveness comes with high worst case complexity in general. Efficient algorithms and implementations of yet expressive, natural modular ASP frameworks are an interesting topic of research.

Concerning efficiency, of course also improvements to solvers for ordinary ASP are desirable. In the recent years, there has been a lot of work on optimizations based on program equivalence, triggered by the seminal paper [118], which introduced a notion of strong equivalence between non-monotonic logic programs that takes nonmonotonicity into account and led to a whole family of notions of equivalence, which may be utilized to rewrite a program into an equivalent one that can be see e.g. [27, 119–121] for more on this issue. Another issue is non-ground ASP processing, in order to overcome the intrinsic grounding bottleneck in the two step architecture of most answer set solvers. Work on this is underway, and techniques for partial and lazy grounding (as used e.g. in [35, 55–57]) are helpful; however, the grounding techniques of advanced AS solvers are highly sophisticated and in most case very effective. Interesting in this regard is also the work of [122], which defines answer sets for first-order theories that can be non-Herbrand models, in terms of a formula in second-order logic. While this avoids grounding, it remains to be seen whether this approach can be effectively implemented (by reducing, e.g., fragments of the formalisms to standard theorem provers).

Finally, for deployment on a larger scale, more software engineering tools and methodologies are needed. Compared to other programming languages, there is currently little support for programmers available, and rich ASP programming environments are lacking, which include debuggers, visualization, libraries etc. The *International Workshop on Software Engineering for Answer Set Programming (SEA)* [123] was initiated as a forum for researchers interested in these issues.[27] Given the work in progress, we may expect significant advances and improvements here in the near future.

In conclusion, though ASP has developed vigorously, there is still much ado in an exciting area of research for theory and applications.

Acknowledgments We thank Minh Dao-Tran, Wolfgang Faber and Gerald Pfeifer for their valuable comments and corrections.

References

1. Provetti, A., Son, T.C., eds.: Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop,

[27] `http://sea07.cs.bath.ac.uk/`

Stanford, March 26-28, 2001. In Provetti, A., Son, T.C., eds.: Answer Set Programming. (2001)

2. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the a-prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
3. Lifschitz, V.: Answer Set Programming and Plan Generation. Artificial Intelligence **138** (2002) 39–54
4. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In Apt, K., Marek, V.W., Truszczyński, M., Warren, D.S., eds.: The Logic Programming Paradigm – A 25-Year Perspective. Springer (1999) 375–398
5. Niemelä, I.: Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. Annals of Mathematics and Artificial Intelligence **25**(3–4) (1999) 241–273
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A., Sadri, F., eds.: Computational Logic: From Logic Programming into the Future. Volume 2408 of LNCS/LNAI., Springer (2002) 413–451
8. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium, Cambridge, Mass., MIT Press (1988) 1070–1080
9. Niemelä (ed.), I.: Language Extensions and Software Engineering for ASP. Technical Report WP3, Working Group on Answer Set Programming (WASP), IST-FET-2001-37004 (September 2005) Available at `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/wasp-wp3-web/`.
10. Kowalski, R.: Algorithm = Logic + Control. Commun. ACM **22**(7) (1979) 424–436
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
12. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425
13. Asparagus homepage: `http://asparagus.cs.uni-potsdam.de/` (Since 2005)
14. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczynski, M.: The First Answer Set Programming System Competition. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Volume 4483 of LNCS., Springer (2007) 3–17
15. Bidoit, N.: Negation in rule-based database languages: A survey. Theor. Comput. Sci. **78**(1) (1991) 3–83
16. Apt, K., Bol, N.: Logic programming and negation: A survey. Journal of Logic Programming **19/20** (1994) 9–71
17. Eiter, T., Ianni, G., Polleres, A., Schindlauer, R., Tompits, H.: Reasoning with rules and ontologies. In Barahona, P., Bry, F., Franconi, E., Sattler, U., Henze, N., eds.: Reasoning Web 2006. Volume 4126 of LNCS. Springer (September 2006) 93–127
18. Eiter, T., Ianni, G., Krennwallner, T., Polleres, A.: Rules and Ontologies for the Semantic Web. In Baroglio, C., Bonatti, P.A., Maluszynski, J., Marchiori, M., Polleres, A., Schaffert, S., eds.: Reasoning Web: 4th International Summer School 2008, Venice Italy, September 7-11, 2008, Tutorial Lectures. Volume 5224 of LNCS. Springer (September 2008) 1–53 Slides available at `http://rease.semanticweb.org/`.
19. Dix, J.: A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. Fundam. Inform. **22**(3) (1995) 227–255
20. Dix, J.: A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. Fundam. Inform. **22**(3) (1995) 257–288
21. Przymusinski, T.C.: On the Declarative Semantics of Deductive Databases and Logic Programs. [124] 193–216

22. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM **38**(3) (1991) 620–650
23. Apt, K., Blair, H., Walker, A.: Towards a Theory of Declarative Knowledge. [124] 89–148
24. Minker, J.: Logic and Databases: A 20 Year Retrospective. In: Proceedings of the International Workshop on Logic in Databases (LID'96). Volume 1154 of LNCS., Springer (1996) 3–57
25. Lifschitz, V.: Twelve definitions of a stable model. In: ICLP. (2008) 37–51
26. Ferraris, P., Lifschitz, V.: Mathematical foundations of answer set programming. In: We Will Show Them! Essays in Honour of Dov Gabbay, Volume One, College Publications (2005) 615–664
27. Gelfond, M.: Answer sets. In F. van Harmelen, V. Lifschitz, B.P., ed.: Handbook of Knowledge Representation. Elsevier (2008) 285–316
28. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135**(2) (June 1997) 69–112
29. Lifschitz, V., Turner, H.: Splitting a Logic Program. In Van Hentenryck, P., ed.: Proceedings of the 11th International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy, MIT Press (June 1994) 23–37
30. Marek, V.W., Truszczyński, M.: Autoepistemic Logic. Journal of the ACM **38**(3) (1991) 588–619
31. Papadimitriou, C.H.: Computational Complexity. Addison Wesley Longman (1994)
32. Šimkus, M., Eiter, T.: FDNC: Decidable non-monotonic disjunctive logic programs with function symbols. In Dershowitz, N., Voronkov, A., eds.: Proceedings 14th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2007). Number 4790 in LNCS, Springer (2007) 514–530 Extended Paper to appear in *ACM Trans. Computational Logic*.
33. Eiter, T., Šimkus, M.: Bidirectional answer set programs with function symbols. In Boutilier, C., ed.: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09), AAAI Press/IJCAI (2009)
34. Syrjänen, T.: Omega-restricted logic programs. In Eiter, T., Faber, W., Truszczynski, M., eds.: Proc. 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01). Volume 2173 of Lecture Notes in Computer Science., Springer (2001) 267–279
35. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings. Volume 5366 of LNCS., Springer (2008) 407–424
36. Eiter, T., Faber, W., Fink, M., Woltran, S.: Complexity results for answer set programming with bounded predicate arities and implications. Annals of Mathematics and Artificial Intelligence **51**(2-4) (2007) 123–165
37. Reiter, R.: On Closed-World Databases. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press, New York (1978) 55–76
38. Reiter, R.: A Logic for Default Reasoning. Artificial Intelligence **13**(1–2) (1980) 81–132
39. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Annals of Mathematics and Artificial Intelligence **12** (1994) 53–87
40. Pearce, D.: Equilibrium logic. Annals of Mathematics and Artificial Intelligence **47**(1-2) (2006) 3–41
41. Pearce, D., Valverde, A.: Quantified equilibrium logic and foundations for answer set programs. [125] 546–560
42. Eiter, T., Leone, N., Pearce, D.: Assumption Sets for Extended Logic Programs. In Gerbrandy, J., Marx, M., de Rijke, M., Venema, Y., eds.: JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday. Amsterdam University Press (1999) ISBN 90 5629 104

1. Available at `http://www.kr.tuwien.ac.at/staff/eiter/et-archive/jfak.pdf`.
43. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI/IAAI. (2002) 112–
44. Lee, J., Lifschitz, V.: Loop Formulas for Disjunctive Logic Programs. In: Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03), Springer Verlag (December 2003) 451–465
45. Lee, J.: A model-theoretic counterpart of loop formulas. In Kaelbling, L.P., Saffiotti, A., eds.: IJCAI, Professional Book Center (2005) 503–508
46. Clark, K.L.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press, New York (1978) 293–322
47. Lifschitz, V., Razborov, A.A.: Why are there so many loop formulas? ACM Trans. Comput. Log. **7**(2) (2006) 261–268
48. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. Annals of Mathematics and Artificial Intelligence **15**(3/4) (1995) 289–323
49. Stockmeyer, L.J.: The polynomial-time hierarchy. Theor. Comput. Sci. **3**(1) (1976) 1–22
50. Lifschitz, V.: Answer set planning. In: ICLP. (1999) 23–37
51. Woltran (ed.), S.: Answer Set Programming: Model Applications and Proofs-of-Concept. Technical Report WP5, Working Group on Answer Set Programming (WASP), IST-FET-2001-37004 (July 2005) Available at `http://www.kr.tuwien.ac.at/projects/WASP/report.html`.
52. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative problem-solving using the DLV system. In Minker, J., ed.: Logic-Based Artificial Intelligence, Kluwer Academic Publishers (2000) 79–103
53. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (July 2006) 499–562
54. Syrjänen, T., Niemelä, I.: The smodels system. In: 6th International Conference on Logic Programming and Nonmotonic Reasoning (LPNMR 2001). Volume 2173., Springer (2001) 434–438
55. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an Incremental ASP Solver. In de La Banda, M., Pontelli, E., eds.: Proceedings 24th International Conference on Logic Programming (ICLP 2008). Number 5366 in LNCS, Springer (2008) 190–205
56. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. In: LaSh 2008: LOGIC AND SEARCH - Computation of structures from declarative descriptions. (2008)
57. Lef'evre, C., Nicolas, P.: Integrating grounding in the search process for answer set computing. In: ASPOCP: Answer Set Programming and Other Constraint Paradigms. (2008) 89–103
58. Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, Institut für Informationssysteme, Technische Universität Wien (2002)
59. Maratea, M., Ricca, F., Faber, W., Leone, N.: Look-back techniques and heuristics in dlv: Implementation, evaluation, and comparison to qbf solvers. J. Algorithms **63**(1-3) (2008) 70–89
60. Leone, N., Faber, W.: The DLV project: A tour from theory and research to applications and market. [125] 53–68
61. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the DLV system. Theory and Practice of Logic Programming **8**(5-6) (2008) 545–580

62. Buccafurri, F., Faber, W., Leone, N.: Disjunctive logic programs with inheritance. Theory and Practice of Logic Programming **2**(3) (2002)
63. Eiter, T.: Answer set programming for the Semantic Web (tutorial). In Niemelä, I., Dahl, V., eds.: Proceedings 23rd International Conference on Logic Programming (ICLP 2007). Number 4670 in LNCS, Springer (2007) 23–26 Slides available at `http://www.dcc.fc.up.pt/iclp07/eiter.pdf`.
64. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining Answer Set Programming with Description Logics for the Semantic Web. Artificial Intelligence **172**(12-13) (2008) 1495–1539
65. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the 16th International Conference on World Wide Web (WWW), ACM (2007) 787–796
66. Rainer, A.: Web Service Composition under Answer Set Programming. In: Proc. KI 2005 Workshop "Planen, Scheduling und Konfigurieren, Entwerfen" (PuK 2005). (2005)
67. Pontelli, E., Son, T.C., Baral, C.: A framework for composition and inter-operation of rules in the semantic web. In Eiter, T., Franconi, E., Hodgson, R., Stephens, S., eds.: RuleML, IEEE Computer Society (2006) 39–50
68. Friedrich et al., G.: Model-based repair of web service processes. Technical Report 2008/001, ISBI research group, University of Klagenfurt (2008) `http://test-informations.info/`.
69. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H., Wang, K.: Forgetting in managing rules and ontologies. In: IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006), Hongkong, Dec. 2006, IEEE Computer Society (2006) 411–419 Preliminary version at ALPSWS 2006.
70. Hoehndorf, R., Loebe, F., Kelso, J., Herre, H.: Representing default knowledge in biomedical ontologies: Application to the integration of anatomy and phenotype ontologies. BMC Bioinformatics **8**(1) (2007) 377
71. de Bruijn, J., Eiter, T., Polleres, A., Tompits, H.: On representational issues about combinations of classical theories with nonmonotonic rules. In Lang, J., Lin, F., Wang, J., eds.: KSEM. Volume 4092 of Lecture Notes in Computer Science., Springer (2006) 1–22
72. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. ACM Transactions on Computational Logic **9**(2:14) (2008)
73. Drabent, W., Eiter, T., Ianni, G., Krennwallner, T., Lukasiewicz, T., Małuszyński, J.: Hybrid reasoning with rules and ontologies. In Bry, F., Małuszyński, J., eds.: Semantic Techniques for the Web: The REWERSE perspective. Number 5500 in LNCS. Springer (2009) 50 pp. To appear.
74. Polleres, A., Feier, C., Harth, A.: Rules with Contextually Scoped Negation. In: Proceedings of the 3rd European Conference on Semantic Web (ESWC 2006). Volume 4011 of LNCS., Springer (2006) 332–347
75. Damásio, C.V., Analyti, A., Antoniou, G., Wagner, G.: Supporting open and closed world reasoning on the web. In Alferes, J.J., Bailey, J., May, W., Schwertel, U., eds.: PPSWR. Volume 4187 of Lecture Notes in Computer Science., Springer (2006) 149–163
76. Analyti, A., Antoniou, G., Damásio, C.V.: A principled framework for modular web rule bases and its semantics. In: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR2008), AAAI Press (September 2008)
77. Alferes, J.J., Amador, R., May, W.: A general language for evolution and reactivity in the semantic web. In Fages, F., Soliman, S., eds.: PPSWR. Volume 3703 of Lecture Notes in Computer Science., Springer (2005) 101–115
78. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the Semantic Web. In Dubois, D., Welty, C., Williams, M.A., eds.: Proceedings Ninth International Conference on Principles of Knowledge Representation and

Reasoning (KR 2004), June 2-5, Whistler, British Columbia, Canada, Morgan Kaufmann (2004) 141–151

79. Baader, F.: Description logics. [126]
80. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Well-founded semantics for description logic programs in the Semantic Web. In: Proceedings RuleML-2004. Volume 3323 of LNCS., Springer (2004) 81–97
81. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R.: Well-founded semantics for description logic programs in the Semantic Web. Technical Report INFSYS RR-1843-09-01, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria (March 2009)
82. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective integration of declarative rules with external evaluations for semantic web reasoning. In Sure, Y., Domingue, J., eds.: Proceedings 3rd European Conference on Semantic Web (ESWC 2006). Number 4011 in LNCS, Springer (2006) 273–287
83. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In: International Joint Conference on Artificial Intelligence (IJCAI) 2005, Edinburgh, UK (August 2005) 90–96
84. Ross, K.A.: Modular stratification and magic sets for datalog programs with negation. Journal of the ACM **41**(6) (1994) 1216–1266
85. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. Journal of Logic Programming **15**(3) (February 1993) 187–230
86. Buccafurri, F., Leone, N., Rullo, P.: Strong and Weak Constraints in Disjunctive Datalog. In Dix, J., Furbach, U., Nerode, A., eds.: Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97). Volume 1265 of Lecture Notes in AI (LNAI)., Dagstuhl, Germany, Springer Verlag (July 1997) 2–17
87. Nieuwenborgh, D.V., Cock, M.D., Vermeir, D.: Computing Fuzzy Answer Sets Using dlvhex. In Dahl, V., Niemelä, I., eds.: Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Porto, Portugal, September 8-13, 2007. Volume 4670 of LNCS., Springer (2007) 449–450
88. Heymans, S., Toma, I.: Ranking Services Using Fuzzy HEX-Programs. In Calvanese, D., Lausen, G., eds.: Proceedings of the 2nd International Conference on Web Reasoning and Rule Systems (RR 2008), Karlsruhe, Germany, October 31-November 1, 2008. Volume 5341 of LNCS., Springer (2008) 181–196
89. Nieuwenborgh, D.V., Eiter, T., Vermeir, D.: Conditional Planning with External Functions. In Baral, C., Brewka, G., Schlipf, J.S., eds.: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). Volume 4483 of LNCS., Springer (2007) 214–227
90. Rosati, R.: $\mathcal{DL}$+log: Tight Integration of Description Logics and Disjunctive Datalog. In: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), AAAI Press (2006) 68–78
91. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence. (2007) 477–482
92. Motik, B., et al. (ed.): OWL 2 Web Ontology Language Profiles. W3C (2009), `http://www.w3.org/TR/owl-profiles/`
93. Lifschitz, V.: Nonmonotonic databases and epistemic queries. In: Proceedings IJCAI-91. (1991) 381–386
94. Lukasiewicz, T.: Probabilistic description logic programs. Int. J. Approx. Reasoning **45**(2) (2007) 288–307
95. Lukasiewicz, T., Straccia, U.: Description logic programs under probabilistic uncertainty and fuzzy vagueness. In: Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 9th European Conference, ECSQARU. (2007) 187–198

96. Analyti, A., Antoniou, G., Damásio, C.V., Wagner, G.: Stable Model Theory for Extended RDF Ontologies. In: Proc. Fourth International Semantic Web Conference (ISWC 2005). (2005) 21–36
97. Swift, T.: Deduction in Ontologies via ASP. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). Volume 2923 of Lecture Notes in AI (LNAI)., Fort Lauderdale, Florida, USA, Springer (January 2004) 275–288
98. Hustadt, U., Motik, B., Sattler, U.: Reducing SHIQ-description logic to disjunctive datalog programs. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada. (2004) 152–162
99. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Open answer set programming for the Semantic Web. J. Applied Logic **5**(1) (2007) 144–169
100. Ricca, F., Gallucci, L., Schindlauer, R., Dell'armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based System for Enterprise Ontologies. Journal of Logic and Computation (2008) doi:10.1093/logcom/exn042.
101. Ianni, G., Krennwallner, T., Martello, A., Polleres, A.: A Rule System for Querying Persistent RDFS Data. In Arroyo, L., Traverso, P., eds.: The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Greece, May 31–June 4, 2009, Proceedings. Volume 5554 of LNCS., Springer (2009) 857–862
102. Arenas, M., Gutierrez, C., Pérez, J.: Foundations of RDF databases. [126]
103. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Conceptual logic programs. Annals of Mathematics and Artificial Intelligence **47**(1-2) (2006) 103–137
104. de Bruijn, J., Pearce, D., Polleres, A., Valverde, A.: Quantified equilibrium logic and hybrid rules. In Marchiori, M., Pan, J.Z., de Sainte Marie, C., eds.: Proc. First International Conference on Web Reasoning and Rule Systems (RR07). Volume 4524 of LNCS. (2007) 58–72
105. Heymans, S., de Bruijn, J., Predoiu, L., Feier, C., Nieuwenborgh, D.V.: Guarded hybrid knowledge bases. Theory and Practice of Logic Programming **8**(3) (2008) 411–429
106. Bonatti, P.A.: Reasoning with infinite stable models. Artificial Intelligence **156**(1) (2004) 75–111
107. Baselice, S., Bonatti, P.A., Criscuolo, G.: On finitely recursive programs. In: Proc. 23rd International Conference on Logic Programming (ICLP 2007). Volume 4670 of Lecture Notes in Computer Science., Springer (2007) 89–103
108. Bonatti, P., Baselice, S.: Composing normal programs with function symbols. In de La Banda, M., Pontelli, E., eds.: Proceedings 24th International Conference on Logic Programming (ICLP 2008). Number 5366 in LNCS, Springer (2008) 425–439
109. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence **50**(3-4) (2007) 333–361
110. Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-1997). Volume 1265 of LNCS., Springer (1997) 290–309
111. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In Hill, P., Warren, D., eds.: Proceedings 25th International Conference on Logic Programming (ICLP 2009). Volume 5649 of LNCS., Springer (July 2009) 145–159
112. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning. Volume 4483 of LNCS., Springer (May 2007) 175–187
113. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for Smodels programs. Theory and Practice of Logic Programming **8**(5–6) (November 2008) 717–761

114. Tari, L., Baral, C., Anwar, S.: A Language for Modular Answer Set Programming: Application to ACC Tournament Scheduling. In: Proceedings of the 3Proceedings of the 3rd International ASP'05 Workshop, Bath, UK, 27th–29th July 2005. Volume 142 of CEUR Workshop Proceedings., CEUR WS (July 2005) 277–293
115. Balduccini, M.: Modules and Signature Declarations for A-Prolog: Progress Report. [123] Available at `http://sea07.cs.bath.ac.uk/downloads/sea07-proceedings.pdf`.
116. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro calls and Use of Ensembles in Modular Answer Set Programming. In: Proceedings of the 22th International Conference on Logic Programming (ICLP 2006). Number 4079 in LNCS, Springer (2006) 376–390
117. Calimeri, F., Ianni, G.: Template programs for Disjunctive Logic Programming: An operational semantics. AI Communications **19**(3) (2006) 193–206
118. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Trans. Comput. Log. **2**(4) (2001) 526–541
119. Eiter, T., Fink, M., Woltran, S.: Semantical Characterizations and Complexity of Equivalences in Answer Set Programming. ACM Trans. Comput. Log. **8**(3) (2007) Article 17 (53 + 11 pages).
120. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In Niemelä, I., Lifschitz, V., eds.: Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004). Number 2923 in LNCS, Springer (2004) 87–99
121. Woltran, S.: A common view on strong, uniform, and other notions of equivalence in answer-set programming. Theory and Practice of Logic Programming **8**(2) (2008) 217–234
122. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In Veloso, M.M., ed.: IJCAI. (2007) 372–379
123. de Vos, M., Schaub, T., eds.: Informal Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming, Tempe, AZ (USA), May 2007. In de Vos, M., Schaub, T., eds.: Informal Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming, Tempe, AZ (USA), May 2007. (2007) Available at `http://sea07.cs.bath.ac.uk/downloads/sea07-proceedings.pdf`.
124. Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufman, Washington DC (1988)
125. de la Banda, M.G., Pontelli, E., eds.: Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings. In de la Banda, M.G., Pontelli, E., eds.: ICLP. Volume 5366 of Lecture Notes in Computer Science., Springer (2008)
126. Franconi, E., Tessaris, S., eds.: Reasoning Web 2009. In Franconi, E., Tessaris, S., eds.: Reasoning Web 2009. LNCS, Springer (August 2009)

A Appendix: A `DLV` specification for the Sudoku problem

```
#maxint=9.

tab(X,Y,1) v tab(X,Y,2) v tab(X,Y,3) v
tab(X,Y,4) v tab(X,Y,5) v tab(X,Y,6) v
tab(X,Y,7) v tab(X,Y,8) v tab(X,Y,9)
   :- #int(X), 0 <= X, X <= 8, #int(Y), 0 <= Y, Y <= 8.

% Check rows and columns
:- tab(X,Y1,Z), tab(X,Y2,Z), Y1<>Y2.
:- tab(X1,Y,Z), tab(X2,Y,Z), X1<>X2.

% Check subtable
:- tab(X1,Y1,Z), tab(X2,Y2,Z), Y1 <> Y2,
   div(X1,3,W1), div(X2,3,W1),
   div(Y1,3,W2), div(Y2,3,W2).

:- tab(X1,Y1,Z), tab(X2,Y2,Z), X1 <> X2,
   div(X1,3,W1), div(X2,3,W1),
   div(Y1,3,W2), div(Y2,3,W2).

%Auxiliary: X divided by Y is Z
div(X,Y,Z) :- XminusDelta = Y*Z,
X = XminusDelta + Delta, Delta < Y.

% Table positions  X=0..8, Y=0..8
tab(0,1,6). tab(0,3,1). tab(0,5,4). tab(0,7,5).
tab(1,2,8). tab(1,3,3). tab(1,5,5). tab(1,6,6).
tab(2,0,2). tab(2,8,1). tab(3,0,8). tab(3,3,4).
tab(3,5,7). tab(3,8,6).
tab(4,2,6). tab(4,6,3).
tab(5,0,7). tab(5,3,9). tab(5,5,1). tab(5,8,4).
tab(6,0,5). tab(6,8,2).
tab(7,2,7). tab(7,3,2). tab(7,5,6). tab(7,6,9).
tab(8,1,4). tab(8,3,5). tab(8,5,8). tab(8,7,7).
```

B Appendix: Fixpoint Theorems of Knaster-Tarski and Kleene

Definition 18. *A complete lattice is a partially ordered set $(V, \leq)$ such that each subset $W \subseteq V$ has a least upper bound* $\sup(W)$ *and a greatest lower bound* $\inf(W)$.

Example 48. The partially ordered set $(V, \leq)$, where V is the set of all Herbrand interpretations of a program P and $\leq$ is set inclusion ($\subseteq$), is a complete lattice.

Definition 19. *An operator on a complete lattice $(V, \leq)$ is a mapping $T\colon V \to V$.*

Example 49. The T_P operator for a program P is an operator on Herbrand interpretations.

Definition 20. *An operator $T\colon V \to V$ on $(V, \leq)$ is* monotone*, if*

$$x \leq y \textit{ implies } T(x) \leq T(y)\ \forall x, y \in V \quad .$$

Monotone operators have nice fixpoint properties.

Theorem 13 (Knaster-Tarski). *Any monotone operator T on a complete lattice $(V, \leq)$ has a least fixpoint $\mathit{lfp}(T)$, and*

$$\mathit{lfp}(T) = \inf(\{x \in V \mid T(x) \leq x\}) \quad .$$

Example 50. The T_P operator for a (positive) program P is monotone.

A stronger theorem holds for continuous operators.

Definition 21. *A set $W \subseteq V$ is* directed, *if for each $x, y \in W$ there exists some $z \in W$ such that $x \leq z$ and $y \leq z$, where $(V, \leq)$ is a partial order.*

Definition 22. *An operator $T\colon V \to V$ on a complete lattice $(V, \leq)$ is* continuous, *if*

$$T(\sup(W)) = \sup(\{T(x) \mid x \in W\})$$

for every directed set $W \subseteq V$.

Intuitively, directedness models convergence (one can build a chain $x_0 < x_1 < \cdots$). It is not difficult to see that continuous operators are also monotone.

Example 51. The T_P operator is also continuous.

Theorem 14 (Kleene). *Any continuous operator T on a complete lattice $(V, \leq)$ has a least fixpoint, and*

$$\mathit{lfp}(T) = \sup(\{T^i \mid i \geq 0\}) \ ,$$

where $T^0 = \inf(V)$ and $T^{i+1} = T(T^i)$, for all integers $i \geq 0$.

Let $T^\infty = \sup(\{T^i \mid i \geq 0\})$. Note that if $T^i = T^{i-1}$ for some i, then $T^\infty = T^i$ holds; in particular, this is the case for T_P if the program P has no function symbols (given P is finite).

Remark. A weaker form of Kleene's Theorem holds for all monotone operators ($\mathit{lfp}(T)$ is constructible by a transfinite sequence T^α, for ordinals $\alpha \geq 0$).