Logic Programming for Knowledge Representation

Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA, mirek@cs.uky.edu

Abstract. This note provides background information and references to the tutorial on recent research developments in logic programming inspired by need of knowledge representation.

1 Introduction

McCarthy and Hayes [46] wrote: "[...] intelligence has two parts, which we shall call the epistemological and the heuristic. The epistemological part is the representation of the world in such a form that the solution of problems follows from the facts expressed in the representation. The heuristic part is the mechanism that on the basis of the information solves the problem and decides what to do." The epistemological part is the concern of knowledge representation. The heuristic part is typically addressed by search.

While not stated explicitly in McCarthy and Hayes' definition, the knowledge representation and the search are closely intertwined. Modeling features of the language affect the design of search methods. Conversely, the availability of fast search techniques for particular computational tasks, for instance, proof finding or model computation, in the case of particular classes of theories, such as Horn theories or propositional theories in CNF, influences the design of modeling languages. Effective computational knowledge representation systems require that the two are integrated.

Logic programming has long been regarded as a prime candidate for a practical instantiation of computational knowledge representation. First, logic program clauses align well with natural language constructs humans use to specify constraints [31,32]. Next, logic programs (Horn logic programs, to be precise) are Turing complete [1]. Finally, there are well understood automated proof techniques for reasoning with logic programs [50,32]. These considerations led to the design of Prolog [6], still a dominant computational knowledge representation language.

However, the presence of negation in the bodies of logic program rules, while convenient from the modeling standpoint, posed a challenge. The semantics of logic programs with negation was not clear and resolving that issue required a major research effort. The completion semantics [5] was the first attempt to address the problem. The answer-set semantics [27,28] provided a definitive solution within the class of 2-valued semantics.

For some time after its introduction, the answer-set semantics was a source of confusion. It was unclear how to use it in practice when modeling application domains, and how to reconcile it with the traditional proof-theory approach to logic programming. The *answer-set programming* paradigm [43,48] offered an alternative to proof-based logic programming and shifted the focus from proof-finding to model-finding. Under the answer-set programming paradigm, a problem is modeled as a logic program so that *answer sets* of the program expanded with an encoding of a particular instance of the problem (not substitutions associated with proofs) correspond to solutions to the problem for that instance.

With the understanding of the meaning of programs with negation came additional evidence of the applicability of logic programs in knowledge representation. According to [26], the goal of knowledge representation is "to design and study languages to capture knowledge about environments, their entities and their behaviors." Such languages must be able to handle modeling challenges posed by the qualification and frame problems, defaults, conditionals and normative statements, (inductive) definitions, and by the need for the elaboration tolerance. Research showed that answer-set programming, provides means that adequately address these problems [4,25,26,3]. While answer-set programming comes with the penalty of syntactic restrictions (no function symbols) and so, the limited expressive power (the class NP-search for normal logic programs, and Σ_2^P-search for disjunctive programs), it has major advantages. First, arguably the task of modeling gets much simpler than in proof-based logic programming — answer-set programs are truly declarative. Second, it becomes possible to take advantage of fast search techniques developed in the area of propositional satisfiability.

In this note I will discuss three recent research directions in logic programming inspired by knowledge representation needs. First, knowledge bases must be constructed in a modular fashion. This brings up the question of equivalence of answer-set programs, as well as the need for methods to decompose programs so that answer-sets of the program can be recovered from answer sets of its components. Second, there has been a need for extending the syntax of answer-set programs with means to model numeric constraints. Such constraints are ubiquitous and, in particular, are common in knowledge representation applications. Finally, there are logics other than logic programs with the answer-set semantics that also give rise to knowledge representation systems based on the principle that models of theories describe problem solutions. One of the most promising such approaches, the ID-logic [8], combines first-order logic (which is used to model constraints) with logic programs under the well-founded semantics [58], a 3-valued approximation to the answer-set semantics (which are used to represent definitions).

Theoretical advances in answer-set programming would remain just that, if they were not followed by practical applications. These applications require working and effective answer-set programming software. There has been much work in that area. I will conclude this paper with brief comments on and references to some of the state-of-the-art implementations.

2 Modularity

A basic design principle in knowledge representation is the principle of *modularity*. It stipulates that knowledge bases be composed of *modules*, each representing a fragment of an application domain being modeled. The principle of modularity gave rise to research problems that have generated much interest among logic programming and answer-set programming researchers. Two of them that I will discuss are:

1. To characterize cases when two knowledge base modules are *equivalent for substitution*, and
2. To improve the efficiency of processing algorithms by taking advantage of the modular structure of the knowledge base.

Informally, two modules are equivalent for substitution if replacing one with the other does not affect the meaning of the knowledge base. When optimizing the knowledge base for conciseness, efficiency of processing, robustness to change, or other appropriate measure of quality, one approach is to optimize each module separately. For that approach to be safe, though, the optimized module should be equivalent for substitution to the original one. Otherwise, optimization might have unwanted side effects. In a related way, the concept of equivalence for substitution has applications in knowledge-base query optimization. Thus, the notion of equivalence for substitution is an important one.

When a knowledge base is represented by a logic program under the answer-set semantics, one can formalize the concept of the equivalence for substitution, or *strong equivalence*, the term more commonly used in answer-set programming, as follows: two programs P and Q are *strongly equivalent* if for every program R, the programs $P \cup R$ and $Q \cup R$ have the same answer sets. Indeed, if P and Q have this property, one can replace P with Q or Q with P within any larger program, and the answer sets (a formal description of the "meaning") will not change.

The concept of strong equivalence was introduced in [33]. That paper also presented a complete characterization of strong equivalence in terms of the equivalence in the logic *here-and-there* [29]. We note in passing that while a necessary condition, having the same answer sets is not sufficient for two programs to be strongly equivalent. Thus, a most direct attempt at extending the concept of equivalence from the classical logic to answer-set programming does not work. [36,56] presented simple characterizations of strong equivalence that do not make explicit references to the logic *here-and-there*. [55] cast the concept in terms of the equivalence in the modal logic S4F. Several generalizations and variations of the notion were proposed and studied in [15,17,19,16,34,23]. [54] extended the problem of strong equivalence to an abstract algebraic setting of the approximation theory of operators on lattices [9].

Modularity can also be exploited in processing programs. It is well known that the problem of the existence of an answer set of a logic program is NP-complete for normal propositional logic programs [45] and Σ_2^P-complete for the disjunctive ones [18]. Modular structure of the program can have dramatic effect on the complexity of this decision problem and the associated search task.

This has been known for quite some time in the case when the dependencies among modules are acyclic. [2] proposed the class of stratified programs, in which dependencies between modules are acyclic, and dependencies within modules are subject to certain restrictions. Each stratified normal logic program has exactly one answer set. Moreover, it can be computed efficiently. The results on stratification have been generalized in the form of a splitting theorem, to the case when the dependencies between modules are still acyclic but the structure of each module is not restricted anymore [35]. When a program can be "split", a form of divide-and-conquer approach can significantly speed up the process of computing answer sets. It is worth noting that, as strong equivalence, splitting also has an algebraic description within the approximation theory [60,59].

Recently, researchers focused on the general case, when dependencies among modules are not acyclic [21,30]. By controlling the way, in which modules interact, [30] managed to characterize answer sets of the overall program in terms of answer sets of individual modules and outlined several possible applications for this general result.

3 Programs with constraints

Numeric constraints are common. Modeling them in the basic language of first-order logic is, however, a tedious task. It requires auxiliary atoms and leads to large programs with no transparent meaning. The problem has been long recognized in the area of database systems, where queries often concern numeric properties of sets of records. To make the process of formulating such queries easier, database query languages are equipped with syntax that provides explicit means to express numeric constraints, referred to in the field of databases as *aggregates*. The same holds true of constraint programming languages.

Applications of answer-set programming in knowledge representation brought up the same problem and motivated extensions of the basic language of program rules with syntax to model constraints directly [51,7,10,20,22,49,52]. These approaches agree on many classes of programs. However, they differ in intuitions, as well as in some technical aspects. To gain a better understanding of extensions of answer-set programming with constraints, and to offer a more principled approach to the semantics of such extensions, researchers proposed and studied answer-set programming formalisms based on *abstract constraint atoms* [42,44,41,53,38]. This approach lead to a theory of programs with constraints based on the concept of a *computation* driven by a generalization of the one-step provability operator [57]. In the case of programs with *monotone* and *convex* constraints the parallels with the normal logic programming are very strong [41]. Recent work extended these parallels also to the case of arbitrary constraints [38]. We will now present, following closely [44] and [40], some of the basic aspects of the theory of programs with abstract constraints concentrating on the case when these constraints are monotone.

We consider a language determined by a fixed countable set At of *propositional variables*. An *abstract constraint atom* is a syntactic expression $A = (C, X)$, where $X \subseteq At$ is a *finite* set of propositional variables, called the *domain* of A, and $C \subseteq \mathcal{P}(At)$ is the set of *satisfiers* of A. We will write A_{dom} for X and A_{sat} for C.

A propositional interpretation $M \subseteq At$ *satisfies* an abstract constraint atom A, denoted $M \models A$, if $M \cap A_{dom} \in A_{sat}$, that is, if the set of elements in the domain of A that are true in M is a satisfier of A.

A *constraint program* is a set of *constraint* rules, that is, expressions of the form

$$A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m) \qquad (1)$$

where A, $A_1, \ldots, A_m$ are constraints and **not** denotes *default negation*. The constraint A is the *head* and the set $\{A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ is the *body* of r. The concept of satisfiability described above extends in the standard way to constraint rules and programs.

We denote by $At(P)$ the set of atoms in the domains of constraints in a constraint program P. We denote by $hset(P)$, the *headset* of P, that is, the union of the domains of the heads of all rules in P.

We will now list several basic definitions and properties of constraint programs mirroring those of normal ones, and culminating with a generalization of the concept of the answer-set.

M-applicable rules. Let $M \subseteq At$ be an interpretation. A rule (1) is M-*applicable* if M satisfies every literal in the body of r. We write $P(M)$ for the set of all M-applicable rules in P.

Supported models. An interpretation M is a *supported* model of a constraint program P if M is a model of P and $M \subseteq hset(P(M))$.

Nondeterministic one-step provability. An interpretation $M' \subseteq At$ is *nondeterministically one-step provable* from an interpretation $M \subseteq At$ by means of a constraint program P, if $M' \subseteq hset(P(M))$ and for every head A of a rule in $P(M)$, $M' \models A$. Given a constraint program P, the *nondeterministic one-step provability operator* T_P^{nd} is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are nondeterministically one-step provable from M by means of P.

Monotone constraints and monotone-constraint programs. A constraint A is *monotone* if for every $M, M' \subseteq At$, $M \subseteq M'$ and $M \models A$ together imply that $M' \models A$. A *monotone-constraint program* is a program built of monotone constraints.

Horn constraint programs. A rule (1) is *Horn* if constraints A, $A_1, \ldots, A_k$ are monotone, and if $k = m$ (no occurrences of default negation). A constraint program is *Horn* if every constraint rule in the program is Horn.

Bottom-up computations for Horn programs. Let P be a Horn constraint program. A *P-computation* is a sequence $\langle X_k \rangle_{k=0}^{\infty}$ such that $X_0 = \emptyset$ and for every k,

$$X_k \subseteq X_{k+1}, \quad \text{and} \quad X_{k+1} \in T_P^{nd}(X_k).$$

The *result* of a P-computation $t = \langle X_k \rangle$ is the set $\bigcup_k X_k$. We denote it by R_t.

We note that if P is a Horn constraint program and t is a P-computation, R_t is a supported model of P.

Derivable models. A set M of atoms is a *derivable model* of a Horn constraint program P if there exists a P-computation t such that $M = R_t$. If a Horn constraint program has a model, one can show that it has computations and, consequently, derivable models. By our comment above, derivable models of a Horn constraint program P are supported models and so, in particular, models of P.

The reduct. Let P be a monotone-constraint program and M a subset of $At(P)$. The *reduct* of P with respect to M, P^M, is the Horn constraint program obtained from P by: (1) removing from P all rules whose body contains a literal $\mathbf{not}(B)$ such that $M \models B$; and (2) removing literals $\mathbf{not}(B)$ for the bodies of the remaining rules.

Answer sets. Let P be a program. A set of atoms M is an *answer set* of P if M is a derivable model of P^M. Since P^M is a Horn constraint program, the definition is sound.

The definitions of the reduct and of answer sets follow and generalize the corresponding definitions proposed for normal logic programs, as in the setting of Horn constraint programs, derivable models play the role of a least model.

As in normal logic programming, answer sets of monotone-constraint programs are supported models and, consequently, models. As shown in [51], monotone-constraint programs generalize programs with weight atoms [51]. Some other results one can prove for monotone-constraint programs are extensions of the characterizations of strong and uniform equivalence of programs, of the concept of the program completion [5], and of loop formulas [37]. The latter two concepts proved useful in designing a solver *pbmodels* [39] for computing answer sets of programs with weight constraints in the syntax of *smodels*. *Pbmodels* placed second in two events of the 1st Answer-Set Programming Contest [24].

We concentrated here on monotone-constraint programs. Programs built of more general classes of constraints offer additional challenges. The class of *convex* constraint atoms (A is convex if for every M, M' and M'' such that $M' \models A$, $M'' \models A$ and $M' \subseteq M \subseteq M''$, $M \models A$, as well) is most closely related to the class of monotone constraints and the basic approach described above extends. For arbitrary constraint atoms there is no simple generalization due to non-monotone behavior of such constraints. [38] developed a principled approach to the problem of the answer-set semantics for programs with arbitrary constraints. However, the general approach of [38] resulted in three candidate semantics, none of which has emerged as the definitive one for programs with arbitrary constraints.

4 Model expansion and ID-logic

Several fundamental knowledge representation and reasoning problems can be stated formally as search problems, and solved by general search methods. In fact, this observation lies behind the answer-set programming paradigm. We will now introduce a simple formalism for modeling search problems and show

how to integrate it with logic programming to produce an effective answer-set programming formalism. This approach stems from the work on the formalism called datalog with constraints [13,14] and research on the role of definitions in knowledge representation [8]. It has been actively studied and developed further in [11,47,12].

We start with basic terminology. A *signature* is a nonempty set σ of relation symbols, each with a positive integer arity. Let U be a fixed infinite countable set (the *universe*), and let σ be a signature. An *instance* of σ over U is a set I of finite relations over U, such that there is a one-to-one correspondence $r \leftrightarrow r^I$ between σ and I, and the corresponding relation symbols r and relations r^I are of the same arity.

We denote by Inst_σ the set of all instances of σ. If $I \in \mathit{Inst}_\sigma$, we define the *domain* of I, $\mathit{dom}(I)$, to be the set of all those elements of U that appear in a tuple of a relation in I. Since all relations in I are finite, $\mathit{dom}(I)$ is finite, too. Let $\sigma' \subseteq \sigma$ be signatures. We say that $K \in \mathit{Inst}_\sigma$ *expands* $I \in \mathit{Inst}_{\sigma'}$ if $\mathit{dom}(I) = \mathit{dom}(K)$ and for every $r \in \sigma'$, $r^I = r^K$. We write $I = K_{|\sigma'}$ to denote that K expands I.

The formalism of *model expansion* (the *logic MX*) is based on the language $\mathcal{L}_\sigma$ of the first-order logic, determined by a signature σ (thus, we assume here no function symbols).

An instance $I \in \mathit{Inst}_\sigma$ determines a first-order logic interpretation $\langle \mathit{dom}(I), I\rangle$ of $\mathcal{L}_\sigma$. With some abuse of notation, for an instance $I \in \mathit{Inst}_\sigma$ and a sentence $\varphi \in \mathcal{L}_\sigma$, we write $I \models \varphi$ instead of $\langle \mathit{dom}(I), I\rangle \models \varphi$.

Let $\sigma' \subseteq \sigma$ be signatures and let $\varphi \in \mathcal{L}_\sigma$ be a sentence. Given an instance $I \in \mathit{Inst}_{\sigma'}$ we call an instance $K \in \mathit{Inst}_\sigma$ an *I-model* of φ if K expands I and $K \models \varphi$ (hence the term *model* expansion). The concept extends in a straightforward way to sets of sentences. We will refer to finite sets of sentences interpreted by I-models as *MX-theories*.

Given signatures $\sigma' \subseteq \sigma$, we can regard an MX-theory T from $\mathcal{L}_\sigma$ as an encoding of a search problem. This problem has Inst_σ as the set of its instances and, for every instance $I \in \mathit{Inst}_\sigma$, I-models as solutions to the instance I.

One can show that the expressive power of MX-theories is the same as that of (normal) logic programs [13]. In some cases, though, the task of modeling the search problems as MX-theories is significantly more complicated than when answer-set programs are used. Modeling definitions is often particularly cumbersome. To address that issue, researchers proposed extensions of the formalism MX with logic programs which, under the *well-founded* semantics [58], are well suited to model definitions [8].

We will present one such extension, the ID-logic [8,12,47]. Let $\sigma' \subseteq \sigma$ be signatures. A theory T modeling a search problem in the formalism of the ID-logic consists of two parts. An MX-theory T' in $\mathcal{L}_\sigma$ forms one of the parts. A normal logic program T'', also in $\mathcal{L}_\sigma$ but with no relation symbol from σ' in the head of a rule, forms the other one. Let σ^{def} be the set of relation symbols in the heads of the rules of the program T''. Intuitively, these are the symbols that need to be defined and that are defined in the ID-logic theory T by its component T''.

Given an instance $I \in Inst_{\sigma'}$, an instance $K \in Inst_{\sigma}$ is an I-*model* of T if K is an I-model of T' and $K_{|\sigma^{def}}$ is the *total* well-founded model of the program $T'' \cup K_{|\sigma \setminus \sigma^{def}}$[1]. Informally, to obtain an I-model of an ID-logic theory T, we guess the extensions of all relation symbols in σ (keeping the extensions of the relation symbols in σ' as they are specified by I and not introducing any new constants), and verify that the extensions of the relation symbols in σ^{def} are fully determined by the program T'' and the extensions of the relation symbols not in σ^{def} under the well-founded semantics.

The ID-logic (under the restriction of no function symbols in the language) has the same expressive power as (non-disjunctive) answer-set programming [14]. However, arguably, its semantics is simpler, as it relies on two intuitive concepts: constraints and definitions. Consequently, the ID-logic has a significant potential for knowledge representation applications. A practical demonstration of the modeling capabilities of ID-logic can be found in [11].

5 Tools

While theoretical challenges make answer-set programming an exciting research area, it is because of the existence of effective computational tools, effective enough to handle several classes of "industrial-grade" problems, that it is steadily gaining on importance. We will now briefly review some of these tools.

The general approach to processing answer-set programs consists of two steps: (1) *grounding*, that is, instantiating and simplifying an input program with variables into a propositional program, and (2) *solving*, that is, computing answer sets of the program resulting from step (1). The grounding step is designed so that all answer sets of the original program can be recovered in step (2).

The *lparse/smodels* software (`www.tcs.hut.fi/Software/smodels/`) constitutes the earliest attempt at making answer-set programming practical. It remains widely used. Arguably, *lparse* is the most widely used grounder by solver developers, and *smodels* remains one of the most competitive solvers.

The *dlv* system (`www.dbai.tuwien.ac.at/proj/dlv/`) was proposed soon after *lparse/smodels*. The *dlv* offers an integrated grounder and solver package capable of computing answer-sets of disjunctive programs with aggregates. With its front-ends for SQL, inheritance, planning, and abduction and diagnosis, it emerges as the most flexible answer-set programming system at present. In this context, it is important to note that *dlv* won the 1st Answer-Set Programming Contest in the modeling/grounding/solving category [24].

Among software addressing only one of the stages in the computation of answer sets, *gringo* `gringo.sourceforge.net/` is a recent newcomer in the area of grounders. Noteworthy solvers are *clasp* (`www.cs.uni-potsdam.de/clasp/`), *pbmodels* (`www.cs.uky.edu/ai/pbmodels/`), *cmodels* (`www.cs.`

[1] We abuse the notation here by viewing instances as sets of the corresponding ground atoms in the language extending $\mathcal{L}_\sigma$ with the elements of U as constants.

utexas.edu/users/tag/cmodels.html), as well as *gnt* (www.tcs.hut.fi/Software/gnt/). Each of these solvers performed very well in the 1st Answer-Set Programming Contest [24], with *clasp* winning in two events. Other notable solver is *assat*, which pioneered the idea of using loop formulas to reduce answer-set computation to propositional satisfiability.

Finally, we mention software for answer-set programming systems based on the ID-logic. The grounder *psgrnd* (www.cs.uky.edu/ai/aspps/) can process ID-logic theories, in which the first-order component is in the clausal form (possibly with weight atoms), and the logic program component is a Horn program. The solver *aspps* (www.cs.uky.edu/ai/aspps/) is designed to compute answer-sets of ground ID-logic theories satisfying these restrictions. Recently more general tools have been developed, including *MXG* (www.cs.sfu.ca/research/groups/mxp/mxg/), an integrated software package for grounding and computing models of ID-logic theories, and *GidL/MidL* package (www.cs.kuleuven.be/~dtai/krr/software/idp.html) for grounding (GidL) and model generation (MidL) of theories in the ID-logic.

6 Closing comments

Answer-set programming, a variant of logic programming with the answer-set semantics, and formalisms such as ID-logic, which combine first-order logic with logic programming under the well-founded semantics, are well suited for knowledge representation applications. After about a decade since they have been proposed, they continue to generate theoretical research challenges. In the same time, thanks to the development of computational software, their practical importance is steadily growing.

Acknowledgments

The author acknowledges the support of NSF grant IIS-0325063 and KSEF grant 1036-RDE-008.

References

1. H. Andréka and I. Németi, *The generalized completeness of Horn predicate logic as a programming language*, Acta Cybernetica **4** (1978/79), no. 1, 3–10.
2. K. Apt, H.A. Blair, and A. Walker, *Towards a theory of declarative knowledge*, Foundations of deductive databases and logic programming, Morgan Kaufmann, 1988, pp. 89–142.
3. C. Baral, *Knowledge Representation, Reasoning and Declarative Problem Solving*, Cambridge University Press, 2003.
4. C. Baral and M. Gelfond, *Logic programming and knowledge representation*, Journal of Logic Programming **19/20** (1994), 73–148.
5. K.L. Clark, *Negation as failure*, Logic and data bases, Plenum Press, New York-London, 1978, pp. 293–322.

6. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel, *Un systeme de communication homme-machine en francais*, Tech. report, University of Marseille, 1973.
7. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and Gerald Pfeifer, *Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV*, Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Morgan Kaufmann, 2003, pp. 847–852.
8. M. Denecker, *The well-founded semantics is the principle of inductive definition*, Proceedings of the 6th European Workshop on Logics in Artificial Intelligence, LNAI, vol. 1489, Springer, 1998, pp. 1–16.
9. M. Denecker, V. Marek, and M. Truszczyński, *Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning*, Logic-Based Artificial Intelligence, Kluwer Academic Publishers, 2000, pp. 127–144.
10. M. Denecker, N. Pelov, and M. Bruynooghe, *Ultimate well-founded and stable semantics for logic programs with aggregates*, Proceedings of the 17th International Conference on Logic Programming (ICLP 2001), LNCS, vol. 2237, Springer, 2001, pp. 212–226.
11. M. Denecker and E. Ternovska, *Inductive situation calculus.*, Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), AAAI Press, 2004, pp. 545–553.
12. ———, *A logic for non-monotone inductive definitions*, ACM Transactions on Computational Logic (2008), To appear.
13. D. East and M. Truszczyński, *Datalog with constraints*, Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), AAAI Press, 2000, pp. 163–168.
14. D. East and M. Truszczyński, *Predicate-calculus based logics for modeling and solving search problems*, ACM Transactions on Computational Logic **7** (2006), 38–83.
15. T. Eiter and M. Fink, *Uniform equivalence of logic programs under the stable model semantics*, Proceedings of the 19th International Conference on Logic Programming (ICLP 2003), LNCS, vol. 2916, Springer, 2003, pp. 224–238.
16. T. Eiter, M. Fink, H. Tompits, and S. Woltran, *Strong and uniform equivalence in answer-set programming: Characterizations and complexity results for the non-ground case.*, Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), 2005, pp. 695–700.
17. T. Eiter, M. Fink, and S. Woltran, *Semantical characterizations and complexity of equivalences in answer set programming*, ACM Transactions on Computational Logic (2006), To appear.
18. T. Eiter and G. Gottlob, *On the computational cost of disjunctive logic programming: propositional case*, Annals of Mathematics and Artificial Intelligence **15** (1995), no. 3-4, 289–323.
19. T. Eiter, H. Tompits, and S. Woltran, *On solution correspondences in answer-set programming*, Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Morgan Kaufmann, 2005, pp. 97–102.
20. I. Elkabani, E. Pontelli, and T. C. Son, *Smodels with CLP and its applications: a simple and effective approach to aggregates in ASP*, Proceedings of the 20th International Conference on Logic Programming (ICLP 2004), LNCS, vol. 3132, Springer, 2004, pp. 73–89.
21. W. Faber, G. Greco, and N. Leone, *Magic sets and their application to data integration*, Proceedings of the 10th International Conference on Database Theory (ICDT 2005), LNCS, vol. 3363, Springer, 2005, pp. 306–320.

22. W. Faber, N. Leone, and G. Pfeifer, *Recursive aggregates in disjunctive logic programs: semantics and complexity.*, Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004), LNAI, vol. 3229, Springer, 2004, pp. 200 – 212.
23. M. Fink, R. Pichler, H. Tompits, and S. Woltran, *Complexity of rule redundancy in non-ground answer-set programming over finite domains*, Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), LNAI, vol. 4483, Springer, 2007, pp. 123–135.
24. M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczyński, *The first answer set programming system competition*, Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007, LNAI, vol. 4483, Springer, 2007, pp. 3–17.
25. M. Gelfond, *Representing knowledge in A-Prolog*, Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II, LNCS, vol. 2408, Springer, 2002, pp. 413–451.
26. M. Gelfond and N. Leone, *Logic programming and knowledge representation – the A-prolog perspective*, Artificial Intelligence **138** (2002), 3–38.
27. M. Gelfond and V. Lifschitz, *The stable semantics for logic programs*, Proceedings of the 5th International Conference on Logic Programming (ICLP 1988), MIT Press, 1988, pp. 1070–1080.
28. ______, *Classical negation in logic programs and disjunctive databases*, New Generation Computing **9** (1991), 365–385.
29. A. Heyting, *Die formalen Regeln der intuitionistischen Logik*, Sitzungsberichte der Preussischen Akademie von Wissenschaften. Physikalisch-mathematische Klasse (1930), 42–56.
30. T. Janhunen, E. Oikarinen, H. Tompits, and S. Wotran, *Modularity aspects of disjunctive stable models*, Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), LNAI, vol. 4483, Springer, 2007, pp. 175–187.
31. R. Kowalski, *Predicate logic as a programming language*, Proceedings of the Congress of the International Federation for Information Processing (IFIP-1974) (Amsterdam), North Holland, 1974, pp. 569–574.
32. ______, *Logic for problem solving*, North Holland, Amsterdam, 1979.
33. V. Lifschitz, D. Pearce, and A. Valverde, *Strongly equivalent logic programs*, ACM Transactions on Computational Logic **2(4)** (2001), 526–541.
34. ______, *A characterization of strong equivalence for logic programs with variables*, Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007), LNAI, vol. 4483, Springer, 2007, pp. 188–200.
35. V. Lifschitz and H. Turner, *Splitting a logic program*, Proceedings of the 11th International Conference on Logic Programming (ICLP 1994), 1994, pp. 23–37.
36. F. Lin, *Reducing strong equivalence of logic programs to entailment in classical propositional logic*, Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR 2002), Morgan Kaufmann, 2002.
37. F. Lin and Y. Zhao, *ASSAT: Computing answer sets of a logic program by SAT solvers*, Proceedings of the 18th National Conference on Artificial Intelligence (AAAI 2002), AAAI Press, 2002, pp. 112–117.
38. L Liu, E. Pontelli, T. C. Son, and M. Truszczyński, *Logic programs with abstract constraint atoms: the role of computations*, Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), LNCS, Springer, 2007 (this volume).

39. L. Liu and M. Truszczyński, *Pbmodels - software to compute stable models by pseudoboolean solvers*, Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-05), LNAI, vol. 3662, Springer, 2005, pp. 410–415.
40. ______, *Properties of programs with monotone and convex constraints*, Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), AAAI Press, 2005, pp. 701–706.
41. ______, *Properties and applications of programs with monotone and convex constraints*, Journal of Artificial Intelligence Research **27** (2006), 299–334.
42. V.W. Marek and J.B. Remmel, *Set constraints in logic programming*, Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2004), vol. 2923, Springer, 2004, LNAI, pp. 167–179.
43. V.W. Marek and M. Truszczyński, *Stable models and an alternative logic programming paradigm*, The Logic Programming Paradigm: a 25-Year Perspective, Springer, Berlin, 1999, pp. 375–398.
44. ______, *Logic programs with abstract constraint atoms*, Proceedings of the 19th National Conference on Artificial Intelligence (AAAI 2004), AAAI Press, 2004, pp. 86–91.
45. W. Marek and M. Truszczyński, *Autoepistemic logic*, Journal of the ACM **38** (1991), no. 3, 588–619.
46. J. McCarthy and P. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence 4, Edinburgh University Press, 1969, pp. 463–502.
47. D.G. Mitchell and E. Ternovska, *A framework for representing and solving NP search problems*, Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005), AAAI Press, 2005, pp. 430–435.
48. I. Niemelä, *Logic programming with stable model semantics as a constraint programming paradigm*, Annals of Mathematics and Artificial Intelligence **25** (1999), no. 3-4, 241–273.
49. N. Pelov, *Semantics of logic programs with aggregates*, PhD Thesis. Department of Computer Science, K.U.Leuven, Leuven, Belgium (2004).
50. J.A. Robinson, *A machine-oriented logic based on resolution principle*, Journal of the ACM **12** (1965), 23–41.
51. P. Simons, I. Niemelä, and T. Soininen, *Extending and implementing the stable model semantics*, Artificial Intelligence **138** (2002), 181–234.
52. T. Son and E. Pontelli, *A constructive semantic characterization of aggregates in answer set programming*, Theory and Practice of Logic Programming (2007), Accepted (available at `http://arxiv.org/abs/cs.AI/0601051`).
53. T. Son, E. Pontelli, and P.H. Tu, *Answer sets for logic programs with arbitrary abstract constraint atoms*, Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006), AAAI Press, 2006, pp. 129–134.
54. M. Truszczyński, *Strong and uniform equivalence of nonmonotonic theories — an algebraic approach*, Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), AAAI Press, 2006, pp. 389–399.
55. M. Truszczyński, *The modal logic S4F, the default logic, and the logic here-and-there*, Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI 2007), AAAI Press, 2007.
56. H. Turner, *Strong equivalence made easy: nested expressions and weight constraints*, Theory and Practice of Logic Programming **3** (2003), 609–622.

57. M.H. van Emden and R.A. Kowalski, *The semantics of predicate logic as a programming language*, Journal of the ACM **23** (1976), no. 4, 733–742.
58. A. Van Gelder, K.A. Ross, and J.S. Schlipf, *The well-founded semantics for general logic programs.*, Journal of the ACM **38** (1991), no. 3, 620–650.
59. J. Vennekens and M. Denecker, *An algebraic account of modularity in ID-logic*, Proceedings of 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005), LNAI, vol. 3662, 2005, pp. 291–303.
60. J. Vennekens, D. Gilis, and M. Denecker, *Splitting an operator: an algebraic modularity result and its applications to logic programming*, Proceedings of the 20th International Conference on Logic Programming (ICLP 2004), 2004, pp. 195–209.