Logic Programs with Abstract Constraint Atoms: The Role of Computations ⋆

Lengning Liu [a] Enrico Pontelli [*,b] Tran Cao Son [b]
Miroslaw Truszczyński [a]

[a] *Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA*

[b] *Department of Computer Science, New Mexico State University, Las Cruces, NM 88003, USA*

Abstract

We provide a new perspective on the semantics of logic programs with *arbitrary abstract constraints*. To this end, we introduce several notions of *computation*. We use the *results* of computations to specify answer sets of programs with constraints. We present the rationale behind the classes of computations we consider, and discuss the relationships among them. We also discuss the relationships among the corresponding concepts of answer sets. One of those concepts has several compelling characterizations and properties, and we propose it as the *correct* generalization of the answer-set semantics to the case of programs with arbitrary constraints. We show that several other notions of an answer set proposed in the literature for programs with constraints can be obtained within our framework as the results of appropriately selected classes of computations.

Key words: Logic programs with abstract constraint atoms, answer sets, computations

1 Introduction and Motivation

We study logic programs with *arbitrary abstract constraints*, or simply, *constraints*. Programs with constraints provide a general framework to study semantics of extensions of logic programs with aggregates. It is due to the fact that normal logic programs, programs with monotone and convex constraints (proposed by Marek

⋆ An extended abstract of this paper appeared in the proceedings of the 2007 International Conference on Logic Programming.

* Corresponding author
Email addresses: lliu1@cs.uky.edu (Lengning Liu), epontell@cs.nmsu.edu (Enrico Pontelli), tson@cs.nmsu.edu (Tran Cao Son), mirek@cs.uky.edu (Miroslaw Truszczyński).

and Truszczyński (2004); Liu and Truszczyński (2005)), and several classes of programs with aggregates (e.g., (Dell'Armi et al., 2003; Faber et al., 2004; Pelov, 2004; Son and Pontelli, 2007)) can be viewed as special programs with arbitrary constraints.

The original definition of the syntax of programs with constraints, along with a possible semantics, has been proposed by Marek and Remmel (2004). An alternative semantics was later proposed by Son, Pontelli, and Tu (2007), and revisited by Shen and You (2007) and by You, Yuan, Liu, and Shen (2007).

In this paper, we introduce a general framework for defining and investigating semantics for programs with constraints. We base our development on the notion of *computation*. The proposed framework builds on general principles that can be elicited from the semantics of traditional normal logic programs (i.e., logic programs with negation as failure).

The *answer-set semantics* of logic programs was introduced by Gelfond and Lifschitz (1990). The semantics generalizes the *stable-model semantics* of Gelfond and Lifschitz (1988), which was proposed for the class of normal logic programs only, to logic programs with two negations (negation as failure and classical negation). In the paper, we consistently use the term answer-set semantics, as it is currently more widely used, and as the bulk of our paper is concerned with programs that are not normal.

The answer-set semantics forms the foundation of *Answer-Set Programming (ASP)* (Marek and Truszczyński, 1999; Niemelä, 1999; Gelfond and Leone, 2002). Intuitively, an answer set of a program represents the set of *"justified"* beliefs of an agent, whose knowledge is encoded by the program. Over the years, researchers have developed several characterizations of answer sets, that identify and emphasize their key features and suggest ways to compute them.

The original definition of answer sets (Gelfond and Lifschitz, 1988) introduces a "guess-and-check" approach to computing answer sets of a program. The process starts by guessing an interpretation, to be used as a candidate answer set, and then proceeds in validating it. The validation consists of recomputing the guessed interpretation, starting from the empty set and iteratively applying the *immediate consequence operator* (van Emden and Kowalski, 1976) for the *Gelfond-Lifschitz* reduct of the program (Gelfond and Lifschitz, 1988). The interpretation is accepted only if it is the limit of this iterative process In this approach, once the guess is made, the validation is entirely deterministic.

Other characterizations of answer sets suggest an alternative scheme for constructing answer sets. The process starts, also in this case, from the empty set. At each step, we add to the set under construction the heads of *some* of the rules applicable at that step. Typically, we use all the rules selected during the previous steps (if they are no longer applicable, the construction terminates with failure) *plus* some additional ones. When the process stabilizes—i.e., no new elements can be introduced in the set—the result is an answer set (Marek, Nerode, and Remmel, 1999; Marek and Truszczyński, 1993). In this approach, we replace the initial non-deterministic step of guessing an entire interpretation with local non-deterministic choices of rules to fire at each step of the construction. Similarly, the task of validation is dis-

tributed across the computation. Observe that this approach for characterizing answer sets represents the underlying model that is employed by several ASP solvers, where 3-valued partial models (Van Gelder, Ross, and Schlipf, 1991) are extended to stable models.

Example 1 *Let us consider the program P_1 consisting of the following rules:*

$$a \leftarrow \mathbf{not}\ b$$

$$b \leftarrow \mathbf{not}\ a$$

$$c \leftarrow a$$

$$d \leftarrow b$$

This program has two answer sets: $\{a, c\}$ and $\{b, d\}$. [1]
In the "guess-and-check" approach, we might guess $\{a, c\}$ as a candidate answer set. To verify the guess, we compute the Gelfond-Lifschitz reduct, consisting of the rules:

$$a \leftarrow$$

$$c \leftarrow a$$

$$d \leftarrow b$$

The validation requires determining the least fixpoint of the immediate consequence operator of the reduct program—i.e., the least Herbrand model of the reduct program—which corresponds to $\{a, c\}$. Since it coincides with the initial guess, the guess is validated as an answer set. In the same way, we can also validate the guess $\{b, d\}$. However, the validation of $\{a\}$ fails—since the reduct program contains the rules

$$a \leftarrow$$

$$c \leftarrow a$$

$$d \leftarrow b$$

and the iteration of its immediate consequence operator converges to $\{a, c\}$, which is different from the initial guess $\{a\}$.
The alternative approach we mentioned starts with the empty interpretation, $\emptyset$, which makes two rules applicable: $a \leftarrow \mathbf{not}\ b$ and $b \leftarrow \mathbf{not}\ a$. The algorithm needs to select some of them for application, say, it selects $a \leftarrow \mathbf{not}\ b$. The choice results in the new interpretation $\{a\}$. Two rules are applicable now: $a \leftarrow \mathbf{not}\ b$ and $c \leftarrow a$. Let us select both rules for application. The resulting interpretation is $\{a, c\}$. The same two rules that were applicable in the previous step are still applicable and no other rules are applicable. Thus, there is no possibility to add new

[1] Since neither in this example, nor anywhere else in the paper, do we consider classical negation, following the tradition of logic programming, we describe an answer set using a set of atoms—that contains all the atoms that are true; the remaining atoms are considered false by default.

elements to the current set. The computation stabilizes at $\{a, c\}$*, thus making* $\{a, c\}$ *an answer set.* □

We note that the first approach starts with a tentative answer set of the program, while the second starts with the *empty* interpretation. In the first approach, we guess the entire answer set at once and, from that point on, proceed in a deterministic fashion. In the second approach we construct an answer set *incrementally* making non-deterministic choices along the way. Thus, each approach involves non-determinism. However, in the second approach, the role of non-determinism could be potentially more limited.

In this paper, we cast these two approaches in terms of abstract principles related to a notion of *computation*. We then lift these principles to the case of programs with abstract constraints and derive from the approach a well-motivated semantics for such programs.

The recent interest in ASP has been fueled by the development of inference engines to compute answer sets of logic programs, most notably systems like SMODELS (Niemelä and Simons, 1997), CMODELS (Lierler and Maratea, 2004), CLASP (Gebser et al., 2007) and DLV (Leone, Pfeifer, Faber, Eiter, Gottlob, Perri, and Scarcello, 2006), which allow programmers to tackle complex real-world problems (e.g., (Balduccini, Gelfond, and Nogueira, 2006; Heljanko and Niemelä, 2003; Erdem, Lifschitz, and Ringe, 2006)). To facilitate declarative solutions of problems in knowledge representation and reasoning, researchers proposed extensions of the logic programming language, which support *aggregates* (Niemelä, 1999; Dell'Armi, Faber, Ielpa, Leone, and Pfeifer, 2003; Denecker, Pelov, and Bruynooghe, 2001; Faber, Leone, and Pfeifer, 2004; Gelfond, 2002; Pelov, 2004; Simons, Niemelä, and Soininen, 2002).

These development efforts stimulated interest in logic programming formalisms based on *abstract constraint atoms*, originally proposed by Marek and Remmel (2004) and Marek and Truszczyński (2004). The objective was not to introduce a knowledge representation language but rather an abstract framework, in which one could study semantics of knowledge representation systems obtained by extending the syntax of logic programs aggregates. The need arose as the introduction of constraints and aggregates into logic programming created a challenge to extend the semantics. Researchers proposed several possible approaches (Faber et al., 2004; Denecker et al., 2001; Son and Pontelli, 2007; Son et al., 2007; Elkabani et al., 2004; Son et al., 2006). These approaches *all* agree on large classes of programs, including

- Normal logic programs (every extension contains that class),
- Programs with *monotone* aggregates such as weight atoms with all weights non-negative and without the upper bound given (they can be regarded as special programs with monotone constraints, as presented by Marek and Truszczyński (2004)), and
- Programs with *convex* aggregates such as weight atoms with all weights non-negative and with both lower and upper bounds given (they can be regarded as special programs with monotone constraints, as presented by Liu and

Truszczyński (2005).
However, the proposed approaches tend to differ on programs with arbitrary aggregates.
What makes the task of defining answer sets for programs with aggregates difficult and interesting is the *non-monotonic* behavior of such constraints. For instance, let us consider the constraint $(\{p(1), p(-1)\}, \{\emptyset, \{p(1), p(-1)\}\})$,[2] which can be seen as an encoding of the aggregate

$$\textsc{Sum}(\{X \mid X \in \{-1, 1\} \wedge p(X)\}) = 0.$$

This aggregate atom is true in the interpretations $\emptyset$ and $\{p(1), p(-1)\}$, but false in $\{p(1)\}$ and $\{p(-1)\}$. Therefore it is not monotone; observe that it is also not convex.
In this paper, we propose and study a general framework for defining semantics (different types of answer sets) of logic programs with constraints. Our proposal relies on the notion of (incremental) *computation*. We introduce several classes of computations and use their results to define different types of answer sets. The approach can be traced back to the two basic methods to characterize answer sets of normal logic programs that we mentioned above.
The notion of a computation we introduce and use here generalizes those developed by Marek and Truszczyński (2004) and Liu and Truszczyński (2005) for programs with monotone and convex constraints. We study properties of the various classes of computations introduced and of the corresponding notions of answer sets. We relate these computation-based concepts of answer sets to earlier proposals. An interesting observation of our investigation is that several characterizations converge to the same semantics—which correspond to the one proposed by Son, Pontelli, and Tu (2007).
A preliminary approach to the notion of computation investigated in this paper has been presented by Liu, Pontelli, Son, and Truszczyński (2007)—the present manuscript deeply revises and reorganizes the ideas from that work, leading to definitions and results that are significantly different and more advanced those presented earlier.
The contributions presented in this work are of importance not only to the field of logic programming, but the overall domain of knowledge representation. Answer-Set Programming has gained a momentum as an instrument for the design of intelligent agents and for investigating properties of reasoning in complex domains (e.g., domains with multiple interacting agents, domains with incomplete knowledge, domains with non-deterministic actions). Several of the recently explored extensions of languages used in Answer-Set Programming have been motivated by the needs of applications in these areas of research (e.g., (Eiter, Faber, Leone, Pfeifer, and Polleres, 2003a,b)). In the invited talk at the AAAI'05 conference, given by Baral (2005), logic programming under the answer-set semantics has been presented as an attractive and suitable knowledge representation language for AI research, as it

[2] We introduce this notation in Section 4.

features several desirable properties. In particular, the formalism:

- is declarative and has a simple syntax;
- is non-monotonic and is expressive enough for representing several classes of problems in the complexity hierarchy (Dantsin, Eiter, Gottlob, and Voronkov, 2001);
- has solid theoretical foundations with a large body of building block results (Baral, 2003)—e.g., equivalence between programs, systematic program development, relationships to other non-monotonic formalisms; and
- is supported by several efficient computational tools, as those mentioned earlier.

The general framework of abstract constraints provides a foundation for the investigation of generalizations of current ASP formalisms to meet the needs of knowledge representation and reasoning applications.

2 Normal Logic Programs and Answer-Set Semantics

A normal logic program P is a set of *rules* of the form

$$a \leftarrow a_1, \ldots, a_m, \mathbf{not}\ a_{m+1}, \ldots, \mathbf{not}\ a_n \tag{1}$$

where $0 \leq m \leq n$, each a_i is an atom in a first-order language $\mathcal{L}$ and $\mathbf{not}$ is the *negation-as-failure* (*default* negation) connective. An expression of the form $\mathbf{not}\ a$, where a is an atom, is a *default literal*. A *literal* is an atom or a default literal. A program is *positive* (or *Horn*) if it does not contain default literals.

The *Herbrand universe* and the *Herbrand base* of a program are defined in the standard way (Lloyd, 1987). All major semantics of programs are restricted to Herbrand interpretations and, under each of them, a program and its ground instantiation are equivalent. That includes the answer-set semantics, which is of interest to us here. Ground programs under the semantics of Herbrand models are, essentially, propositional programs. Thus, from now on, we consider only propositional programs over a fixed countable set At of propositional atoms.

For a propositional rule r of the form (1), a is the *head* of r. We denote it by $hd(r)$. We set $pos(r) = \{a_1, \ldots, a_m\}$, and $neg(r) = \{a_{m+1}, \ldots, a_n\}$. Finally, we denote with $body(r)$ the *body* of the rule r, that is, the set of the literals in the right-hand side of the rule r.

We represent Herbrand interpretations as subsets of At. An atom a is satisfied by an Herbrand interpretation $M \subseteq At$ if $a \in M$. A default literal $\mathbf{not}\ a$ is satisfied by M if $a \notin M$. We write $M \models \ell$ to denote that a literal ℓ is satisfied by M. Similarly, if S is a set (conjunction) of literals, we write $M \models S$ to denote that $M \models \ell$, for every $\ell \in S$.

A rule r is *M-applicable* if $M \models body(r)$. We denote by $P(M)$ the set of all M-applicable rules in P. An atom a is *supported* by M in P if a is the head of at least one M-applicable rule $r \in P$. An interpretation M satisfies P, or is a *model* of P, if it contains all atoms supported by M in P. An interpretation M is a *supported* model of P if M is a model of P and every atom in M is supported by M in P. Observe that not every model of a program is a supported model. For example, the

set $\{a, b\}$ is a model of a program $P = \{a \leftarrow \mathbf{not}\ b\}$; however, it is not a supported model of P, since neither a nor b is supported by $\{a, b\}$.
The *immediate consequence operator*, also referred to as the *one-step provability operator*, maps interpretations to interpretations. Specifically, it assigns to an interpretation M the set of the heads of all rules in $P(M)$, that is, the set of all atoms that are supported by M in P. We denote this operator by T_P. Formally,

$$T_P(M) = \{hd(r)\ :\ M \models body(r), \text{for some } r \in P\}.$$

One can check that the fixpoints of T_P are supported models of P (Apt, 1990).
For later use, let us introduce the following notion of iterated applications of T_P. Let $\langle T_P{\uparrow}i\rangle_{i=0}^{\infty}$ be the sequence:

$$\begin{aligned} T_P{\uparrow}0 &= \emptyset \\ T_P{\uparrow}(i+1) &= T_P(T_P{\uparrow}i). \end{aligned}$$

We note that if P is a positive program then T_P is a monotone (and continuous) operator whose least fixpoint, denoted by $\mathit{lfp}(T_P)$, is the (unique) least Herbrand model of P. It is well known that $\mathit{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P{\uparrow}i$. Many other properties of T_P have been discussed, for example, by Lloyd (1987).
The *Gelfond-Lifschitz* reduct of P w.r.t. M, denoted by P^M, is the program obtained from P by deleting
(1) each rule whose body contains a default literal $\mathbf{not}\ a$ such that $a \in M$, and
(2) all default literals in the bodies of the remaining rules.
Note that P^M is a positive program whose least model is $\mathit{lfp}(T_{P^M})$. An interpretation M is a *stable model* or, as we will say here, an *answer set* of P, if and only if M is the least model of P^M. Thus, M is an answer set of P if and only if $M = \mathit{lfp}(T_{P^M})$. It also follows directly from the definitions that the least model of a positive program is the only answer set of that program.

3 Computations in Normal Logic Programs: Principles

We start by motivating the notion of a *computation*, which is central to our paper, as a tool to determine answer sets of a normal logic programs. Our starting point is the collection of considerations introduced in Example 1. In particular, we show how to use computations to characterize answer sets.
We define computations for a normal program P as sequences $\langle X_i\rangle_{i=0}^{\infty}$ of sets of atoms (propositional interpretations), where X_i represents the state of the computation at step i. In particular, we *require* that $X_0 = \emptyset$. A key intuition is that at each step $i \geq 1$, we use P to revise the state X_{i-1} of the computation into its new state X_i. We base the revision on a non-deterministic operator, $Concl_P$. Given an interpretation X, $Concl_P(X)$ consists of all possible revisions of X that are *"grounded"* in P.
Formally, a set of atoms Y is *grounded* in a set of atoms X and a program P if $Y \subseteq$

$T_P(X)$, that is, if every atom in Y is supported by X in P. Thus, $Concl_P(X) = \{Y \mid Y \subseteq T_P(X)\}$.
The first principle we will impose on computations formalizes the way in which states of computations are revised. We call it the *principle of revision*.

($\mathbf{R}$) *Revision*: each successive element in a computation must be grounded in the preceding one and the program, that is, $X_i \in Concl_P(X_{i-1})$, for every $i \geq 1$.

Computations of answer sets of a program, using the methods described in Example 1, produce sequences of sets that are monotonically growing (w.r.t. set inclusion), each set being a part of the answer set under construction. Thus, at each step not only new atoms are computed, but also all atoms established earlier are recomputed. This suggests another principle for computations, the principle of *persistence of beliefs*:

($\mathbf{P}$) *Persistence of beliefs*: each next element in the computation must contain the previous one (once we "revise an atom in", we keep it), that is, $X_{i-1} \subseteq X_i$, for every i, $1 \leq i$.

For a sequence $\langle X_i \rangle_{i=0}^{\infty}$ satisfying the principle ($\mathbf{P}$), we define $X_\infty = \bigcup_{i=0}^{\infty} X_i$ and we refer to X_∞ as the *result* of $\langle X_i \rangle_{i=0}^{\infty}$. The result of a computation should be an interpretation that cannot be revised any further. This suggests one additional basic principle for computations, the principle of *convergence*:

($\mathbf{C}$) *Convergence*: a computation continues until it stabilizes (no additional revisions can be made). Formally speaking, convergence requires $X_\infty = T_P(X_\infty)$, where T_P is the one-step provability operator for P. (In particular, convergence implies that X_∞ is a supported model of P).

These observations can be summarized in the following definition.
Definition 1 *Let P be a normal logic program. A sequence of interpretations $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for P if $X_0 = \emptyset$ and $\langle X_i \rangle_{i=0}^{\infty}$ satisfies the principles ($\mathbf{R}$), ($\mathbf{P}$) and ($\mathbf{C}$).*
Computations are relevant to the task of describing answer sets of normal logic programs. We have the following result.
Proposition 1 *Let P be a normal logic program. If a set of atoms X is an answer set of P then there exists a computation $\langle X_i \rangle_{i=0}^{\infty}$ for P such that $X = X_\infty$.*
Proof If X is an answer set of P then X is the least fixpoint of T_{PX}. Let us define $X_i = T_{PX} \uparrow i$. It follows from the property of the operator T_{PX} that $X = lfp(T_{PX}) = \bigcup_{i=0}^{\infty} T_{PX} \uparrow i = \bigcup_{i=0}^{\infty} X_i$. To complete the proof, it suffices to show that $\langle X_i \rangle_{i=0}^{\infty}$ is a computation for P.
First, we note that T_{PX} is a monotone operator and $X_0 = \emptyset$. It follows that for every $i \geq 1$, $X_{i-1} \subseteq T_{PX}(X_{i-1}) = X_i$. Thus, the principle of persistence of beliefs holds. We also have that for every $i \geq 0$, $X_i \subseteq X$ and $X = X_\infty$. Consequently, for every $i \geq 1$, $X_i = T_{PX}(X_{i-1}) \subseteq T_P(X_{i-1})$, that is, the principle of revision holds. Finally, since X is an answer set of P, X is a supported model of P. Consequently, $T_P(X) = X$ and the principle of convergence holds. □

Proposition 1 implies that the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ give a notion of computation broad enough to encompass all answer sets. Is this concept of computation what is needed to characterize answer sets? In other words, does *every* sequence of sets of atoms starting with the $\emptyset$ and satisfying the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ result in an answer set? It is indeed the case for *positive* programs.

Proposition 2 *Let P be a positive logic program. The result of every computation is equal to the least model of P, that is, the unique answer set of P.*

Proof Let $\langle X_i \rangle_{i=0}^{\infty}$ be a computation for P and M be the least model of P. Since P is positive, we have that T_P is a monotone operator. By definition of a computation and the monotonicity of T_P, $X_i \subseteq T_P \uparrow i$. This implies that $X_\infty \subseteq M$ since $M = \bigcup_{i=0}^{\infty} T_P \uparrow i$. On the other hand, since M is the least model of P and X_∞ is also a model of P, we have that $M \subseteq X_\infty$. This proves the proposition. □

However, in the case of arbitrary normal programs, there are computations that do not result in answer sets, as shown in the following example.

Example 2 *Let us consider the program P_2 containing the two rules*

$$a \leftarrow \mathbf{not}\ a$$

$$a \leftarrow a$$

This program has no answer sets. The sequence $X_0 = \emptyset$, $X_1 = \{a\}$, $X_2 = \{a\}$, ... satisfies $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$, thus it is a computation for P. However, $X = \bigcup_{i=0}^{\infty} X_i = \{a\}$ is not an answer set of P_2. □

It follows that the notion of computation defined by the principles $(\mathbf{R})$, $(\mathbf{P})$ and $(\mathbf{C})$ is too broad to capture precisely the notion of answer set. Let us reconsider Example 2. In that example, $a \in X_1$ because the body of the first rule is satisfied by the interpretation $\emptyset$. However, the body of the first rule is *not* satisfied in any set X_i for $i \geq 1$. On the other hand, $a \in X_i$, for $i \geq 2$, since the body of the *second* rule is satisfied by X_{i-1}. Thus, the reason for the presence of a in the next revision *changes* between the first and the second step. This is the reason why the computation does not result in an answer set, even though it satisfies the principle $(\mathbf{P})$.

These considerations suggest that useful classes of computations can be obtained by requiring that not only atoms, but also the *reasons* for including atoms persist. Intuitively, we would like to associate with each atom included in X_i a rule that supports the inclusion, and this rule should remain applicable from that point on. More formally, we state this principle as follows:

> (**Pr**) *Persistence of Reasons*: for every $a \in X_\infty$ there is a rule $r_a \in P$ (called the *reason* for a) whose head is a and whose body holds in every X_i, $i \geq i_a - 1$, where i_a is the least integer such that $a \in X_{i_a}$.

It turns out that persistence of reasons is exactly what is needed to characterize answer sets of normal logic programs.

Definition 2 *Let P be a normal logic program. A computation $\langle X_i \rangle_{i=0}^{\infty}$ for P is* persistent *if it satisfies the principle $(\mathbf{Pr})$.*

The next proposition shows that the principle of persistence of reasons is exactly what we need to capture the answer-set semantics by computations.

Proposition 3 *Let P be a normal logic program. A set X is an answer set of P if and only if there is a persistent computation for P whose result is X.*

Proof Let X be an answer set of P. It is easy to see that the computation constructed in the proof of Proposition 1, whose result is X, is a persistent computation for P. That proves the "only-if" part of the proposition.

Let $\langle X_i \rangle_{i=0}^{\infty}$ be a persistent computation for P and $X = X_\infty$. First, we observe that for every $i \geq 0$, $X_i \subseteq T_{P^X}{\uparrow}i$. We prove that by induction. The base case being evident, we proceed to the induction step and consider $a \in X_{i+1}$. By the persistence of reasons, there is a rule r_a and an integer $i_a \leq i$ such that $hd(r_a) = a$ and $X_j \models body(r_a)$, for every $j \geq i_a$. It follows that for every default literal $\mathbf{not}\ b \in body(r_a)$, $b \notin X$. Thus, $r_a^X \in P^X$, where with r_a^X we denote the rule obtained from r_a by removing all default literals from the body of r_a. Since $X_{i_a} \models body(r_a^X)$, $i_a \leq i$, and $X_{i_a} \subseteq X_i$, we have that $a \in T_{P^X}(X_i)$. By the induction hypothesis and the monotonicity of T_{P^X}, $a \in T_{P^X}(T_{P^X}{\uparrow}i)$. It follows that $a \in T_{P^X}{\uparrow}(i+1)$ and so, $X_{i+1} \subseteq T_{P^X}{\uparrow}(i+1)$. That completes the induction and shows that $X \subseteq \bigcup_{i=0}^{\infty} T_{P_X}{\uparrow}i = \mathit{lfp}(T_{P^X})$.

On the other hand, since X is a supported model of P, we have that X is also a model of P^X. Hence, $\mathit{lfp}(T_{P^X}) \subseteq X$. Thus, $X = \mathit{lfp}(T_{P^X})$, that is, X is an answer set of P, which proves the "if" part of the proposition and completes the proof. □

In general, the operator $Concl_P$ offers several choices for revising the current interpretation X_{i-1} to X_i during a computation. A natural question is whether this freedom is needed, or whether we can restrict the principle $(\mathbf{R})$ without loosing the ability to characterize the answer sets of normal logic programs.

Example 3 *Let P_3 be the normal logic program:*

$$a \leftarrow \mathbf{not}\ b$$

$$c \leftarrow \mathbf{not}\ b$$

$$e \leftarrow a, c$$

$$f \leftarrow a, \mathbf{not}\ c$$

This program has only one answer set $M = \{a, c, e\}$, that can be generated by the computation:

$$\emptyset, \quad \{a, c\}, \quad \{a, c, e\}.$$

In this computation, at each step $i = 1, 2$, we select X_i to be the greatest element of $Concl_{P_3}(X_{i-1})$, which exists and is given by $T_{P_3}(X_{i-1})$. Thus, the next element of the computation is the result of firing all applicable *rules.*

*On the other hand, selecting an element in $Concl_{P_3}(X)$ other than $T_{P_3}(X)$ can result in sequences that cannot be extended to a computation. For example, the sequence $\emptyset$, $\{a\}$, $\{a, f\}$ represents a potential computation since it satisfies the (**R**) and (**P**) principles. Yet, no possible extension of this sequence satisfies the (**C**)*

principle. □

This example indicates that interesting classes of computations can be obtained by restricting the operator $Concl_P$. Since for every X we have that $T_P(X) \in Concl_P(X)$, we could restrict the choice for possible revisions of X based on P to $T_P(X)$ only. The class of computations obtained under this restriction is a *proper* subset of the class of computations. For instance, the program P_1 from Example 1 does not admit computations that revise X_0 into $X_1 = T_P(X_0)$. Thus, the class of such computations is not adequate for the task of characterizing answer sets of normal logic program. We note, however, that they do characterize answer sets for certain special classes of logic programs, for instance, for stratified logic programs (Apt, 1990).

To obtain a general characterization of answer sets by restricting the choices offered by $Concl_P(X)$, we need to modify the operator $T_P(X)$. The first approach to computing answer sets, discussed in the introduction provides a clue: we need to modify the notion of satisfiability used in the definition of $T_P(X)$. Let M be an interpretation. We define the satisfiability relation $\models_M$, between sets of atoms and conjunctions of literals, as follows: given a set of atoms S and a conjunction of literals F, the relation $S \models_M F$ holds if $S \models F$ and $M \models F$. That is, the satisfaction is based not only on S (i.e., the current state of the computation), but also on M (the "context" of the computation). We can define the context-based one-step provability operator T_P^M as follows:

$$T_P^M(X) = \{hd(r) \mid X \models_M body(r), \text{for some } r \in P\}.$$

We note that $T_P^M(X) \subseteq T_P(X)$ and, consequently, $T_P^M(X) \in Concl_P(X)$. Thus, we obtain the following result.

Proposition 4 *Let P be a normal logic program and M be an interpretation. A sequence $\langle X_i \rangle_{i=0}^{\infty}$, where $X_i = T_P^M(X_{i-1})$ for $i = 1, 2, \ldots$, is a computation for P if and only if it satisfies the principles* $(\mathbf{P})$ *and* $(\mathbf{C})$.

Given an interpretation M, the sequence $\langle X_i \rangle_{i=0}^{\infty}$ such that $X_i = T_P^M(X_{i-1})$, for $i = 1, 2, \ldots$, is uniquely determined by M. Whenever the sequence satisfies the principles $(\mathbf{P})$ and $(\mathbf{C})$, we will refer to it as the *M-computation*. We observe that not all M-computations define answer sets, as illustrated in the following example.

Example 4 *Let P_4 be the normal logic program:*

$$a \leftarrow a$$

$$a \leftarrow \mathbf{not}\ b$$

$$b \leftarrow a$$

Let $M = \{a\}$. One can check that for each rule $r \in P_4$, $M \models body(r)$. Thus, for every set X of atoms, $T_P^M(X) = T_P(X)$. Consequently, $\emptyset$, $\{a\}$, $\{a, b\}, \ldots$ is an M-computation. However, $\{a, b\}$ is not an answer set of P_4. □

The problem is that M-computations may fail to be persistent. In fact, the M-

computation described in Example 4 does not satisfy the persistence of reasons principle. However, persistent M-computations, being special persistent computations, do result in answer sets. Moreover, every answer set is the result of a persistent M-computation.

Proposition 5 *Let P be a normal logic program. A set $M \subseteq At$ is an answer set of P if and only if the M-computation is persistent and its result is M.*

Proof If M is the result of a persistent X-computation, then M is, in particular, the result of a persistent computation. Thus, by Proposition 3, M is an answer set.

On the other hand, let us assume that M is an answer set of P. We observe that for every set of atoms Y, $T_{P(M)}(Y) \subseteq T_{P^M}(Y)$. Moreover, if $Y \subseteq M$, the converse inclusion holds, too. Indeed, if $Y \subseteq M$ and $a \in T_{P^M}(Y)$ then there is a rule $r \in P$ such that $hd(r) = a$, $M \models body(r)$, and $Y \models body(r^M)$ (where r^M denotes the rule obtained from r by removing from $body(r)$ all default literals). In particular, it follows that $r \in P(M)$ and $a \in T_{P(M)}(Y)$. The two inclusions together imply that if $Y \subseteq M$, then $T_{P(M)}(Y) = T_{P^M}(Y)$.

We now observe that $T_P^M(Y) = T_{P(M)}(Y)$. Thus, for every $Y \subseteq M$, $T_P^M(Y) = T_{P^M}(Y)$. It follows that the computation constructed in the proof of Proposition 1 is an M-computation and it is persistent (Proposition 3). Since its result is M, the assertion follows. □

An even stronger result can be proved, in which answer sets are characterized by a proper subclass of persistent M-computations. We call an M-computation *self-justified* if its result is M. In general, the class of self-justified M-computations is a proper subclass of M-computations. Indeed, as shown in Example 4, there are M-computations that are not persistent while, as we prove below, self-justified M-computations satisfy the persistence of reasons property. We use that fact to show that self-justified computations do indeed characterize answer sets.

Proposition 6 *Let P be a normal logic program. A set of atoms M is an answer set of P if and only if the M-computation is self-justified.*

Proof In the proof of Proposition 5, we showed that if M is an answer set of P, then it generates an M-computation with the result M, which proves one direction of the proposition.

Conversely, let M be a set of atoms that determines a self-justified M-computation and let $\langle X_i \rangle_{i=0}^{\infty}$ be that computation. Since $\langle X_i \rangle_{i=0}^{\infty}$ is a computation, to conclude the desired result it suffices to show that $\langle X_i \rangle_{i=0}^{\infty}$ is persistent—then the result will be immediate from Proposition 3.

Thus, let $a \in M$, and let i be the least integer such that $a \in X_i$. It follows that $i \geq 1$ and $a \in T_P^M(X_{i-1})$. Thus, there is a rule $r \in P(M)$ such that $M \models body(r)$ and $X_{i-1} \models body(r)$. For every $j \geq i$, $X_{i-1} \subseteq X_j \subseteq M$. Thus, $X_j \models body(r)$ and so the persistence of the computation follows. □

We can summarize our discussion in this section as follows. Our goal was to characterize answer sets of normal logic programs in terms of computations. More specifically, taking two ways of computing answer sets as the starting point, we introduced three characterizations of answer sets in terms of computations: persistent compu-

tations, persistent M-computations, and self-justified M-computations, with each subsequent class being a proper subclass of the preceding one. In Sections 5 and 6, we will show how to generalize the classes of computations discussed here to the case of programs with constraints. We will use these generalized computations to define and characterize answer sets of such programs.

4 Programs with Abstract Constraints: Basic Definitions

We will recall here some basic definitions concerning programs with constraints (Marek and Remmel, 2004; Marek and Truszczyński, 2004; Liu and Truszczyński, 2005). As before, we fix a countable infinite set At of propositional atoms. An *arbitrary abstract constraint* (or, simply, a *constraint*) is an expression $A = (X, C)$, where $X \subseteq At$ is a *finite* set, and $C \subseteq \mathcal{P}(X)$—where $\mathcal{P}(X)$ denotes the powerset of X. The set X is called the *domain* of A, while the elements of C are called *satisfiers* of A. Given a constraint $A = (X, C)$, we denote X with A_{dom} and C with A_{sat}. Intuitively, the sets in A_{sat} are precisely those subsets of A_{dom} that *satisfy* the constraint.

It is common to recognize special types of constraints:

- A constraint is *inconsistent* if it has no satisfiers. We will distinguish a special inconsistent constraint, $(\emptyset, \emptyset)$, and we will denote it by $\bot$.
- A constraint A is *monotone* if, for every $X \in A_{sat}$ and for every Y such that $X \subseteq Y \subseteq A_{dom}$, we have that $Y \in A_{sat}$.
- A constraint A is *convex* if for every $X, Y \in A_{sat}$ and for every Z such that $X \subseteq Z \subseteq Y$, we have that $Z \in A_{sat}$.

Constraints are building blocks of rules and programs. A *rule* is an expression

$$A \leftarrow A_1, \ldots, A_k \tag{2}$$

where A, $A_1, \ldots, A_k$ are constraints. A *constraint program* (or a *program*) is a collection of rules. A program is *monotone* (*convex*) if every constraint occurring in it is monotone (convex).

Given a rule r of the form (2), the constraint A is the *head* of r and the set of constraints $\{A_1, \ldots, A_k\}$ is the *body* of r; sometimes we view the body of a rule as the *conjunction* of its constraints. Following the notation introduced earlier, we denote the head and the body of r with $hd(r)$ and $body(r)$, respectively. We define the *headset* of r ($hset(r)$) to be the domain of the head of r, that is, $hset(r) = hd(r)_{dom}$. For a set of rules P, we define $hset(P) = \cup_{r \in P} hset(r)$.

We view subsets of At as interpretations. We say that $M \subseteq At$ *satisfies* a constraint A, denoted by $M \models A$, if $M \cap A_{dom} \in A_{sat}$. For a rule r, M satisfies r, denoted by $M \models r$, if M satisfies $hd(r)$ or M does not satisfy some constraint in $body(r)$. An interpretation M is a model of a program P if it satisfies all rules in P.

Let M be an interpretation. A rule is *M-applicable* if M satisfies every constraint in $body(r)$, i.e., $M \models body(r)$. As in Section 2, we denote with $P(M)$ the set of all M-applicable rules in P. Let P be a program. A *model* M of P is *supported* if $M \subseteq hset(P(M))$. Observe that, from this definition, one can conclude that if a

model M is supported then M will satisfy $hd(r)$ for every rule r applicable in M. Let P be a program and M a set of atoms. A set X is *non-deterministically one-step provable* from M by means of P, if $X \subseteq hset(P(M))$ and $X \models hd(r)$ for every rule $r \in P(M)$. The *nondeterministic one-step provability operator*

$$T_P^{nd} : \mathcal{P}(At) \rightarrow \mathcal{P}(\mathcal{P}(At))$$

for a program P is an operator where $T_P^{nd}(M)$ consists of all sets that are non-deterministically one-step provable from M by means of P, for every $M \subseteq At$. In other words,

$$T_P^{nd}(M) = \{X \ : \ X \subseteq hset(P(M)),\ \forall r \in P(M).\ (X \models hd(r))\ \}.$$

Observe that, for every $X \in T_P^{nd}(M)$, X is a model of $P(M)$.

For an arbitrary atom $a \in At$, the constraints $(\{a\}, \{\{a\}\})$ and $(\{a\}, \{\emptyset\})$ are said to be *elementary*. Since $(\{a\}, \{\{a\}\})$ has the same models as a, we identify and denote the constraint $(\{a\}, \{\{a\}\})$ simply with a. For analogous reasons, we identify the constraint $(\{a\}, \{\emptyset\})$ with the literal $\mathbf{not}\ a$.

Given a normal logic program P and a rule $r \in P$, we denote with $C(r)$ the rule obtained by replacing every positive atom a in r with the constraint $(\{a\}, \{\{a\}\})$, and replacing every literal $\mathbf{not}\ a$ in r with the constraint $(\{a\}, \{\emptyset\})$. Let $C(P) = \{C(r) \mid r \in P\}$. We call $C(r)$ and $C(P)$ the *constraint* representation of r and P, respectively. It is easy to see that $C(P)$ is a convex program. It is possible to show that supported models of P coincide with supported models of $C(P)$, and answer sets of P coincide with answer sets of $C(P)$—according to the definition of answer sets presented by Liu and Truszczyński (2005). In other words, programs with constraints are sufficient to express normal logic programs. We conclude this section with an example illustrating the different concepts related to constraints.

Example 5 *Consider the program P*

$$\begin{aligned}
r_1 :&\quad (\{a,b,c\}, \{\{a,b\}, \{a,b,c\}, \{c\}\}) \leftarrow (\{a,b\}, \{\emptyset, \{a,b\}\}) \\
r_1 :&\quad (\{a,b,c\}, \{\{b\}, \{c\}, \{b,c\}\}) \leftarrow (\{a,b\}, \{\emptyset, \{a,b\}\}) \\
r_2 :&\quad a \leftarrow b \\
r_3 :&\quad b \leftarrow a \\
r_4 :&\quad c \leftarrow
\end{aligned}$$

We have that:

- *all constraints occurring in P are consistent;*
- $(\{a\}, \{\{a\}\})$ *(written as a in rules r_2 and r_3) is a monotone constraint;*
- $(\{a,b,c\}, \{\{b\}, \{c\}, \{b,c\}\})$ *is not a monotone constraint; however, it is a convex constraint;*
- $(\{a,b\}, \{\emptyset, \{a,b\}\})$ *is neither monotone nor convex;*

- *$hset(r_1) = hset(P) = \{a, b, c\}$ and $hset(r_2) = \{a\}$;*
- *the set $M = \{a\}$ satisfies the head of r_1 but does not satisfy the body of r_1;*
- *for $M = \{a\}$, the set of M-applicable rules is $\{r_3, r_4\}$, i.e., $P(\{a\}) = \{r_3, r_4\}$;*
- *the set $\{a\}$ is not a model of the program since it does not satisfy r_3;*
- *the set $\{c\}$ is a model of the program and so is $\{a, b, c\}$, both are supported;*
- *for $M = \emptyset$, $P(M) = \{r_1, r_4\}$, which implies that $T_P^{nd}(M) = \{\{c\}, \{b, c\}\}$.* □

5 Computations for Programs with Constraints

In this section we extend the notion of a computation to programs with constraints, and use computations to define a generalization of the answer-set semantics for such programs. Our approach is based on exploiting the intuitions that we have developed in Section 3 for the case of normal logic programs.

In order to define computations for programs with constraints, we consider the principles identified in Section 3. The key step is to generalize the revision principle. For normal programs, this principle was based on sets of atoms grounded in a set of atoms X (i.e., the current interpretation) and P. We will now extend this concept to programs with constraints.

Definition 3 *Let P be a program with constraints and let $X \subseteq At$ be a set of atoms. A set Y is* grounded *in X and P if there exists a set of rules $Q \subseteq P(X)$ such that $Y \in T_Q^{nd}(X)$. We denote by $Concl_P(X)$ the collection of all sets Y grounded in X and P.*

The intuition is analogous to the one used in the case of normal logic programs. There, a set Y is grounded in X and a normal logic program P if Y can be justified by means of *some* X-applicable rules in P. That is, $Y \in Concl_P(X)$ if and only if $Y = T_Q(X)$ for some $Q \subseteq P(X)$. Thus, the definition of $Concl_P(X)$ for a constraint program P indeed generalizes the earlier definition.

With this definition of $Concl_P(X)$, the principle $(\mathbf{R})$ lifts without any changes to program with constraints—we will refer to the version of principle $(\mathbf{R})$ in the case of program with constraints as $(\mathbf{R}')$. The same is true for the principle $(\mathbf{P})$, now referred to as $(\mathbf{P}')$. An appropriate generalization of the principle $(\mathbf{C})$ can be expressed in terms of supported models as follows:

$(\mathbf{C}')$ *Convergence*: X_∞ is a supported model of P, that is, $X_\infty \in T_P^{nd}(X_\infty)$.

Finally, the principle $(\mathbf{Pr})$ can be generalized, as well. At a step i of a computation that satisfies $(\mathbf{R}')$, we select as X_i an element of $Concl_P(X_{i-1})$. From the definition of $Concl_P(X_{i-1})$, there is a program $P_{i-1} \subseteq P(X_{i-1})$ such that $X_i \in T_{P_{i-1}}^{nd}(X_{i-1})$. Each such program can be viewed as a *reason* for X_i. We can now state the generalized principle $(\mathbf{Pr}')$ as follows:

$(\mathbf{Pr}')$ *Persistence of Reasons*: There is a sequence of programs $\langle P_i \rangle_{i=0}^{\infty}$ such that for every $i \geq 0$, $P_i \subseteq P_{i+1}$, $P_i \subseteq P(X_i)$, and $X_{i+1} \in T_{P_i}^{nd}(X_i)$.

The definition implies that for every $j \geq i$, $P_i \subseteq P_j$ and $P_i \subseteq P(X_j)$. That is, the principle requires that rules used at step i "persist" ($P_i \subseteq P_j$, for $j \geq i$) and are applicable at all successive steps ($P_i \subseteq P(X_j)$, for $j \geq i$).
Having generalized the principles $(\mathbf{R})$, $(\mathbf{P})$, $(\mathbf{C})$ and $(\mathbf{Pr})$ to define $(\mathbf{R}')$, $(\mathbf{P}')$, $(\mathbf{C}')$ and $(\mathbf{Pr}')$ for the class of programs with constraints, we can now extend the concept of a computation by literally lifting the definitions developed for the case of normal logic programs. However, the resulting notion has some undesired properties, as shown in the next example.

Example 6 *Let P_5 be the program:*

$$(\{a, b\}, \{\{a, b\}\}) \leftarrow (\{a, b\}, \{\emptyset, \{a, b\}\})$$

The constraint in the body is satisfied by interpretations where a and b are either both true or both false. The head of the rule is satisfied only in the case both a and b are true.
It is easy to see that the sequence $\emptyset, \{a, b\}, \{a, b\}, \ldots$ satisfies the properties $(\mathbf{R}')$, $(\mathbf{P}')$, $(\mathbf{C}')$ and $(\mathbf{Pr}')$. On the other hand, this outcome is not satisfactory: a and b are in the result of this computation only because they "self-support" themselves. Given that $\{a, b\}$ is the result of the sequence, the only set ensuring that the rule $(\{a, b\}, \{\{a, b\}\}) \leftarrow (\{a, b\}, \{\emptyset, \{a, b\}\})$ remains applicable independently of what may happen later in the computation is $\{a, b\}$ itself. Indeed, $\emptyset$ (the only other satisfier of the body of the rule) is too weak due to the non-monotonicity of the constraint. The empty set might potentially be revised into $\{a\}$ or $\{b\}$ (not possible here, but a priori *possible if other rules were present in the program). For each of these sets the rule would not be applicable and persistence of reasons would be violated.*
To better present this point let us contrast this situation with the case of a logic program P_6 consisting of the following rules (in a syntax resembling that of SMODELS*):*

$$a \leftarrow$$

$$b \leftarrow 1\{a, b, c\}2$$

The body of the second rule is satisfied by any interpretation that contains at least one and at most two of a, b, and c. The sequence $\emptyset, \{a\}, \{a, b\}, \{a, b\}, \ldots$ satisfies the properties $(\mathbf{R}')$, $(\mathbf{P}')$, $(\mathbf{C}')$ and $(\mathbf{Pr}')$. Also in this case, it may look as if b is self-supported. But, in fact it is not. Given that the result of the computation is $\{a, b\}$, once we establish a in the computation, the second rule will remain applicable no matter how the current state is revised (as long as the revision does not go beyond $\{a, b\}$). In other words, in view of the second rule, for b to hold it suffices to know a, and b really depends on a and not on itself.
The key behind this is the convexity of the constraint $1\{a, b, c\}2$, which ensures that there are no "gaps" between the derivation of the atom a, which is a trigger *for the rule, and the result of the computation ($\{a, b\}$). Every set in between is guaranteed to "activate" the rule by satisfying its body. This is exactly the property that was missing in the case of program P_5.* □

Our example suggests that, in the case of programs with arbitrary constraints, the four properties we introduced do not capture all that is needed for a computation to give rise to a reasonable notion of an answer set. We also need to require that the result of the computation is "well-founded", that is, that every element derived in the process remains founded in elements derived earlier *throughout* the rest of the computation.

In order to formalize this concept, we need to introduce some additional definitions. Let A be a constraint and X and Y a set of atoms. The set Y is an *X-trigger* for A if, for every set Z such that $Y \subseteq Z \subseteq X$, we have that $Z \models A$. If r is a rule, Y is an *X-trigger* for r if it is an X-trigger for every constraint A in the body of R; i.e., for every set Z, such that $Y \subseteq Z \subseteq X$, we have that $Z \models body(r)$. Thus, if Y is an X-trigger for r, then as long as a computation does not go beyond X, Y is a sufficient justification for applying r, and atoms derived based on r can be regarded as founded only in Y. This idea is captured by the following property:

($\mathbf{FPr}$) *Founded persistence of reasons*: There is a sequence of programs $\langle P_i \rangle_{i=0}^{\infty}$ such that for every $i \geq 0$, $P_i \subseteq P_{i+1}$, $P_i \subseteq P(X_i)$, $X_{i+1} \in T_{P_i}^{nd}(X_i)$, and each rule in P_i has an X_∞-trigger contained in X_i.

Thus, the principle of founded persistence of reasons simply strengthens that of persistence of reasons by the foundedness requirement.

We now use the principles introduced above to define several types of computations for programs with arbitrary constraints.

Definition 4 *Let P be a program with constraints. A sequence of interpretations $\langle X_i \rangle_{i=0}^{\infty}$ is a* computation *for P if $X_0 = \emptyset$ and the sequence satisfies the principles ($\mathbf{R}'$), ($\mathbf{P}'$) and ($\mathbf{C}'$). A computation is* persistent *if it satisfies the principle ($\mathbf{Pr}'$). A computation is* founded *if it satisfies the principle ($\mathbf{FPr}$).*

Persistent computations are computations. As in the case of normal programs, the converse is not true in general. Let us consider the program $C(P_2)$, where P_2 is the normal logic program from Example 2, The sequence $\emptyset, \{a\}, \{a\}, \ldots$ is a computation but not a persistent one. It is not a coincidence that we could derive a counterexample from a normal logic program. We have the following general result.

Proposition 7 *Let P be a normal logic program. The class of computations for P, as defined in Section 3, coincides with the class of computations of $C(P)$, as defined in this section. Similarly, the class of persistent computations for P, as defined in Section 3, coincides with the class of persistent computations for $C(P)$ as defined in this section.*

Proof It is straightforward to verify that the sequence $\langle X_i \rangle_{i=0}^{\infty}$ satisfies properties ($\mathbf{R}'$), ($\mathbf{P}'$) and ($\mathbf{C}'$) with respect to P (as defined in Section 3) if and only if $\langle X_i \rangle_{i=0}^{\infty}$ satisfies properties ($\mathbf{R}'$), ($\mathbf{P}'$) and ($\mathbf{C}'$) with respect to $C(P)$. Thus, the notions of computation in P and in $C(P)$ coincide.

Let us show that the same result holds for the case of persistent computations. Let us assume that $\langle X_i \rangle_{i=0}^{\infty}$ satisfies the property ($\mathbf{Pr}'$) for P (as defined in Section 3). For each atom $a \in X_\infty$, let r_a denote the rule satisfying the condition of persistence in ($\mathbf{Pr}'$). Let $P_i = \{C(r_a) \mid a \in X_{i+1}\}$. From the persistence of reasons for P,

$P_i \subseteq P_{i+1}$, $P_i \subseteq P(X_i)$, and $\{X_{i+1}\} = T^{nd}_{P_i}(X_i)$. Hence, $\langle P_i \rangle^{\infty}_{i=0}$ is a sequence of subprograms of $C(P)$ satisfying the conditions of $(\mathbf{Pr}')$ for $C(P)$, as defined in this section.
Conversely, let us assume that $\langle X_i \rangle^{\infty}_{i=0}$ satisfies the property $(\mathbf{Pr}')$ for $C(P)$ (as defined in this section) and let $\langle P_i \rangle^{\infty}_{i=0}$ be the sequence of subprograms of $C(P)$ that demonstrates that.
For each $a \in X_{\infty}$, let i be the index such that $a \in X_{i+1} \setminus X_i$. Because $X_{i+1} \in T^{nd}_{P_i}(X_i)$, we can conclude that there exists some rule $r_a \in P$ such that $C(r_a) \in P_i$ and $hset(r_a) = \{a\}$. Since $P_i \subseteq C(P)(X_i)$, we have $X_i \models body(r_a)$ and $hd(r_a) = a$. The monotonicity of $\langle P_i \rangle^{\infty}_{i=0}$ implies that $C(r_a) \in P_j$ for $j \geq i$. The condition $P_j \subseteq C(P)(X_j)$ implies that $C(r_a)$ is applicable in every X_j for $j \geq i$. Thus, r_a is applicable in every X_j for $j \geq i$. Hence, $\langle X_i \rangle^{\infty}_{i=0}$ is a persistent computation for P, according to the definition in Section 3. □

For normal logic programs, we do not need to impose the principle of founded persistence explicitly. It is possible to show that, in the case of normal logic programs, persistent computations are always founded. [3] Thus, for normal logic programs, the two classes of computations coincide. In fact, this property holds for a larger classes of programs—the class of programs with convex constraints, which includes normal logic programs or, more precisely, programs of the form $C(P)$, where P is a normal program.

Proposition 8 *Let P be a program with convex constraints. A computation for P is founded if and only if it is persistent.*

Proof From the previous observations, we know that each founded computation is persistent. Thus, we need to show that every persistent computation for P is founded. Let $\langle X_i \rangle^{\infty}_{i=0}$ be a persistent computation for P and let $\langle P_i \rangle^{\infty}_{i=0}$ be a sequence of programs demonstrating that $\langle X_i \rangle^{\infty}_{i=0}$ satisfies the property $(\mathbf{Pr}')$.
Let $r \in P_i$ (for some $i \geq 0$). It follows that $r \in P(X_i)$, that is $X_i \models body(r)$. From the persistence of reasons, $r \in P(X_j)$, for every $j \geq i$, and, because of the finiteness of the domains of constraints, $r \in P(X_{\infty})$. Thus, $X_{\infty} \models body(r)$. Since all constraints in the body of r are convex, for every Z such that $X_i \subseteq Z \subseteq X_{\infty}$, $Z \models body(r)$. It follows that X_i is an X_{∞}-trigger for r. Consequently, the computation $\langle X_i \rangle^{\infty}_{i=0}$ is founded. □

However, in the general case of programs with arbitrary constraints, the class of persistent computations is a *proper* subclass of the class of founded computations. The inclusion follows directly from the definition. The program P_5 from Example 6 shows that it is proper. For programs with arbitrary constraints, we use founded computations as the basis for the generalization of the answer-set semantics.

Definition 5 *Let P be a program with constraints. A set X is an* answer set *of P if there is a founded computation for P whose result is X.*

Propositions 7 and 8 imply that our concept of an answer set, as specified by Definition 5, generalizes that for normal logic programs.

[3] We assume a natural definition of founded persistence for normal logic programs based on the correspondence between P and $C(P)$.

Corollary 1 *Let P be a normal logic program. A set $X \subseteq At$ is an answer set of P if and only if X is an answer set of $C(P)$.*
Proof We have that X is an answer set of P if and only if there is a persistent computation for P whose result is X. From Proposition 7, it is the case if and only if there is a founded computation for $C(P)$ whose result is X, that is, if and only if X is an answer set of $C(P)$. □

We can also show that this definition of answer set generalizes the notion of answer set for programs with convex constraints (proposed by Liu and Truszczyński (2005)), a property that we formally state and prove later in the paper.
We conclude this section by observing that we have determined so far four distinct classes of models of programs with constraints: answer sets, models obtained via persistent computations, models obtained as results of computations, and supported models. Each class of models is a subclass (in general, proper) of the successive one. The role of the last three classes of models, and their properties require further studies, that are beyond the scope of this paper.
We also note that, thanks to the principle $(\mathbf{C}')$, supported models form indeed a superclass of the class of the results of computations. However, not all supported models of programs can be obtained as the results of computations. For instance, $\{a\}$ is a supported model of the program $C(P)$, where $P = \{a \leftarrow a\}$, but there is no computation for $C(P)$ with the result $\{a\}$. To capture supported models as the results of some bottom-up process, the notion of a computation has to be broadened by relaxing some of the key principles. In the following section, we show one way in which that goal can be accomplished.

6 Computations and Quasi-Satisfiability Relations

The notion of a computation discussed so far makes use of the non-deterministic operator $Concl_P$ to revise the interpretations occurring in a computation. As we mentioned earlier, the use of $Concl_P$ provides a wide range of choices for revising a state of a computation, considering all the subsets of applicable rules.
In this section, we will study sequences of interpretations that can be generated by narrowing down the set of choices allowed in $Concl_P(X)$ as possible revisions of X. In the case of normal logic programs, we accomplished this goal by means of an operator T_P^M, based on the satisfiability relation $\models_M$. In that case, the whole computation is determined just by the choice of M. Thus, the only non-deterministic decision is the selection of M. Once that is done, there is no non-determinism left. We proved that M is an answer set of a logic program P if and only if M is the result of the computation it generates. In other words, the computation is context-dependent. This idea has been studied in the context of default logic by Marek and Truszczyński (1993). We will now generalize that approach to the case of programs with constraints.
Definition 6 *A sequence $\langle X_i \rangle_{i=0}^{\infty}$ is a* weak computation *for a program with constraints P if it satisfies the properties $(\mathbf{P}')$ and $(\mathbf{C}')$ and $X_0 = \emptyset$.*
Thus, weak computations are sequences that do not rely on a program P when mov-

ing from step i to step $i+1$. Next, we will define a broad class of weak computations that, at least to some degree, restores the role of P as a revision mechanism.
Let $\triangleright$ be a relation between sets of atoms (interpretations) and abstract constraints. We extend the relation $\triangleright$ to the case of conjunctions (or sets) of constraints as follows: $X \triangleright \{A_1, \ldots, A_k\}$ if $X \triangleright A_i$, for every i, $1 \leq i \leq k$. This relation is intended to represent some concept of satisfiability of constraints and their conjunctions. We will call such relations *quasi-satisfiability* relations. They will later allow us to generalize the relation $\models_M$.
For a quasi-satisfiability relation $\triangleright$, we define

$$P^{\triangleright}(X) = \{r \in P \mid X \triangleright body(r)\}.$$

In other words, $P^{\triangleright}(X)$ is the set of all rules in P that are applicable with respect to X under the satisfiability relation $\triangleright$. Next, we define

$$T_P^{nd;\triangleright}(X) = \{Y \,:\, Y \subseteq hset(P^{\triangleright}(X)),\, \forall r \in P^{\triangleright}(X).\, (Y \models hd(r))\,\}.$$

In other words, $T_P^{nd;\triangleright}(X)$ consist of all sets $Y \subseteq hset(P^{\triangleright}(X))$ such that $Y \models hd(r)$, for every $r \in P^{\triangleright}(X)$. Thus, $T_P^{nd;\triangleright}$ works similarly to T_P^{nd}, except that rules in $P^{\triangleright}(X)$ are "fired" rather than those in $P(X)$.
Definition 7 *Let $\triangleright$ be a quasi-satisfiability relation. A weak computation $\langle X_i\rangle_{i=0}^{\infty}$ is a* weak $\triangleright$-computation *for P if $X_{i+1} \in T_P^{nd;\triangleright}(X_i)$, for $i \geq 1$.*
Since we do not impose any particular properties on the quasi-satisfiability relation $\triangleright$, it is not guaranteed that $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$. Thus, weak $\triangleright$-computations are not guaranteed to be computations. There is, however, a simple sufficient conditions guaranteeing that weak $\triangleright$-computations are computations.
We say that a quasi-satisfiability relation $\triangleright$ is a *sub-satisfiability* relation if, for every $X \subseteq At$ and every abstract constraint A, $X \triangleright A$ implies $X \models A$. Observe that if $\triangleright$ is a sub-satisfiability relation then $P^{\triangleright}(X) \subseteq P(X)$. This property and the appropriate definitions imply that

$$T_P^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd}(X). \tag{3}$$

We can observe that the relation $\models_M$ considered in Section 3 is a sub-satisfiability relation.
Proposition 9 *Let P be a program with constraints. If $\triangleright$ is a sub-satisfiability relation then for every $X \subseteq At$, we have that $T_P^{nd;\triangleright}(X) \subseteq Concl_P(X)$.*
Proof Consider $Y \in T_P^{nd;\triangleright}(X)$. Since $\triangleright$ is a sub-satisfiability relation, $P^{\triangleright}(X) \subseteq P(X)$. Moreover, from Equation (3), $T_P^{nd;\triangleright}(X) = T_{P^{\triangleright}(X)}^{nd}(X)$. Thus, it follows that $Y \in Concl_P(X)$. □

The following corollary is an obvious consequence of Proposition 9.
Corollary 2 *If $\triangleright$ is a sub-satisfiability relation, then every weak $\triangleright$-computation is a computation.*
Thus, from now on, if $\triangleright$ is a sub-satisfiability relation, we will write $\triangleright$-computation interchangeably with *weak* $\triangleright$-computation.

We recall that if P is a normal logic program, then M is an answer set of P if and only if M is the result of an M-computation for P. We will now use weak $\triangleright$-computations and $\triangleright$-computations to generalize the notion of an M-computation to programs with constraints. To this end, we extend the approach proposed and studied by Son et al. (2007). Our method requires a mapping f that assigns to a set X of atoms a quasi-satisfiability relation $\triangleright_X^f$. Thus, we explore the possibility that the quasi-satisfiability relation can change as a function of the target model X we are trying to achieve. Later in this section, we will explore one particular mapping f; alternative mappings of interest are explored in a later section.

Definition 8 *Let P be a program with constraints and f a mapping assigning to every set X of atoms a quasi-satisfiability relation $\triangleright_X^f$. If C is a weak $\triangleright_X^f$-computation for P and X is the result of C, then C is a self-justified weak $\triangleright_X^f$-computation for P. A set of atoms X is an* f-model of P *if X is the result of a self-justified weak $\triangleright_X^f$-computation for P.*

The definition of an f-model is sound. Since weak computations satisfy the property $(\mathbf{C}')$, their results are indeed models of P. In fact, they are supported models. Several interesting classes of models of programs with constraints can be described in terms of f-models by specializing the mapping f. We will demonstrate the flexibility of the approach presented above later in the paper.

In order to specialize the general approach of self-justified weak computations so that it might generalize the notion of an answer set, we need to identify mappings f that ensure that self-justified weak computations are computations—i.e., they satisfy the revision principle—and are founded. We have already seen that if $\triangleright_X^f$ is a sub-satisfiability relation then weak $\triangleright_X^f$-computations are computations (referred to as $\triangleright_X^f$-computations). We will now seek conditions guaranteeing the foundedness of $\triangleright_X^f$-computations.

We first address the weaker property of persistence of reason $(\mathbf{Pr}')$. The next proposition follows from Equation 3.

Proposition 10 *Let $\triangleright$ be a sub-satisfiability relation and let $C = \langle X_i \rangle_{i=0}^{\infty}$ be a $\triangleright$-computation. If for every constraint A and every $i = 0, 1, \ldots$, $X_i \triangleright A$ implies that $X_{i+1} \triangleright A$, then C is persistent.*

Proof Since $\triangleright$ is a sub-satisfiability relation, $P^{\triangleright}(X) \subseteq P(X)$. It follows from Equation 3 and Definition 7 that the sequence $P_i = P^{\triangleright}(X_i)$ satisfies the persistence condition. □

We will now show that the sub-satisfiability relation proposed by Son et al. (2007) gives rise to founded computations. Given a set X of atoms we define $\triangleright_X^{spt}$ as follows: $Y \triangleright_X^{spt} A$ if for each set Z such that $Y \subseteq Z \subseteq X$, $Z \models A$ (or equivalently, $Z \cap A_{dom} \in A_{sat}$). It is easy to see that $\triangleright_X^{spt}$ is a sub-satisfiability relation. Thus, it defines computations. Secondly, if C is a $\triangleright_X^{spt}$-computation, then it is persistent as the relation $\triangleright_X^{spt}$ satisfies the assumptions of Proposition 10. Using the terminology introduced earlier, we can restate the definition of $\triangleright_X^{spt}$ as follows: $Y \triangleright_X^{spt} A$ if Y is an X-trigger for A. The connection to the definition of founded persistence is evident and, not surprisingly, self-justified $\triangleright_X^{spt}$-computations are founded.

Proposition 11 *Let P be a program with constraints. If C is a self-justified $\triangleright_X^{spt}$-*

computation then it is a founded computation.

Proof To simplify notation, we write $\triangleright$ for $\triangleright^{spt}_X$ in the proof. Let $C = \langle X_i \rangle_{i=0}^{\infty}$ be the self-justified $\triangleright$-computation. We already argued that C is a persistent computation by using the sequence $P_i = P^{\triangleright}(X_i)$, $i = 0, 1, \ldots$, to show that the property $(\mathbf{Pr}')$ holds. We will now show that every rule in P_i has an X-trigger contained in X_i. To this end, let $r \in P_i$. It follows that $X_i \triangleright \mathit{body}(r)$. By the definition of $\triangleright$ $(= \triangleright^{spt}_X)$, if $X_i \subseteq Z \subseteq X$, then $Z \triangleright \mathit{body}(r)$. Thus, X_i is an X-trigger for r. Consequently, C is founded. □

Not every founded computation C is a self-justified $\triangleright^{spt}_X$-computation. This can be seen in the following example.

Example 7 *Consider the program P (remember that a is shorthand for $(\{a\}, \{\{a\}\})$):*

$$a \leftarrow$$

$$b \leftarrow$$

$$c \leftarrow (\{a, b\}, \{\emptyset, \{a, b\}\})$$

We have that $\emptyset, \{a\}, \{a, b\}, \{a, b, c\}, \{a, b, c\} \ldots$ is a founded computation for P which is not a self-justified $\triangleright^{spt}_{\{a,b\}}$-computation because it does not end in the set $\{a, b\}$. □

Although not every founded computation is a self-justified $\triangleright^{spt}_X$-computation, for every answer set X there is a self-justified $\triangleright^{spt}_X$-computation. Thus, the results of self-justified $\triangleright^{spt}_X$-computations for P are precisely the answer sets of P. Formally, we have the following result:

Proposition 12 *Let P be a program with constraints and let X be an answer set for P. Then there is a self-justified $\triangleright^{spt}_X$-computation for P.*

Proof Also in this proof, to simplify the notation we write $\triangleright$ for $\triangleright^{spt}_X$. Let $\langle X_i \rangle_{i=0}^{\infty}$ be a founded computation with the result X and let $\langle P_i \rangle_{i=0}^{\infty}$ be a sequence of programs demonstrating that $(\mathbf{FPr})$ holds.

Let us define $Y_0 = \emptyset$ and $Y_{i+1} = \mathit{hset}(P^{\triangleright}(Y_i)) \cap X$, for $i = 0, 1, \ldots$. It follows that for every $i = 0, 1, \ldots$, $Y_i \subseteq X$. Furthermore, $X \models P$ and so, $X \models P^{\triangleright}(Y_i)$. By the definition of $\triangleright$ $(= \triangleright^{spt}_X)$, for every rule $r \in P^{\triangleright}(Y_i)$, $X \models \mathit{body}(r)$. Thus, $X \models \mathit{hd}(r)$. Since $\mathit{hset}(r) \subseteq \mathit{hset}(P^{\triangleright}(Y_i))$, $Y_{i+1} \models \mathit{hd}(r)$. It follows that $Y_{i+1} \in T^{nd}_{P^{\triangleright}(Y_i)}(Y_i)$. Since $T^{nd;\triangleright}_P(Y_i) = T^{nd}_{P^{\triangleright}(Y_i)}(Y_i)$, $Y_{i+1} \in T^{nd;\triangleright}_P(Y_i)$. Thus, $\langle Y_i \rangle_{i=0}^{\infty}$ is a $\triangleright$-computation for P.

Moreover, for every i, $X_i \subseteq Y_i$. We prove this claim by induction. For the base case, we note that $X_0 = \emptyset = Y_0$. Let us now assume that $X_i \subseteq Y_i$. We have $X_{i+1} \in T^{nd}_{P_i}(X_i)$. Since $X_i \subseteq Y_i$, foundedness of $\langle X_i \rangle_{i=0}^{\infty}$ implies that $P_i \subseteq P^{\triangleright}(Y_i)$. Indeed, if $r \in P_i$, then r has an X-trigger U_r such that $U_r \subseteq X_i$. It follows that for every Z such that $U_r \subseteq Z \subseteq X$, $Z \models \mathit{body}(r)$. Thus, for every Z such that $Y_i \subseteq Z \subseteq X$, $Z \models \mathit{body}(r)$ and so, $r \in P^{\triangleright}(Y_i)$.

Clearly, $P_i \subseteq P^{\triangleright}(Y_i)$ implies that $X_{i+1} \subseteq \mathit{hset}(P^{\triangleright}(Y_i) \cap X) = Y_{i+1}$. This completes the proof of the claim. The claim, in turn, implies that $\bigcup_{i=0}^{\infty} Y_i = X$. It follows that $\langle Y_i \rangle_{i=0}^{\infty}$ is a self-justified $\triangleright^{spt}_X$-computation. □

7 Discussion

In this section, we discuss several additional properties of the semantics of answer sets we introduced in the paper. In particular, we present two alternative characterizations of the semantics—one based on the concept of strong groundedness, which is based on the existence of a ranking of atoms, and the second based on program transformation. Finally, we will discuss some alternative proposals for semantics of programs with constraints (or aggregates).

7.1 Strong Groundedness of Answer Sets

One of the key properties of answer sets of normal logic programs is that they are not self-supported. Namely, we have the following property that can be derived from more general results (Theorem 1) proposed by Erdem and Lifschitz (2003).

Theorem 1 *Let P be a normal logic program and M a model of P. Then M is an answer set of P if and only if there is a ranking k assigning non-negative integers to atoms in M, so that for every atom $a \in M$, there is a rule $r \in P(M)$ such that $hd(r) = a$ and for every $b \in pos(r)$, we have that $k(a) > k(b)$.*

We will now extend this property to the case of programs with arbitrary constraints and the concept of an answer set we introduced here. The notion of an M-trigger and the property of founded persistence of reasons are critical. We recall that a set X is an M-trigger for a rule r precisely when for every Y, $X \subseteq Y \subseteq M$, $Y \models body(r)$. In other words, if X is an M-trigger for r, having computed X guarantees that the rule will remain applicable later in the computation as long as the computation "stays" within M. In such case, we say that X is an *M-justification* for every atom $a \in hset(r) \cap M$. We note that a rule can have several M-triggers and each of them can be used as an M-justification for the elements in the headset of the rule.

We say that a model M of a program with arbitrary constraints is *strongly grounded* if there is a ranking of atoms in M such that each atom has an M-justification consisting entirely of atoms with strictly lower ranks. We will show that answer sets are precisely those models that are strongly grounded. We start with an example.

Example 8 *Let us consider the following program P_6:*

$$a \leftarrow (\{a, b\}, \{\emptyset, \{a, b\}\})$$
$$a \leftarrow b$$
$$b \leftarrow a$$

Clearly, $M = \{a, b\}$ is a model of P_6. It is easy to see that M is not strongly grounded. Indeed, the only M-justification for the atom a provided by the first rule is $\{a, b\}$—the other satisfier, $\emptyset$, is not an M-trigger for that rule as some of its supersets (e.g., $\{a\}$) do not satisfy the body of the rule. The only M-justification for a provided by the second rule is $\{b\}$ and the only M-justification for b (it is provided by the third rule) is a. Thus, it is clear that no ranking necessary for

strong groundedness exists. One can also verify that P_6 has no answer sets. □

The next theorem relates the notions of answer sets and strongly groundedness.

Theorem 2 *A model M of a program with arbitrary constraints is an answer set of P if and only if it is strongly grounded.*

Proof First, let us assume that M is an answer set of P. It follows that there is a founded computation $\langle X_i\rangle_{i=0}^{\infty}$ such that $X_\infty = M$. Let $a \in M$. We define $k(a)$ as the minimum i such that $a \in X_i$. Since $M = \bigcup_{i=0}^{\infty} X_i$, the value $k(a)$ is well defined for every atom $a \in M$.

We will show that the ranking k is a witness of M being strongly grounded. To this end, we need to show that every atom in M has an M-justification. Let $a \in M$. Since $\langle X_i\rangle_{i=0}^{\infty}$ is founded, there is a rule r and a set of atoms Y such that $hd(r) = a$, $Y \subseteq X_{i-1}$ and Y is an M-trigger for r. We observe that for every $b \in Y$, $k(b) \leq i-1$. Thus, Y is an M-justification for a.

Conversely, let us assume that M is a strongly grounded model of P. We will show that M is an answer set by constructing a founded computation $\langle X_i\rangle_{i=0}^{\infty}$ such that $M = X_\infty$. Let k be a ranking that witnesses strong groundedness of M. We define $X_0 = \emptyset$. We then proceed inductively as follows. We define P_i, $i \geq 0$, to consist of all rules for which X_i is an M-trigger. We define X_{i+1}, $i \geq 0$, by setting $X_{i+1} = hset(P_i) \cap M$.

Directly from the definition, it follows that for every $i \geq 0$, $X_i \subseteq M$ and $P_i = P_i(X_i) = P_i(M)$. Since M is a model of P, we have that for every $r \in P_i$, $hset(P_i) \cap M \models hd(r)$. Thus, $X_{i+1} \in T_{P_i}^{nd}(X_i)$ (and $X_{i+1} \in Concl_P(X_i)$) and so, $\langle X_i\rangle_{i=0}^{\infty}$ satisfies the principle $(\mathbf{R}')$.

Next, we will prove that $\langle X_i\rangle_{i=0}^{\infty}$ satisfies the principle $(\mathbf{P}')$, that is, that for every $i \geq 0$, $X_i \subseteq X_{i+1}$. We proceed by induction. The inclusion $X_0 \subseteq X_1$ is evident. By the induction hypothesis, $X_{i-1} \subseteq X_i$. Since $X_i \subseteq M$, X_i is an M-trigger for every rule in P_{i-1}. Thus, $P_{i-1} \subseteq P_i$. It follows that

$$X_i = hset(P_{i-1}) \cap M \subseteq hset(P_i) \cap M = X_{i+1}.$$

That completes the proof of the inductive step and of the claim (for property $(\mathbf{P}')$). The principle $(\mathbf{P}')$ implies that for every $i \geq 0$, $P_i \subseteq P_{i+1}$. Together with earlier observations, that implies that $\langle X_i\rangle_{i=0}^{\infty}$ satisfies the principle $(\mathbf{FPr})$.

By definition, $X_\infty \subseteq M$. We will now prove the converse inclusion. To this end, let us consider $a \in M$. We will prove by induction on the rank k of a that $a \in X_{k+1}$.

If $a \in M$ has rank 0, then a has an empty M-justification, that is, there is a rule r such that $a \in hset(r)$ and $\emptyset$ is an M-trigger for r. It follows that $r \in P_0$ and so, $a \in hset(P_0) \cap M = X_1$. The reasoning for the inductive step is essentially the same. Let us consider $a \in M$ such that $k(a) = k \geq 1$. By definition, there is a rule r such that Y is an M-trigger for r and Y consists of atoms of ranks strictly lower than a. By the induction hypothesis, $Y \subseteq X_k$. Thus, $r \in P_k$ and so, $a \in X_{k+1}$.

It follows that $M \subseteq X_\infty$. Consequently, $M = X_\infty$. To complete the proof all that remains is to show is that X_∞ is a supported model of P, that is, that $\langle X_i\rangle_{i=0}^{\infty}$ satisfies the principle $(\mathbf{C}')$. To this end, we recall that for every $i \geq 0$, $P_i = P_i(M)$. Thus, $P_i \subseteq P(M)$ and, since $X_i \subseteq hset(P_i)$, $M = X_\infty \subseteq hset(P(M))$. Since M

is a model of P, it follows that X_∞ is a supported model of P. □

7.2 Answer-Set Semantics and Program Transformations

Program transformations are mappings assigning programs to programs. They form a useful tool in the studies of semantics of programs. Invariance of a semantics to a particular program transformation yields methods for program rewriting and simplification, as well as normal form representations of programs. Alternatively, program transformations can be used to generalize a semantics defined for programs of some simple syntactic form to those without syntactic restrictions.
We will consider here a simple transformation of programs with arbitrary constraints to programs with convex constraints. We will show that the transformation preserves the semantics of answer sets. Given that all major proposals for the semantics of programs with convex constraints coincide, that result provides additional support for the notion of an answer set as we defined it here.
First, we show that every program with arbitrary constraints can be transformed into a head-convex program. A rule r is *head-convex* if $hd(r)$ is a convex constraint. A program is *head-convex* if all its rules are head-convex.
Let $r = A \leftarrow A_1, \ldots, A_k$ be a rule with arbitrary constraints. We represent it by two head-convex rules $cstr(r)$ and $sppt(r)$. The first rule is

$$cstr(r) = \bot \leftarrow \overline{A}, A_1, \ldots, A_k$$

where $\overline{A}$ denotes the complement of A, that is, the constraint B such that $B_{dom} = A_{dom}$ and $B_{sat} = \{X \subseteq A_{dom} \mid X \notin A_{sat}\}$. The second rule is

$$sppt(r) = (A_{dom}, \{X \mid X \subseteq A_{dom}\}) \leftarrow A_1, \ldots, A_k.$$

Clearly, both $\bot$ and $(A_{dom}, \{X \mid X \subseteq A_{dom}\})$ are convex. Thus, rules $cstr(r)$ and $sppt(r)$ are head-convex. The role of $cstr(r)$ is to capture precisely the same constraint as the one expressed by r. The role of $sppt(r)$ is to "support" precisely the same atoms as r and in exactly the same circumstances. For a program with arbitrary constraints, P, we define

$$hc(P) = \{cstr(r) \mid r \in P\} \cup \{sppt(r) \mid r \in P\}.$$

Theorem 3 *Let P be a program with arbitrary constraints and $M \subseteq At$. Then M is an answer set of P if and only if M is an answer set of $hc(P)$.*
Proof It is evident that M is a model of P if and only if M is a model of $hc(P)$. Moreover, directly from the definition it follows that M is strongly grounded with respect to P if and only if M is strongly grounded with respect to $hc(P)$. Thus, the result follows from Theorem 2. □

Next, we will show that every head-convex program can be transformed into a program with convex constraints so that the answer sets are preserved. Let A, B be a constraints. We say that A is a *subconstraint* of B if $A_{dom} = B_{dom}$ and $A_{sat} \subseteq B_{sat}$.

For a constraint A, we denote by $mc(A)$ the set of *maximal convex* subconstraints of A. We note that every satisfier X of A is a satisfier of at least one of the constraints in $mc(A)$.
Given a head-convex rule $A \leftarrow A_1, \ldots, A_k$, we define $Cvx(r)$ to be the set of all rules of the form $A \leftarrow A'_1, \ldots, A'_k$, where $A'_i \in mc(A_i)$. For a head-convex program P, we set

$$Cvx(P) = \bigcup_{r \in P} Cvx(r).$$

It is clear that $Cvx(P)$ is a program with convex constraints.
Theorem 4 *Let P be a head-convex program with arbitrary constraints and $M \subseteq At$. Then M is an answer set of P if and only if M is an answer set of $Cvx(P)$.*
Proof (Sketch) As before, we first note that M is a model of P if and only if M is a model of $Cvx(P)$. Next, we note that X is an M-trigger of a rule $r \in P$ if and only if there is a rule $r' \in Cvx(r)$ such that X is an M-trigger for r'. It follows that M is strongly grounded in P if and only if M is strongly grounded in $Cvx(P)$. Thus, as before, the result follows from Theorem 2. □

It follows that the transformation of P to $Cvx(hc(P))$ preserves answer sets. We note that the role of the transformation is conceptual rather than practical. The program $Cvx(hc(P))$ may grow exponentially relative to P (the size explosion potentially occurs in the second step of the transformation).

7.3 More About Self-justified Weak $\rhd_c^f$-Computations

Several interesting classes of models can be defined by specifying the mapping from sets to quasi-satisfiability relations, which determines a class of self-justified weak computations and so, a class of models. Each class of models that can be defined in this way consists of supported models—since we require that weak computations satisfy the convergence principle. We will now show that, in particular, the class of supported models can be defined in these terms.
Definition 9 *Let X be a set of atoms. For a set Y of atoms and a constraint A we define the relation $\rhd_X^{supp}$ as follows:*

$$Y \rhd_X^{supp} A \;\text{ if }\; X \models A.$$

We denote the mapping $X \mapsto \rhd_X^{supp}$ by $supp$.
Let M be a supported model of P. We have that the sequence $C = \langle \emptyset, M, M, \ldots \rangle$ satisfies $(\mathbf{P}')$, $(\mathbf{C}')$, and hence, is a weak computation. Furthermore, since M is a supported model of P, $P^{supp}(M) = P(M)$. This implies that $M \in T_P^{nd;supp}(M)$. Thus, C is a weak $\rhd_M^{supp}$-computation. As M is the result of C, C is a self-justified weak $\rhd_M^{supp}$-computation. Thus, supported models of P are $supp$-models of P. As we observed earlier, all f-models are supported models. It follows that supported models of P are precisely $supp$-models of P.

Next, we will discuss the semantics of programs with abstract constraints proposed by Marek and Remmel (2004). That proposal is based on a specialized notion of

program reduct. Let A be a constraint. By $\widehat{A}$ we denote the *closure* of A, that is, the constraint $(A_{dom}, \{Y \mid Y \subseteq A_{dom}, (\exists Z \in A_{sat})(Z \subseteq Y)\})$. Now, given a program P and an interpretation M, the *NSS-reduct* of P with respect to M is obtained by *(i)* removing all rules whose body is not satisfied by M, and *(ii)* replacing each remaining rule $A \leftarrow A_1, \ldots, A_k$ with the set of rules $\{a \leftarrow \widehat{A_1}, \ldots, \widehat{A_k} \mid a \in M \cap A_{dom}\}$. A set M of atoms is an *mr-answer set* if M is the unique least model of the NSS-reduct of P with respect to M; one can show that NSS-reducts do have unique least models (Marek and Remmel, 2004).

Definition 10 *Let X be a set of atoms. For a set Y of atoms and a constraint A we define the relation $\triangleright_X^{mr}$ as follows:*

$$Y \triangleright_X^{mr} A \quad \text{if} \quad \text{there exists } Y' \subseteq Y \text{ such that } Y' \models A, \text{ and } X \models A.$$

We will now show that mr-models of P are precisely the mr-answer sets of P. Let M be an mr-answer set of P and Q be the *NSS-reduct* of P with respect to M. By definition, M is the fixpoint of the operator[4] $T_Q(X) = \{hd(r) \ : \ r \in Q, X \models body(r)\}$. Consider the sequence $C = \langle X_i \rangle_{i=0}^{\infty}$ where $X_i = T_Q^i(\emptyset)$. Because every constraint in Q is monotone, we have that T_Q is a monotonic operator. Thus, C is a weak computation. Furthermore, by definition of $\triangleright_M^{mr}$ we have that for every $X \subseteq Y \subseteq M$, $P^{\triangleright_M^{mr}}(X) \subseteq P^{\triangleright_M^{mr}}(Y)$, and hence, $X_{i+1} \in T^{nd;\triangleright_M^{mr}}(X_i)$. This implies that C is a self-justified weak $\triangleright_M^{mr}$-computation whose result is M, i.e., M is an mr-model of P.

Conversely, if M is an mr-model of P, then M is the result of a self-justified weak $\triangleright_M^{mr}$-computation $C = \langle X_i \rangle_{i=0}^{\infty}$. Again, let Q be the *NSS-reduct* of P with respect to M. Since $M \models A$ implies $M \models \widehat{A}$ for every constraint A, we have that M is also a model of Q. Let $Y_i = T_Q^i(\emptyset)$. By definition of the operator $T^{nd;\triangleright_M^{mr}}$, the construction of Q, and the operator T_Q, we can conclude that $X_1 \subseteq Y_1$. Using induction and similar arguments, we conclude that $X_i \subseteq Y_i$. Thus, the least fixpoint of Q, say M', satisfying that $M \subseteq M'$. This, together with the fact that M' is the least model of Q, implies that $M' = M$, i.e., M is an mr-answer set of P.

The approach of self-justified weak $\triangleright_X^f$-computations can be extended further. Namely, instead of regarding f as assigning quasi-satisfiability relations to *sets of atoms*, we could assume that f assigns quasi-satisfiability relations to *computations*. To illustrate this point, let us assume that $C = \langle X_i \rangle_{i=0}^{\infty}$ is a computation and define $\triangleright_C$ as follows: $Y \triangleright_C A$ if $Y = X_i$, for some i, $0 \leq i$, and $X_j \models A$ for every j, $i \leq j$. This particular approach has interesting connections to the basic idea of a reduct. We recall that in the normal case we take a set of atoms as a context, and re-justify it in the "check" phase. Here, we take the whole computation as a context and have to re-justify all its elements in the "check" phase. The corresponding notion of an $\triangleright_C$-model has several desirable properties, yet it seems to be too weak. One can show that $\triangleright_C$-models are answer sets but, in general, not conversely.

[4] Abusing the notation, we use the immediate consequence operator T_P as defined for normal logic programs.

Even in this more general version, the approach to define classes of models through self-justified weak computations has its limitations. Let us recall the semantics for programs with aggregates proposed by Faber et al. (2004) (we restate it for programs with constraints). A set of atoms M is an *FLP-answer set* of a program with constraints P if M is a minimal model of $P(M)$. So far, we have been unable to cast that semantics in terms of self-justified weak computations.

7.4 Answer Sets and Programs with Aggregates

As discussed in other papers considering abstract constraints (Marek and Remmel, 2004; Marek and Truszczyński, 2004; Marek, Niemelä, and Truszczyński, 2008), abstract constraints can be used to represent several extensions of logic programs, including aggregates (e.g., (Denecker et al., 2001; Faber et al., 2004; Ferraris, 2005; Pelov, 2004; Son et al., 2006)). Intuitively, an aggregate atom A can be represented by a constraint $C_A = (A_{dom}, A_{sat})$, where A_{dom} is the set of atoms occurring in the set expression of A and A_{sat} contains the subsets of A_{dom} that satisfy the aggregate. In the remainder of this section, by $\mathcal{A}(P)$ we mean the program with constraints obtained from a program with aggregates by replacing aggregates with the corresponding constraints. Proposition 12, and the results developed by Son, Pontelli, and Elkabani (2006); Son, Pontelli, and Tu (2007), allow us to relate answer sets as we introduced them with semantics for programs with aggregates proposed by others. First, we note that when only monotone aggregates (constraints) are allowed, all approaches coincide.

Proposition 13 ((Son et al., 2007)) *For a program with monotone aggregates, say P, M is an answer set of $\mathcal{A}(P)$ if and only if it is an answer set of P with respect to the definitions proposed by Faber et al. (2004) and Ferraris (2005).*

Another important approach to the semantics of aggregates in logic programming has been investigated by Pelov, Denecker, and Bruynooghe (2004) and Pelov (2004), based on the approximation theory developed in Denecker et al. (2000). The proposal of Pelov et al. (2004); Pelov (2004) defines and exploits the so-called *ultimate approximation* operator Φ_P^{aggr}. A complete (two-valued) interpretation M is an answer set according to Pelov et al. (2004) if it is a fixpoint of Φ_P^{aggr}. If P is a *positive* program with aggregates (the notion defined by Pelov et al. (2004); Pelov (2004)), the program $\mathcal{A}(P)$ has only monotone constraints in bodies of the rules, and we have the following result.

Proposition 14 *Let P be a positive program with aggregates. Then, every answer set of $\mathcal{A}(P)$ is an answer set according to the definition of Pelov et al. (2004) and vice versa.*

The proof of this proposition relies on Proposition 12 and Theorem 3 by Son and Pontelli (2007).

The relationship between the semantics discussed above brakes when we allow programs with aggregates that are not monotone. This is illustrated by the following examples.

Example 9 *Consider the program P*

$$p(1) \leftarrow (\{p(1), p(-1)\}, \{\emptyset, \{p(1), p(-1)\}\})$$
$$p(1) \leftarrow p(-1)$$
$$p(-1) \leftarrow p(1)$$

Intuitively, the abstract atom $A = (\{p(1), p(-1)\}, \{\emptyset, \{p(1), p(-1)\}\})$ represents the aggregate atom $\text{SUM}(\{X \mid p(X)\}) = 0$ or $\text{COUNT}(\{X \mid p(X)\}) \neq 1$ (where the domain of the variable X is $\{1, -1\}$). This program has a model $M_1 = \{p(1), p(-1)\}$. The approaches by Marek and Remmel (2004), Faber et al. (2004), and Ferraris (2005) accept M_1 as an answer set, while our approach and that by Pelov (2004) and Denecker et al. (2001) do not admit any answer sets. It is easy to see that there is no founded computation for P whose result is M_1. In fact, this program does not have a computation: the only applicable rule given $X_0 = \emptyset$ is the first rule and so $X_1 = \{p(1)\}$; the only applicable rule with respect to X_1 is the last rule, and thus, $X_2 = \{p(-1)\}$; and the sequence violates the property $(\mathbf{P}')$.
Alternatively, we could argue against semantics allowing $\{p(1), p(-1)\}$ as an answer set by noting that the model $\{p(1), p(-1)\}$ is self-supported. Indeed, it is not strongly grounded, that is, no "proper" ranking of its elements can be found (we note that the program P is isomorphic to the program P_6 considered in Example 8).
Let us now consider the program consisting of two rules

$$p \leftarrow p$$
$$p \leftarrow (\{p\}, \emptyset)$$

The approach by Pelov et al. (2004) accepts $M = \{p\}$ as an answer set, while our approach does not. □

Finally, we note that the proposals developed by Faber et al. (2004) and Pelov (2004) do not allow aggregates in the head of the rules, but they consider disjunctive logic programs with aggregates.

7.5 Other Semantics of Programs with Abstract Constraints

We have already mentioned the relationship between the computation characterization of answer sets and the characterization proposed by Son et al. (2007)—the latter can be described by the sub-satisfiability relation $\rhd_X^{spt}$.
For the semantics proposed by Marek and Remmel (2004) we have the following observations as corollaries from our results and the results by Son et al. (2007):

(1) Each answer set of P according to the definition given in this paper is also an answer set according to Marek and Remmel—this result can be derived from the observation that the analogous property holds for answer sets as discussed

by Son et al. (2007).

(2) There are answer sets according to Marek and Remmel that are not answer sets according to the definitions in this paper. This can be seen in the program:

$$a \leftarrow$$
$$b \leftarrow$$
$$c \leftarrow (\{a, b, c\},\ \ \{\{a\}, \{a, b, c\}\})$$

The characterization by Marek and Remmel accepts $\{a, b, c\}$ as an answer set. On the other hand, a computation, in order to produce $\{a, b, c\}$ will need to be organized as follows: $\emptyset, \{a\}, \{a, b, c\}, \ldots$. But this computation is clearly not founded.

Marek and Truszczyński (2004) and Marek et al. (2008) propose a characterization of answer sets for program with monotone constraints, later extended to the case of convex constraints by Liu and Truszczyński (2005). These two semantics are proved to coincide with the answer sets obtained using $\rhd_X^{spt}$ by Son et al. (2007), that is, with answer sets as defined in this paper.

Another interesting line of research has been recently proposed by Shen and You (2007). The authors propose a model-theoretic semantic characterization for programs with constraints which is based on a generalized Gelfond-Lifschitz transformation. The resulting semantics is proved to coincide with the notion of answer sets we discuss in this work (and with the answer sets as defined by Son et al. (2007)). This provides further reinforcement of the validity of the semantics we discuss in this paper. The work by Shen and You (2007) nicely complements our computation-based approach, by offering a model-theoretic characterization of answer sets.

8 Conclusions

In this paper, we conducted an in-depth investigation of semantics of logic programs with general abstract constraints. Programs with such constraints are important. Arbitrary abstract constraints subsume many types of constraints and aggregates that arise in practice. Moreover, the use of arbitrary constraints allows us to eliminate the explicit use of the negation as failure without compromising the expressive power. Our effort extends and complements earlier proposals, generalizing the well-established notions of answer sets of normal logic programs, and of programs with monotone and convex abstract constraints.

The backbone of our proposal is the notion of a *computation*, viewed as a regulated sequence of interpretations. Computations are specified in terms of some basic principles: revision, convergence, persistence of beliefs, persistence of reasons, and founded persistence of reasons. These principles have been derived through an analysis of properties of answer sets of normal logic programs. Building on that connection, we proposed as answer sets of programs with arbitrary constraints the results of computations that are *founded*, that is, satisfy all the properties listed

above.

The problem of assigning an "answer-set" semantics to programs with arbitrary constraints has received much attention lately. In the paper, we compared our proposal with several alternative ones. Clearly, the question which of the proposed generalizations of the answer-set semantics from the case of normal programs to the case of programs with arbitrary constraints is the "correct" one cannot be given a definitive answer. Indeed, the concept of "correctness" does not have a formal definition. However, one can identify some desirable properties that answer sets should satisfy, and evaluate proposed semantics based on how they behave relative to those properties.

In this respect, we note that our concept of an answer set has a strong constructive flavor. It is rooted in the notion of a computation which, in turn, is based on some fundamental principles computations should obey. Next, it coincides with an earlier proposal by Son et al. (2007) and, as should be expected of any semantics of answer sets for programs with arbitrary constraints, it generalizes answer sets of normal logic programs and of programs with convex constraints. Furthermore, answer sets as defined here have several equivalent characterizations. They are: the definition provided in Section 5, the definition given by Son et al. (2007), a closely related characterization in terms of sub-satisfiability relations from Section 6, and the characterization in terms of a transformation of programs with arbitrary constraints to programs with convex ones. All these characterizations point to the multitude of intuitions that underlie the semantics we proposed here. Lastly, our answer sets are free of self-supportedness, a feature lacking in other proposals for the semantics of programs with arbitrary constraints, most notably the proposal developed by Faber et al. (2004).

We conclude by noting a challenging open problem. Namely, so far we have been unable to extend our computation-based approach so that to capture as a special case the semantics of answer sets of *disjunctive* logic programs. Answer sets of disjunctive programs are minimal models. Therefore, we are after a class of computations that are guaranteed to produce minimal models only. However, finding such a concept of a computation is not easy. In particular, an obvious attempt to require that computations increase minimally in each step of a computation does not address the problem (Pelov and Truszczyński, 2004). For instance, let us consider a program consisting of two rules: $a \vee b$ and $b \leftarrow a$. Every reasonable class of computations (that do not look ahead to the result but decides how to expand based on what has been computed so far) seems to have to contain the computation $\emptyset, \{a\}, \{a, b\}, \{a, b\}, \ldots\}$. But the result of that computation is not a minimal model. In the same time, there is no way to determine that computing $\{a\}$ in the first step is incorrect based only on the current (in this case, initial) state of the computation. It seems possible that in order to capture minimality it is necessary to impose some "global" minimality requirement (as opposed to "local" definition of how to increment from step to step). This difficulty seems also to be at the heart of the problem of expressing the semantics of Faber et al. (2004) in terms of computations, which we noted earlier.

Acknowledgments

The authors wish to thank the anonymous reviewers for their insightful comments.

References

Apt, K., 1990. Logic programming. In: van Leeuven, J. (Ed.), Handbook of theoretical computer science. Elsevier, Amsterdam, pp. 493–574.

Balduccini, M., Gelfond, M., Nogueira, M., 2006. Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence, 47(1–2):183–219.

Baral, C., 2003. Knowledge Representation, Reasoning, and Declarative Problem Solving, Cambridge University Press.

Baral, C., 2005. From Knowledge to Intelligence – Building Blocks and Applications. Invited Talk, AAAI, `www.public.asu.edu/~cbaral/aaai05-invited-talk.ppt`.

Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A., 2001. Complexity and Expressive Power of Logic Programming. ACM Computing Surveys, 33(3):374–425.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G., 2003. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003. pp. 847–852.

Denecker, M., Marek, V., Truszczyński, M., 2000. Ultimate semantic treatment of default and autoepistemic logics. In: Proceedings of KR, Principles of Knowledge Representation and Reasoning, pp. 74–84.

Denecker, M., Pelov, N., Bruynooghe, M., 2001. Ultimate well-founded and stable semantics for logic programs with aggregates. In: Codognet, P. (Ed.), Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings. Vol. 2237 of Lecture Notes in Computer Science. Springer, pp. 212–226.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A., 2003. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. Artificial Intelligence 144 (1-2):157–211.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A., 2003. Answer Set Planning Under Action Costs. Journal of Artificial Intelligence Research, 19:25–71.

Elkabani, I., Pontelli, E., Son, T. C., 2004. Smodels with CLP and Its Applications: A Simple and Effective Approach to Aggregates in ASP. In: Demoen, B., Lifschitz, V. (Eds.), Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings. Vol. 3132 of Lecture Notes in Computer Science. Springer, pp. 73–89.

Erdem, E., Lifschitz, V., 2003. Tight logic programs. Theory and Practice of Logic Programming, 3(4-5):499–518.

Erdem, E., Lifschitz, V., Ringe, D., 2006. Temporal phylogenetic networks and logic programming. Theory and Practice of Logic Programming, 6(5):539–558.

Faber, W., Leone, N., Pfeifer, G., 2004. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J. J., Leite, J. A. (Eds.), Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings. Vol. 3229 of Lecture Notes in Computer Science. Springer, pp. 200–212.

Ferraris, P., 2005. Answer sets for propositional theories. In: Baral, C., Greco, G., Leone, N., Terracina, G. (Eds.), Logic Programming and Non-monotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings. Vol. 3662 of Lecture Notes in Computer Science. Springer, pp. 119–131.

Gebser, M., Kaufmann, B., Schaub, T., 2007. *clasp*: a conflict driven answer set solver. In: Baral, C., Brewka, G., Schlipf, J. (Eds.), Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, USA, May 15–17, 2007, Proceedings. Vol. 4483 of Lecture Notes in Computer Science. Springer, pp. 260–265.

Gelfond, M., 2002. Representing Knowledge in A-Prolog. In: Kakas, A., Sadri, F. (Eds.), Computational Logic: Logic Programming and Beyond. Springer Verlag, pp. 413–451.

Gelfond, M., Leone, N., 2002. Logic programming and knowledge representation – the A-Prolog perspective. Artificial Intelligence, 138(1-2):3–38.

Gelfond, M., Lifschitz, V., 1988. The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (Eds.), Logic Programming: Proceedings of the Fifth International Conference and Symposium, The MIT Press, pp. 1070–1080.

Gelfond, M., Lifschitz, V., 1990. Logic programs with classical negation. In: Warren, D., Szeredi, P. (Eds.), Logic Programming: Proceedings of the Seventh International Conference, The MIT Press, pp. 579–597.

Heljanko, K., Niemelä, I., 2003. Bounded LTL model checking with stable models. Theory and Practice of Logic Programming, 3(4–5):519–550.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F., 2006. The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic, 7(3):499–562.

Lierler, Y., Maratea, M., 2004. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: Lifschitz, V., Niemelä, I. (Eds.), Proceedings of the 7th International Conference on Logic Programming and Non-monotonic Reasoning Conference (LPNMR'04). Vol. 2923. Springer Verlag, LNCS 2923, pp. 346–350.

Liu, L., Pontelli, E., Son, T. C., Truszczyński, M., 2007. Logic programs with abstract constraint atoms: The role of computations. In: Dahl, V., Niemelä, I. (Eds.), Proceedings of the 23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal, September 8-13. Vol. 4670 of Lecture Notes in Computer Science. Springer, pp. 286–301.

Liu, L., Truszczyński, M., 2005. Properties of programs with monotone and convex constraints. In: Veloso, M. M., Kambhampati, S. (Eds.), Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth

Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA. AAAI Press AAAI Press / The MIT Press, pp. 701–706.

Lloyd, J., 1987. Foundations of logic programming, 2nd edition. Springer Verlag.

Marek, V. W., Nerode, A., Remmel, J. B., 1999. Logic programs, well-orderings, and forward chaining. Ann. Pure Appl. Logic, 96(1-3):231–276.

Marek, V. W., Niemelä, I., Truszczyński, M., 2008. Logic programs with monotone abstract constraint atoms. Theory and Practice of Logic Programming, 8(2):167–199.

Marek, V. W., Remmel, J. B., 2004. Set constraints in logic programming. In: Logic Programming and Non-monotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004, Proceedings. Vol. 2923 of Lecture Notes in Computer Science. Springer Verlag, pp. 167–179.

Marek, V. W., Truszczyński, M., 1999. Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-year Perspective, Springer Verlag, pp. 375–398.

Marek, V. W., Truszczyński, M., 2004. Logic programs with abstract constraint atoms. In: Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA. AAAI Press / The MIT Press, pp. 86–91.

Marek, W., Truszczyński, M., 1993. Non-monotonic Logic: Context dependent reasoning. Springer Verlag.

Niemelä, I., 1999. Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence, 25(3–4):241–273.

Niemelä, I., Simons, P., 1997. SMODELS - an implementation of the stable model and well-founded semantics for normal logic programs. In: In: Logic Programming and Non-monotonic Reasoning, 7th International Conference, LPNMR 1997, Springer Verlag, pp. 420–429.

Pelov, N., 2004. Semantic of Logic Programs with Aggregates. Ph.D. thesis, Katholieke Universiteit Leuven, `www.cs.kuleuven.ac.be/publicaties/doctoraten/cw/CW2004_02.abs.html`.

Pelov, N., Denecker, M., and Bruynooghe, M. Partial Stable Models for Logic Programs with Aggregates. In: Logic Programming and Non-monotonic Reasoning, 7th International Conference, LPNMR 2004, Springer Verlag, pp. 207–219.

Pelov, N., Truszczyński, M., 2004. Semantics of Disjunctive Programs with Monotone Aggregates — an Operator-Based Approach. In: Delgrande, J.P., Schaub, T. (eds.) Proceedings of the 10th International Workshop on Non-Monotonic Reasoning, NMR-04, pp. 327–334.

Shen, Y.-D., You, J.-H., 2007. A Generalized Gelfond-Lifschitz Transformation for Logic Programs with Abstract Constraints. In: AAAI. AAAI Press, pp. 483–488.

Simons, P., Niemelä, N., Soininen, T., 2002. Extending and Implementing the Stable Model Semantics. Artificial Intelligence, 138:(1–2):181–234.

Son, T. C., Pontelli, E., 2007. A Constructive Semantic Characterization of Aggre-

gates in Answer-Set Programming. Theory and Practice of Logic Programming, 7(3):355–375.

Son, T. C., Pontelli, E., Elkabani, I., 2006. An Unfolding-Based Semantics for Logic Programming with Aggregates. Computing Research Repository. CS.SE/0605038.

Son, T. C., Pontelli, E., Tu, P. H., 2007. Answer Sets for Logic Programs with Arbitrary Abstract Constraint Atoms. Journal of Artificial Intelligence Research, 29:353–389.

van Emden, M., Kowalski, R., 1976. The semantics of predicate logic as a programming language. Journal of the ACM, 23(4):733–742.

Van Gelder, A., Ross, K., Schlipf, J., 1991. The well-founded semantics for general logic programs. Journal of ACM, 38(3):620–650.

You, J.-H., Yuan, L.-Y., Liu, G., Shen, Y.-D., 2007. Logic Programs with Abstract Constraints: Representation, Disjunction and Complexities. In: Baral, C., Brewka, G., Schlipf, J. S. (Eds.), Logic Programming and Non-monotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings. Vol. 4483 of Lecture Notes in Computer Science. Springer, pp. 228–240.