TLP **3** (4 & 5): 551–567, 2003. © 2003 Cambridge University Press
DOI: 10.1017/S1471068403001777 Printed in the United Kingdom

On the expressibility of stable logic programming

VICTOR W. MAREK
Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA
(*e-mail:* marek@cs.uky.edu)

JEFFREY B. REMMEL
Department of Mathematics, University of California, San Diego, La Jolla, CA 92093, USA
(*e-mail:* jremmel@ucsd.edu)

Abstract

Schlipf (1995) proved that Stable Logic Programming (SLP) solves all NP decision problems. We extend Schlipf's result to prove that SLP solves all search problems in the class NP. Moreover, we do this in a uniform way as defined in Marek and Truszczyński (1991). Specifically, we show that there is a single DATALOG$^{\neg}$ program P_{Trg} such that given any Turing machine M, any polynomial p with non-negative integer coefficients and any input σ of size n over a fixed alphabet Σ, there is an extensional database $edb_{M,p,\sigma}$ such that there is a one-to-one correspondence between the stable models of $edb_{M,p,\sigma} \cup P_{Trg}$ and the accepting computations of the machine M that reach the final state in at most $p(n)$ steps. Moreover, $edb_{M,p,\sigma}$ can be computed in polynomial time from p, σ and the description of M and the decoding of such accepting computations from its corresponding stable model of $edb_{M,p,\sigma} \cup P_{Trg}$ can be computed in linear time. A similar statement holds for Default Logic with respect to Σ_2^{P}-search problems.[1]

KEYWORDS: Answer Set Programming, Turing machines, expressibility

1 Introduction

The main motivation for this paper comes from recent developments in Knowledge Representation, especially the appearance of a new generation of systems (Cholewiński *et al.* 1996; Niemelä and Simons 1996; Eiter *et al.* 1997) based on the so-called Answer Set Programming (ASP) paradigm (Niemelä 1998; Cadoli and Palipoli 1998; Marek and Truszczyński 1999; Lifschitz 1999). The emergence of these ASP systems suggest that we need to revisit one of the basic issues in the foundations of ASP, namely, how can we characterize what such ASP systems can theoretically compute.

Throughout this paper, we shall focus mostly on one particular ASP formalism, specifically, the Stable Semantics for Logic Programs (SLP) (Gelfond and Lifschitz

[1] The proof of this result involves additional technical complications and will be a subject of another publication.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

1988). We note that the underlying methods of ASP are similar to those used in logic programming (Apt 1990) and Constraint Programming (Jaffar and Maher 1994; Marriott and Stuckey 1998). That is, like Logic Programming, ASP is a declarative formalism and the semantics of all ASP systems are based on logic. Like constraint programming, certain clauses of an ASP program act as *constraints*. There is a fundamental difference between ASP programs and constraint logic programs, however. That is, in constraint programming, the constraints act on individual elements of Herbrand base of the program while the constraint clauses in ASP programs act more globally in that they place restrictions on what subsets of the Herbrand base can be acceptable answers for the program. For example, suppose that we have a problem Π whose solutions are *subsets* of some Herbrand base H. In order to solve the problem, an ASP programmer essentially writes a logic program P that describes the constraints on the subsets of H which can be answers to Π. The basic idea is that the program P should have the property that there is an easy decoding of solutions of Π from stable models of P and that all solutions of Π can be obtained from stable models of P through this decoding. The program P is then submitted to the ASP engine such as *smodels* (Niemelä and Simons 1996), `dlv` (Eiter *et al.* 1997) or DeReS (Cholewiński *et al.* 1996), which computes the stable models of the program P. Thus the ASP engine finds the stable models of the program (if any exists) and one reads off the solutions to Π from these stable models. Notice that the idea here is that all solutions are equally good in the sense that any solution found in the process described above is acceptable. Currently, the systems based on ASP paradigm are being tested on the problems related to planning (Lifschitz 1999; Niemelä 1998), product configuration (Soininen and Niemelä 1999), combinatorial optimization problems (Cholewiński *et al.* 1999; Niemelä 1998) and other domains.

It is a well-known fact that the semantics of existing logic programming systems such as Prolog, Mercury and LDL have serious problems, principally due to necessary compromises in the implementations. For instance, the unification algorithms used by most dialects of Prolog do not enforce the occurs check and hence these systems can produce incorrect results (Apt and Pellegrini 1994). Moreover, the processing strategies of Prolog and similar languages have the effect that correct logic programs can be non-terminating (Apt and Pedreschi 1994). While good programming techniques can overcome these problems, it is clear that such deficiencies have restricted the appeal of the Logic Programming systems for ordinary programmers and system analysts. The promise of ASP and, in particular, of SLP and its extensions, such as Disjunctive Logic Programming (Gelfond and Lifschitz 1991; Eiter *et al.* 1997), is that a new generation of logic programming systems can be built which have a clear semantics and are easier to program than the previous generation of logic programming systems. In particular, both of the problems referred to above, namely, the occurs check problem and the termination problem, do not exist in SLP. Of course, there is a price to pay, namely, SLP systems only accept programs without function symbols. Consequently, one of the basic data structures used in Prolog, the *term*, is not available in SLP. Thus SLP systems require the programmer to explicitly construct many data structures. In SLP programming, predicates are used to construct the required data structures and clauses that serve as constraints are

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

used to ensure that the predicates behave properly with respect to semantics of the program. SLP programs are always terminating because the Herbrand base is finite, and hence there are only a finite number of stable models. In addition, unlike the case of usual logic programming, the order of the clauses of the program does not affect the set of stable models of the program.[2] Finally, the stable semantics of logic programs is well understood so that SLP programs have clear semantics.

The restriction that ASP programs do not contain function symbols is crucial. First, it is well known that once one allows function symbols in a logic program P, the Herbrand base becomes infinite. Moreover, the stable models of logic programs *with* function symbols can be immensely complex. For example, for stratified logic programs (Apt *et al.* 1988; Przymusiński 1988), the perfect model is the unique stable model of that program (Gelfond and Lifschitz 1988). Apt and Blair (Apt and Blair 1990) showed that perfect models of stratified logic programs capture precisely the arithmetic sets. That is, they show that for a given arithmetic set X of natural numbers, there is a finite stratified logic program P_X such that in the perfect model of P_X, some predicate p_X is satisfied by precisely the numbers in X. This was the first result that showed that it is not possible to have meaningful practical programming with general stratified programs *if we allow* unlimited use of function symbols. The result of Apt and Blair (1990) was extended in Blair *et al.* (1995), where it was shown that the set of stable models of a locally stratified program can capture any set in the hyperarithmetic hierarchy. Marek *et al.* (1994) showed that the problem of finding a stable model of a finite (predicate) logic program P is essentially equivalent to finding a path through an infinite branching recursive tree. That is, given an infinite branching recursive tree $T \subseteq \omega^{<\omega}$, there is a finite program P_T such that there is a one-to-one degree-preserving correspondence between the infinite paths through T and the stable models of P_T and, vice versa, given a finite program P, there is a recursive tree T_P such that there is one-to-one degree preserving correspondence between the stable models of P and the infinite paths through T_P. One consequence of this result is that the problem of determining whether a finite predicate logic program has a stable model is Σ^1_1-complete. More results on the structure of the family of stable models of programs can be found in Cenzer and Remmel (1999).

All the results mentioned in the previous paragraph show that the stable model semantics for logic programs admitting function symbols can be used practically, only in a very limited setting. XSB system attempts to do deal with this problem by computing only the well-founded semantics. When the well-founded semantics is total, the resulting model is the unique stable model of the program. Unfortunately, the class of programs for which it succeeds is not intuitive (Rao *et al.* 1997). Yet another attempt to return the power of function symbols to the language has been made in Bonatti (2001). The class of programs considered in Bonatti (2001) allows one to express recursively enumerable sets, but not more complex sets so that, at best, one could get a more compact representation of problems solved with ordinary Prolog.

[2] However it is the case that the order of clauses can affect the processing time of the ASP engine.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

As stated above, ASP systems propose a more radical solution to the problem of complexity of stable models of logic programs with function symbols, namely, abandoning function symbols entirely. Once this is accepted, the semantics of a logic program P can be defined in two stages. First, we assume, as in standard Logic Programming, that we interpret P over the Herbrand universe of P determined by the predicates and constants that occur in P. Since the set of constants occurring in the program is finite, we can ground the program in these constants to obtain a finite propositional logic program P_g. The stable models of P are by definition the stable models of P_g. The process of grounding is performed by a separate grounding engine such as *lparse* (Niemelä and Simons 1996). The grounded program P_g is then passed to an engine that computes stable models of propositional logic programs. It is then easy to check that the features of SLP mentioned above, i.e. the absence of occurs check and termination problems and the independence of the semantics from the ordering of the clauses of the program, automatically hold. That is, since grounding uses only very limited part of unification, the occurs check problem is eliminated. The space of candidates for stable models is finite and so there is no termination problem. Finally, the stable semantics of propositional programs does not depend on the order of clauses.

The language of logic programming without function symbols was studied by the database community with the hope that it could lead to new, more powerful, database language (Ullman 1988). This language is called DATALOG$^\neg$ and some database systems such as DB2 implement the positive part of DATALOG$^\neg$. The fact that admitting negation in the bodies of clauses leads to multiple stable models was unacceptable from the database perspective. Hence the database community preferred other semantics for DATALOG$^\neg$ programs such as the well-founded semantics (Van Gelder *et al.* 1991) or the inflationary semantics (Abiteboul *et al.* 1995).

The main purpose of this paper is to revisit the question of what can be computed by logic programs without functions symbols under the stable model semantics. First, consider the case of finite propositional programs. Here the situation is simple. Given a set At of propositional atoms, let $\mathscr{F}$ be a finite antichain of subsets of At, i.e. whenever $X, Y \in \mathscr{F}$, $X \subseteq Y$, then $X = Y$. Then one can show that there is a logic program $P_{\mathscr{F}}$ such that $\mathscr{F}$ is precisely the class of all stable models of $P_{\mathscr{F}}$ (Marek and Truszczyński 1993). Moreover, the family of stable models of any program P forms such an antichain. Thus in the case of finite propositional logic programs, we have a complete characterization of the possible sets of stable models. However, this result by itself does not tell us anything about the uniformity and the effectiveness of the construction. The basic complexity result for SLP propositional programs is due to Marek and Truszczyński (1991), who showed that the problem of deciding whether a finite propositional logic program has a stable model is NP-complete. For DATALOG$^\neg$, an analogous result has been obtained in Schlipf (1995).

To precisely formulate our question about what can be computed by logic programs without functions symbols under the stable model semantics, we first need to recall the notion of *search problem* (Garey and Johnson 1979) and of a *uniform* logic program (Marek and Truszczyński 1999). A search problem is a set $\mathscr{S}$

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

of finite instances such that, given any instance $I \in \mathscr{S}$, there is a set S_I of solutions to $\mathscr{S}$ for the instance I. It is possible that for some instances I, S_I is the empty set. For example, the search problem may be to find Hamiltonian paths in a graph. Thus, the set of instances of the problem is the set of all finite graphs. Then, given any instance, i.e. a graph G, S_G is the set of all Hamiltonian paths of G. We say that an algorithm solves a search problem $\mathscr{S}$ if it returns a solution $s \in S_I$ whenever S_I is non-empty and it returns the string "empty" otherwise. We say that a search problem $\mathscr{S}$ *is in* NP if there is such an algorithm which can be computed by a non-deterministic polynomial time Turing machine. We say that search problem $\mathscr{S}$ *is solved by a uniform logic program* if there exists a single logic program $P_{\mathscr{S}}$, a polynomial time extensional data base transformation function $edb_{\mathscr{S}}$ and a polynomial time solution decoding function $sol_{\mathscr{S}}(\cdot,\cdot)$ such that for every instance I in $\mathscr{S}$,

1. $edb_{\mathscr{S}}(I)$ is a finite set of facts, i.e. clauses with empty bodies and no variables,
2. whenever $sol_{\mathscr{S}}(I)$ is non-empty, $sol_{\mathscr{S}}(I,\cdot)$ maps the set of stable models of the $edb_{\mathscr{S}}(I) \cup P_{\mathscr{S}}$ onto the set of solutions S_I of I and
3. if $sol_{\mathscr{S}}(I)$ is empty, then $edb_{\mathscr{S}}(I) \cup P_{\mathscr{S}}$ has no stable models.

We note that decision problems can be viewed as special cases of search problems. Schlipf (1995) has shown that the class of *decision* problems in NP is captured precisely by uniform logic programs. Specifically he proved that a decision problem is solved by a uniform logic program if and only if it is in NP. An excellent review of the complexity and expressibility results for Logic Programming can be found in Dantsin *et al.* (2001). The goal of this paper is to prove a strengthening of Schlipf's result as well as prove a number of related facts. We will prove that Schlipf's result can be extended to all NP *search* problems. That is, we shall show that there is a single logic program P_{Trg} that is capable of simulating polynomial time nondeterministic Turing machines in the sense that given any polynomial time nondeterministic Turing machine M, any input σ, and any run-time polynomial $p(x)$, there is a set of facts $edb_{M,p,\sigma}$ such that a stable model of $P_{Trg} \cup edb_{M,p,\sigma}$ codes an accepting computation of M started with input σ that terminates in $p(|\sigma|)$ or fewer steps and any such accepting computation of M is coded by some stable model of $P_{Trg} \cup edb_{M,p,\sigma}$. This result will show that logic programs without function symbols under the stable model semantics capture all NP-search problems.[3] The converse implication, that is, a search problem computed by a uniform logic program P is an NP-search problem is obvious since one can compute a stable model M of a program by first guessing M and then doing a polynomial time check to verify that M is a stable model of the program.

[3] As pointed by M. Truszczyński, for our goal of describing the complexity of the Stable Logic Programming, a weaker result is sufficient. That is, we need only show that for each instance I of an NP search problem Π, there is a program P_I and a polynomial time projection from the collection of stable models of P_I to the set of solutions of I. Our result shows that this property holds in a stronger form. Namely, there is a single program with a varying extensional database.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

2 Technical preliminaries

In this section we shall formally introduce several notions that will be needed for the proof of our main result. The proof of our main result uses the basic idea used by Cook (1971) in his proof of the NP-completeness of the satisfiability problem. First, we introduce the set of logic programs that we will study. We will consider here only so called DATALOG$^{\neg}$ programs. Specifically, a clause is an expression of the form

$$p(\overline{X}) \leftarrow q_1(\overline{X}), \ldots, q_m(\overline{X}), \neg\, r_1(\overline{X}), \ldots, \neg\, r_n(\overline{X}) \tag{1}$$

where $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are atoms, possibly with variables and/or constants. Here we abuse notation by writing $p(\overline{X})$ to mean that the variables that occur in the predicate p are contained in $\overline{X}$. A program is a finite set P of clauses of the form (1). We assume that the underlying language $\mathscr{L}_P$ of any given program P is determined by the constants and predicate symbols that occur in the program. Thus the Herbrand universe U_P of P is just the set of all constant terms occurring in P and the Herbrand base H_P of P is the set of all ground atoms of the language $\mathscr{L}_P$. Since there are no function symbols in our programs, both the Herbrand universe and the Herbrand base of the program are finite.

A ground instance of a clause C of the form (1) is the result of a simultaneous substitution of constants $\overline{c}$ for variables $\overline{X}$ occurring in C. Given a program P, P_g is the propositional program consisting of all ground substitutions of the clauses of P. Given a propositional program P and a set M included in its Herbrand base, H_P, the Gelfond-Lifschitz transformation of P relative to M is the program $GL(P, M)$ arising from P as follows. First, eliminate all clauses C in P such that for some j, $1 \leqslant j \leqslant n$, $r_j(\overline{c}) \in M$. Next, in any remaining clause, eliminate all negated atoms. The resulting set of clauses forms a program, $GL(P, M)$, which is a Horn program and hence it possesses a least model N_M. We say that M is a *stable model of the propositional program* P if $M = N_M$. Finally, given any program P with variables, we say that M is a stable model of a program P if M is a stable model of the propositional program P_g.

A nondeterministic Turing Machine is a septuple of the form

$$M = (Q, \Sigma, \Gamma, D, \delta, s_0, f).$$

Here Q is a finite set of states and Σ is a finite alphabet of input symbols. We assume Q always contains two special states, s_0, the start state, and f, the final state. We also assume that there is a special symbol B for "blank" such that $B \notin \Sigma$. The set $\Gamma = \Sigma \cup \{B\}$ is the set of tape symbols. The set D is the set of move directions consisting of the elements l, r, and λ where l is the "move left" symbol, r is the "move right" symbol and λ is the "stay put" symbol. The function $\delta : Q \times \Gamma \to \mathscr{P}(Q \times \Gamma \times D)$ is the transition function of the machine M. Here $\mathscr{P}(Q \times \Gamma \times D)$ denotes the power set of the set $Q \times \Gamma \times D$. We can also think of δ as a 5-ary relation. Thus we can represent the transition function of the machine M as a collection of atoms describing 5-tuples. We assume M operates on a one-way infinite tape where the cells of the tape are labeled from left to right by $0, 1, 2, \ldots$. To visualize the behavior

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

of the machine M, we shall talk about the read-write head of the machine. At any given time in a computation, the read-write head of M is always in some state $s \in Q$ and is reading some symbol $p \in \Gamma$ which is in a cell c of the tape. It then picks an instruction $(s1, p1, d) \in \delta(s, p)$ and then replaces the symbol p by $p1$, changes its state to state $s1$, and moves according to d. We assume that at the start of the computation of M on input $\sigma = (\sigma(0), \ldots, \sigma(n-1))$, cells $0, \ldots, n-1$ contain the symbols $\sigma(0), \ldots, \sigma(n-1)$ respectively and all cells to the right of cell $n-1$ are blank. We do not impose (as it is often done) any special restrictions on the state of the tape and the position of the head at the end of computation. However, we assume that at the start of any computation, the read-write head is in state s_0 and is reading the symbol in cell 0.

Suppose we are given a Turing machine M whose runtimes are bounded by a polynomial $p(x) = a_0 + a_1 x + \cdots + a_k x^k$ where each $a_i \in N = \{0, 1, 2, \ldots\}$ and $a_k \neq 0$. That is, on any input of size n, an accepting computation terminates in at most $p(n)$ steps. Then any accepting computation on input σ can affect at most the first $p(n)$ cells of the tape. Thus in such a situation, there is no loss in only considering tapes of length $p(n)$. Hence in what follows, one shall implicitly assume that the tape is finite. Moreover, it will be convenient to modify the standard operation of M in the following ways:

1. We shall assume $\delta(f, a) = \{(f, a, \lambda)\}$ for all $a \in \Gamma$.
2. Given an input x of length n, instead of immediately halting when we first get to the final state f reading a symbol a, we just keep executing the instruction (f, a, λ) until we have completed $p(n)$ steps. That is, we remain in state f, we never move, and we never change any symbols on the tape after we get to state f.

The main effect of these modifications is that all accepting computations will run for exactly $p(n)$ steps on an input of size n.

3 Uniform coding of Turing Machines by a logic program

In this section, we describe the logic program P_{Trg} and our extensional data base function $edb_{M,p,\sigma}$ described above. The key to our construction is the fact that at any given moment of time, the behavior of a Turing machine M depends only on the current state of tape, the position of the read-write head and the set of available instructions. Our coding of Turing machine computation reflects this simple observation. First, we define the language (i.e. a signature) of the program P_{Trg}. The set of predicates that will occur in our extensional database are the following:

time(X) for "X is a time step",
cell(X) for "X is a cell number",
symb(X) for "X is a symbol",
state(S) for "S is a state",
i_position(P) for "P is the initial position of the read-write head",
data(P, Q) for "Initially, the tape stores the symbol Q at the cell P",

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

$delta(X, Y, X1, Y1, Z)$ for "the triple $(X1, Y1, Z)$ is an executable instruction when the read-write head is in state X and is reading the symbol Y" (thus *delta* represents the five-place relation δ),
$neq(X, Y)$ for "X is different from Y"
$eq(X, Y)$ for "X is equal to Y",[4]
$succ(X, Y)$ for "Y is equal to $X + 1$"[5]

Fix a nondeterministic Turing machine $M = (Q, \Sigma, \Gamma, D, \delta, s_0, f)$, a run-time polynomial $p(x)$ and an input $\sigma = (\sigma(0), \ldots, \sigma(n-1))$ of length n. This given, we now define the extensional database $ext_{M,p,\sigma}$. First, $ext_{M,p,\sigma}$ will contain the following the following set of constant symbols:

(1) $0, 1, \ldots, p(n)$,
(2) s, for each $s \in S$ (Note two constants s_0 (for initial state), and f (for final state) will be present in every extensional database),
(3) B (blank symbol) and x for each $x \in \Sigma$, and
(4) r, l, λ.

We let $edb_{M,\sigma,p}$ consist of the following set of facts that describe the machine M, the segment of integers $0, \ldots, p(n)$ and the initial configuration σ of the tape.

1. For each $s \in Q$, the clause $state(s) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
2. For each $x \in \Gamma$, the clause $symb(x) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
3. For every pair $(s, x) \in Q \times \Gamma$ and every triple $(s1, x1, d) \in \delta(s, x)$, the clause $delta(s, x, s1, x1, d) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
4. For $0 \leqslant i < p(n)$, the clause $succ(i, i+1) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
5. For $0 \leqslant i \leqslant p(n)$, the clause $time(i) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
6. For $0 \leqslant i \leqslant p(n) - 1$, the clause $cell(i) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
7. For $0 \leqslant m \leqslant |\sigma| - 1$, the clause $data(m, \sigma(m)) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
8. For $|\sigma| \leqslant m \leqslant p(n) - 1$, the clause $data(m, B) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
9. The clauses $dir(l) \leftarrow$, $dir(r) \leftarrow$ and $dir(\lambda) \leftarrow$ belong to $ext_{M,p,\sigma}$.
10. The clause $i_position(0) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
11. For all $a, b \in S \cup \Gamma \cup \{0, \ldots, p(n)\}$ with $a \neq b$, the clause $neq(a, b) \leftarrow$ belongs to $ext_{M,p,\sigma}$.
12. For all $a \in S \cup \Gamma \cup \{0, \ldots, p(n)\}$, the clause $eq(a, a) \leftarrow$ belongs to $ext_{M,p,\sigma}$.

The remaining predicates of P_{Trg} are the following:

$tape(P, Q, T)$ for "the tape stores symbol Q at cell P at time T",
$position(P, T)$ for "the read-write head reads the content cell P at time T",
$state(S, T)$ for "the read-write head is in state S at time T" (notice that we have both a unary predicate $state/1$ with the content consisting of states, and $state/2$ to describe the evolution of the machine),

[4] Technically, we should use a separate equality and inequality relation for each type, but we will not do so.
[5] For the clarity of presentation we will use equality symbol $=$, inequality symbol, $\neq$ and relation described by the successor function $+1$, instead of eq, neq, and $succ$.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

$instr(S, Q, S1, Q1, D, T)$ for "instruction $(S1, Q1, D)$ belonging to $\delta(S, Q)$ has been selected for execution at time T",
$otherInstr(S, Q, S1, Q1, D, T)$ for "instruction other than $(S1, Q1, D)$ belonging to $\delta(S, Q)$ has been selected for execution at time T",
$instr_def(T)$ for "there is an instruction to be executed at time T",
completion for "computation successfully completed", and
A, a propositional letter.[6]

In the program P_{Trg}, there should be no constants. For notational convenience, we will not be absolutely strict in this respect. That is, to simplify our presentation, we will use the constants 0, f, and s_0 in P_{Trg}. These can easily be eliminated by introducing appropriate unary predicates. Finally, to simplify the clauses, we will follow here the notation used in the *smodels* syntax. That is, we use $p(X_1; \ldots; X_k)$ as an abbreviation for $p(X_1), \ldots, p(X_k)$. This given, we are now ready to write the program P_{Trg}.

Group 1. Our first four clauses are used to describe the position of the read-write head at any given time t.

(1.1) (Initial position of the read-write head)
$position(P, T) \leftarrow time(T), cell(P), eq(T, 0), i_position(P)$

(1.2) $position(P1, T1) \leftarrow time(T; T1), cell(P; P1), state(S; S1), dir(D), symb(Q; Q1), T1 = T + 1, P1 + 1 = P, position(P, T), state(S, T), tape(P, Q, T), instr(S, Q, S1, Q1, D, T), D = l, P \neq 0$

(1.3) $position(P1, T1) \leftarrow time(T; T1), cell(P; P1), state(S; S1), dir(D), symb(Q; Q1), T1 = T + 1, P1 = P + 1, position(P, T), state(S, T), tape(P, Q, T), instr(S, Q, S1, Q1, D, T), D = r, P \neq p(n) - 1$

(1.4) $position(P1, T1) \leftarrow time(T; T1), cell(P; P1), state(S; S1), symb(Q; Q1), dir(D), T1 = T + 1, P = P1, position(P, T), state(S, T), tape(P, Q, T), instr(S, Q, S1, Q1, D, T), D = \lambda$

Group 2. Our next three clauses describe how the contents of the tape change as instructions get executed.

(2.1) $tape(P, Q, T) \leftarrow time(T), cell(P), symb(Q), T = 0, data(P, Q)$

(2.2) $tape(P, Q1, T1) \leftarrow time(T; T1), cell(P), state(S; S1), symb(Q; Q1), dir(D), T1 = T + 1, position(P, T), state(S, T), tape(P, Q, T), instr(S, Q, S1, Q1, D, T)$

(2.3) $tape(P, Q, T1) \leftarrow time(T; T1), cell(P; P1), symb(Q), T1 = T + 1, tape(P, Q, T), position(P1, T), P \neq P1$

Group 3. Our next two clauses describe how the state of the read-write head evolves in time.

(3.1) $state(S, T) \leftarrow time(T), state(S), T = 0, S = s_0$

[6] The propositional letter A will be used whenever we write clauses acting as constraints. That is, the symbol A will occur in the following syntactical configuration. A will be the head of some clause, and the negation of A will also occur in the body of that same clause. In such situation a stable model *cannot* satisfy the remaining atoms in the body of that clause.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

(3.2) $state(S1, T1) \leftarrow time(T; T1), cell(P), symb(Q; Q1), state(S; S1), dir(D), T1 = T + 1, position(P, T), state(S, T), tape(P, Q, T), instr(S, Q, S1, Q1, D, T)$

Group 4. Our next two clauses describe how we select a unique instruction to be executed at time T.

(4.1) Selecting instruction at step 0.
$instr(S, Q, S1, Q1, D, T) \leftarrow state(S; S1), symb(Q; Q1), dir(D), time(T), T = 0, S = s_0, i_position(P), tape(P, Q, T), delta(S, Q, S1, Q1, D), \neg otherInstr(S, Q, S1, Q1, D, T)$

(4.2) Selecting instruction at other steps.
$instr(S, Q, S1, Q1, D, T) \leftarrow state(S; S1), symb(Q; Q1), dir(D), time(T), T \neq 0, position(P, T), state(S, T), tape(P, Q, T), delta(S, Q, S1, Q1, D), \neg otherInstr(S, Q, S1, Q1, D, T)$

Group 5. Our next set of clauses defines the *otherInstr* predicate. Here clauses (5.6) and (5.7) are designed to ensure that exactly one instruction is selected for execution at any given time T.

(5.1) $otherInstr(S, Q, S1, Q1, D1, T) \leftarrow state(S; S'; S1; S2), symb(Q; Q'; Q1; Q2), time(T), dir(D1; D2), instr(S', Q', S2, Q2, D2, T), S2 \neq S1$

(5.2) $otherInstr(S, Q, S1, Q1, D1, T) \leftarrow state(S; S'; S1; S2), symb(Q; Q'; Q1; Q2), time(T), dir(D1; D2), instr(S', Q', S2, Q2, D2, T), Q2 \neq Q1$

(5.3) $otherInstr(S, Q, S1, Q1, D1, T) \leftarrow state(S; S'; S1; S2), symb(Q; Q'; Q1; Q2), time(T), dir(D1; D2), instr(S', Q', S2, Q2, D2, T), D2 \neq D1$

(5.4) $otherInstr(S, Q, S1, Q1, D1, T) \leftarrow state(S; S'; S1; S2), symb(Q; Q'; Q1; Q2), time(T), dir(D1; D2), instr(S', Q', S2, Q2, D2, T), S' \neq S$

(5.5) $otherInstr(S, Q, S1, Q1, D1, T) \leftarrow state(S; S'; S1; S2), symb(Q; Q'; Q1; Q2), time(T), dir(D1; D2), instr(S', Q', S2, Q2, D2, T), Q' \neq Q$

(5.6) The definition of the *instr_def* predicate.
$instr_def(T) \leftarrow state(S; S1), symb(Q; Q1), dir(D), time(T), instr(S, Q, S1, Q1, D, T)$

(5.7) The clause to ensure that there is an instruction to be executed at any given time.
$A \leftarrow time(T), \neg instr_def(T), \neg A$

Group 6. Constraints for the coherence of the computation process.

(6.1) When the task is completed.
$completion \leftarrow symb(Q), instr(f, Q, f, Q, \lambda, p(n))$.

(6.2) The atom *completion* belongs to every stable model.
$A \leftarrow \neg\, completion, \neg A$

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

Notice that the program P_{Trg} is *domain-restricted* (Syrjänen 2001), that is, every variable in the body of a clause is bound by a positive occurrence of an extensional database predicate. This restriction does not limit the expressive power of such programs, but greatly reduces the work of the grounding engine (Niemelä and Simons 1997).

4 Main results

Our first proposition immediately follows from our construction.

Proposition 1
There is a polynomial q so that for every machine M, polynomial p, and an input σ, the size of the extensional database $edb_{M,p,\sigma}$ is less than or equal to $q(|M|, |\sigma|, p(|\sigma|))$.

We shall now prove that for any nondeterministic Turing Machine M, runtime polynomial $p(x)$, and input σ of length n, the stable models of $edb_{M,p,\sigma} \cup P_{Trg}$ encode the sequences of tapes of length $p(n)$ which occur in the steps of an accepting computation of M starting on σ and that any such sequence of steps can be used to produce a stable model of $edb_{M,p,\sigma} \cup P_{Trg}$.

Theorem 1
The mapping of Turing machines to DATALOG$^{\neg}$ programs defined by $\langle M, \sigma, p\rangle \mapsto edb_{M,p,\sigma} \cup P_{Trg}$ has the property that there is a 1-1 polynomial time correspondence between the set of stable models of $edb_{M,p,\sigma} \cup P_{Trg}$ and the set of computations of M of the length $p(n)$, starting on the tape corresponding to the input σ, and ending in the state f.

Proof
We first need to see what is a valid run of a machine M that ends in the state f. To this end let us define an *instruction* of the machine M as a quintuple $\langle q, a, q_1, a_1, d\rangle$ such that $(q_1, a_1, d) \in \delta(q, a)$. A *state of tape* is a sequence S of symbols of the length $p(n)$ from alphabet $\Sigma \cup \{B\}$. A *configuration* is a triple $\langle i, S, k\rangle$ where i is an instruction $\langle q, a, q_1, a_1, d\rangle$, S is a state of tape, and k is an integer $\leqslant p(n)$ and such that $S(k) = a \in \Sigma \cup \{B\}$. Informally, k is the index of the cell on which the read-write head is pointing at the time the configuration is observed and $a = S(k)$ is the content of that cell. The *coherence* condition $S(k) = a$ says that the instruction i is applicable at this moment. A *one step-transition* is a pair of configurations

$$\langle\langle i, S, k\rangle, \langle j, T, m\rangle\rangle$$

where $i = \langle q, a, q_1, a_1, d\rangle$ and $j = \langle r, b, r_1, b_1, e\rangle$ satisfy the following:

1. $r = q_1$ (i.e. in the transition we moved to the next state as required by i),
2. $m = \begin{cases} k-1 & \text{if } k \neq 0, d = l \\ k+1 & \text{if } k \neq p(n)-1, d = r \\ k & \text{if } d = \lambda \end{cases}$
   (i.e. the read-write head moved as required by the instruction i),

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

3. $T(n) = \begin{cases} a_1 & \text{if } n = k \\ S(n) & \text{otherwise} \end{cases}$
   (i.e. the state of tape has been altered in just one place, namely k and a_1 has been put there), and
4. $b = T(m)$. (i.e. the instruction j is coherent with the cell observed by the read-write head).

We write $C \vdash D$ when C and D are configurations and $\langle C, D\rangle$ is a one-step transition. A *run* of a machine M is a sequence of configurations $\langle C_0, \ldots, C_{p(n)}\rangle$ such that

1. $C_0 = \langle\langle s_0, a, t, a_1, d\rangle, S, 0\rangle$ (that is the machine M is in the start state s_0, $a = S(0)$ is the content of the cell 0, and the read-write head points to cell 0) and
2. for all $0 \leqslant k < p(n)$, $C_k \vdash C_{k+1}$.

A *valid run* of the machine M is a run where $C_{p(n)} = \langle i, S, k\rangle$, $i = \langle f, a, f, a, \lambda\rangle$. Thus a valid run of M is a run where the last state of the machine is f.

For the rest of this proof, we shall only consider valid runs $\langle C_0, \ldots, C_{p(n)}\rangle$ of M such that $C_0 = \langle i, S, 0\rangle$ where $S(i) = \sigma(i)$ for $i \leqslant n-1$ and $S(i) = B$ for $i > n$. That is, we shall only consider valid runs of M which start on an input $\sigma = (\sigma(0), \ldots, \sigma(n-1))$ of length n. We will show that each such valid run determines a unique stable model of $edb_{M,p,\sigma} \cup P_{Trg}$ and conversely every stable model of the program $edb_{M,p,\sigma} \cup P_{Trg}$ determines such a valid run of M. First, given a valid run $\mathscr{C} = \langle C_0, \ldots, C_{p(n)}\rangle$ of the machine M, where for m, $0 \leqslant m \leqslant p(n)$,

$$C_m = \langle i_m, S_m, k_m\rangle$$

we define the set of atoms $N_{\mathscr{C}}$ which consists of the union of sets of atoms $N_1 \cup \ldots \cup N_7$ where:

$$\begin{aligned}
N_1 &= edb_{M,p,\sigma} \\
N_2 &= \{position(m, k_m) : 0 \leqslant m \leqslant p(n)\} \\
N_3 &= \{tape(r, S_m(r), m) : 0 \leqslant m \leqslant p(n),\ 0 \leqslant r \leqslant p(n) - 1\} \\
N_4 &= \{instr(q, a, q', a', d, m) : i_m = \langle q, a, q', a', d\rangle, 0 \leqslant m \leqslant p(n)\} \\
N_5 &= \{otherInstr(q'', a'', q''', a''', d''', m) : \langle q''', a''', d'''\rangle \in \delta(q'', a''), \\
&\qquad i_m = \langle q, a, q', a', d\rangle, \langle q'', a'', q''', a''', d'''\rangle \neq \langle q, a, q', a', d\rangle, 0 \leqslant m \leqslant p(n)\} \\
N_6 &= \{instr_def(m) : 0 \leqslant m \leqslant p(n)\} \\
N_7 &= \{completion\}
\end{aligned}$$

We show that $N_{\mathscr{C}}$ is a stable model of $edb_{M,p,\sigma} \cup P_{Trg}$. Indeed, after we ground $edb_{M,p,\sigma} \cup P_{Trg}$ and reduce it with respect to $N_{\mathscr{C}}$, it is straightforward to prove by simultaneous induction on m that the least model of the reduct contains

(a) for each m, $0 \leqslant m \leqslant p(n)$ exactly one atom of the form $instr(q, a, q', a', d, m)$ and that this atom belongs to $N_{\mathscr{C}}$,
(b) for each m, $0 \leqslant m \leqslant p(n)$ all atoms $otherInstr(q'', a'', q''', a''', d''', m)$ with $\langle q''', a''', d'''\rangle \in \delta(q'', a'')$, where $i_m = \langle q, a, q', a', d\rangle$, and

$$\langle q'', a'', q''', a''', d'''\rangle \neq \langle q, a, q, a', d\rangle,$$

and that these atoms belong to $N_{\mathscr{C}}$,

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

(c) for each m and r, $0 \leqslant m \leqslant p(n)$, $0 \leqslant r \leqslant p(n)-1$ exactly one atom of the form $tape(r, x, m)$ that these atoms belong to $N_{\mathscr{C}}$

(d) for each m, $0 \leqslant m \leqslant p(n)$ exactly one atom of the form $position(m, k)$ and that $k = k_m$,

(e) for each m, $0 \leqslant m \leqslant p(n)$ exactly one atom of the form $instr_def(m)$, and

(f) the atom *completion*.

Thus $N_{\mathscr{C}}$ is a stable model of $edb_{M,p,\sigma} \cup P_{Trg}$. Moreover, it is clear that the mapping $\mathscr{C} \mapsto N_{\mathscr{C}}$ is an injection since two different valid runs $\mathscr{C}$ and $\mathscr{C}'$ differ in some least place m and, hence, the atoms of group N_4 involving the relational symbol *instr* at time m must differ in $N_{\mathscr{C}}$ and $N_{\mathscr{C}'}$.

Conversely, suppose that N is a stable model of $edb_{M,p,\sigma} \cup P_{Trg}$. First observe that $A \notin N$. That is, all the clauses that have A in the head also have $\neg A$ in the body. Thus if $A \in N$, then there are no clauses with head A in $GL(edb_{M,p,\sigma} \cup P_{Trg}, N)$ so that N could not be a stable model. Since, A is not in N, then it is easy to to see that clause (6.2) forces N to contain the atom *completion*. Since the only way to derive the atom *completion* is via clause (6.1), it follows that N must contain the atom $instr(f, q, f, q, \lambda, p(n))$ for some symbol $q \in \Gamma$.

Similarly, by clause (5.7), for every $0 \leqslant t \leqslant p(n)$, it must be the case that $instr_def(t)$ must be in N. Since the only way to derive $instr_def(t)$ is via clause (5.6), it follows that for each $0 \leqslant t \leqslant p(n)$, there must exist $s, q, s1, q1$ and d such that $instr(s, q, s1, q1, d, t) \in N$. There cannot be a time t with $0 \leqslant t \leqslant p(n)$ such that there two different 6-tuples $(s, q, s1, q1, d, t)$ and $(s', q', s1', q1', d', t)$ such that both atoms $instr(s, q, s1, q1, d, t)$ and $instr(s', q', s1', q1', d', t)$ are in N because then it follows from clauses (5.1)–(5.5), that $otherInstr(s2, q2, s3, q3, d3, t)$ holds for all 5-tuples in $S \times \Gamma \times S \times \Gamma \times D$ such that $(s3, q3, d3) \in \delta(s2, q2)$. But then the only clauses that have $instr(s, q, s1, q1, d, t)$ in the head are the clauses in either (4.1) or (4.2) and all such clauses would all be eliminated in the construction of $GL(edb_{M,p,\sigma} \cup P_{Trg}, N)$ so that there could be no $instr(s, q, s1, q1, d, t)$ in N. Thus it follows that for each $0 \leqslant t \leqslant p(n)$, there is a unique $(q_t, a_t, q'_t, a'_t, d_t)$ such that $instr(q_t, a_t, q'_t, a'_t, d_t, t)$ is in N.

It is then easy to check that our clauses in groups (1) through (4) ensure that the instructions $\{(q_t, a_t, q'_t, a'_t, d_t) : 0 \leqslant t \leqslant p(n)\}$ determine a valid run of the Turing machine M started on input σ. In particular, for each $0 \leqslant t \leqslant p(n)$, there is a unique position p_t such that the atom $position(p_t, t)$ is in N and q_t is the only state such that $state(q_t, t)$ is in N and a_t is the only symbol such that $tape(p_t, a_t, t)$ is in N. Moreover it is easy to check that for each time $0 \leqslant t \leqslant p(n)$ and each cell $0 \leqslant c \leqslant p(n)-1$, there is a unique symbol $a_{t,c}$ such that $tape(c, a_{t,c}, t)$ is in N. It follows if we define the sequence

$$\mathscr{C}_N = \langle C_0, \ldots, C_{p(n)} \rangle$$

so that for each $0 \leqslant t \leqslant p(n)$, $C_t = \langle i_t, S_t, k_t \rangle$ where

1. $i_t = \langle q_t, a_t, q'_t, a'_t, d_t \rangle$ and $instr(q_t, a_t, q'_t, a'_t, d_t, t) \in N$,
2. $S_t = \{< r, a >: tape(r, a, t) \in N\}$ and
3. k_t is the only k such that $position(k, t) \in N$,

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

then $\mathscr{C}_N$ is a valid run of M. Finally, it is easy to show by induction that $N_{\mathscr{C}_N} = N$. This, together with the fact that $\mathscr{C} \mapsto N_{\mathscr{C}}$ is one-to-one completes our argument. □

Corollary 1

A search problem $\mathscr{S}$ can be solved by means of a uniform logic program in SLP if and only if $\mathscr{S}$ is an NP-search problem.

A *supported model* M of a propositional logic program P is a subset of the Herbrand base of P that is a fixed point of the one step provablility operator, T_P associated with P. That is, M is a supported model of P if and only if $M = T_P(M)$. Thus a supported model M consists precisely of heads of clauses with bodies satisfied by M.

Given a logic program P we say that M is supported model of P if and only if M is a supported model of P_g.

Lemma 1

For all M, p, and σ, M is a supported model of $edb_{M,p,\sigma} \cup P_{Trg}$ if and only if M is stable model of $edb_{M,p,\sigma} \cup P_{Trg}$.

Proof

It is well known that for every program P, every stable model of P is a supported model of P.

For the other direction, suppose that $R = edb_{M,p,\sigma} \cup P_{Trg}$ and suppose that M is a supported model of R so that $T_R(M) = M$. First we observe that A cannot be in M. That is, all the clauses that have A in the head also have $\neg A$ in the body. Thus if $A \in M$, then there are no clauses with head A whose body is satisfied by M and hence A would not be in $T_R(M)$ which would violate that our assumption that $T_R(M) = M$. Since A is not in M, then it is easy to see that clause (6.2) forces M to contain the atom *completion* since otherwise A would be in $T_R(M)$. Similarly, by clause (5.7), for every $0 \leqslant t \leqslant p(n)$, it must be the case that $instr_def(t)$ must be in M. Since the only way to derive $instr_def(t)$ is via clause (5.6), it follows that for each $0 \leqslant t \leqslant p(n)$, M must satisfy the body of clause (5.6) where $T = t$. Thus there must exist $s, q, s1, q1$ and d such that $instr(s, q, s1, q1, d, t) \in M$. We claim that there cannot be a time t with $0 \leqslant t \leqslant p(n)$ such that there two different 6-tuples $(s, q, s1, q1, d, t)$ and $(s', q', s1', q1', d', t)$ such that both atoms $instr(s, q, s1, q1, d, t)$ and $instr(s', q', s1', q1', d', t)$ are in M. Otherwise the clauses (5.1)-(5.5) would show that that $otherInstr(s2, q2, s3, q3, d3, t) \in T_R(M) = M$ for all 5-tuples in $S \times \Gamma \times S \times \Gamma \times D$ such that $(s3, q3, d3) \in \delta(s2, q2)$. But then the only clauses that have $instr(s, q, s1, q1, d, t)$ in the head are the clauses in either (4.1) or (4.2) and the body of all such clauses would not be satisfied by M. Hence there could be no $instr(s, q, s1, q1, d, t)$ in $T_R(P)$ which would violate our assumption that $T_R(M) = M$. Thus it follows that for each $0 \leqslant t \leqslant p(n)$, there is a unique $(q_t, a_t, q'_t, a'_t, d_t)$ such that $instr(q_t, a_t, q'_t, a'_t, d_t, t)$ is in M.

We can then proceed exactly as in our proof of Theorem 1 to prove by induction that the fact $T_R(M) = M$ implies that M must be of the form $N_{\mathscr{C}}$ where

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

$\mathscr{C} = \langle C_0, \ldots, C_{p(n)} \rangle$ is a valid run of the machine M started on input σ. Thus M must be a stable model of M. □

Lemma 1 implies that analogue of Corollary 1 holds for Supported Logic Programming, SuLP.

Corollary 2
A search S problem can be solved by means of a uniform logic program in SuLP if and only if S is an NP-search problem.

We can also prove similar results for default logic programs without function symbols with respect to nondeterministic Turing machines with an oracle for 3-SAT.

Theorem 2
For each $n \in N$ there is a default theory $\langle W_n, D_n \rangle$ such that for every 3-SAT oracle Turing machine M, every polynomial $p \in N[x]$, and every finite input σ where $|\sigma| = n$, there is a polynomial-time one-to-one correspondence between the accepting computations of length $p(n)$ of M on input σ and the Reiter extensions of the default theory $\langle edb_{M,p,\sigma} \cup W_n, D_n \rangle$.

The proof of this result is more involved and requires additional technical means. It will be a subject of a separate publication. It follows from Theorem 2 that a search problem S can be solved by means of a uniform default logic program if and only if S is in Σ_2^P. A version of this result for decision problems has been proved in Cadoli *et al.* (1998).

Acknowledgements

The first author's research has been partially supported by the NSF grant IIS-0097278.[7] The second author's research has been partially supported by the ARO contract DAAD19-01-1-0724.

References

Abiteboul, S., Hull, R. and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Apt, K. R. 1990. Logic programming. In: *Handbook of Theoretical Computer Science*, pp. 475–574. Elsevier.

Apt, K. R. and Blair, H. A. 1990. Arithmetical classification of perfect models of stratified programs. *Fundamenta Informaticae 12*, 1–17.

Apt, K. R., Blair, H. A. and Walker, A. 1988. Towards a theory of declarative knowledge. *Foundations of Deductive Databases and Logic Programming*, pp. 89–142. Morgan Kaufmann.

Apt, K. R. and Pedreschi D. 1994. Reasoning about termination of pure Prolog programs. *Information and Computation 106*, 109–157.

[7] Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

Apt, K. R. and Pellegrini, A. 1994. On the occur-check free pure Prolog programs. *ACM Transactions on Programming Languages and Systems 16*, 687–726.

Blair, H. A., Marek, W. and Schlipf, J. 1995. The expressiveness of locally stratified programs. *Annals of Mathematics and Artificial Intelligence 15*, 209–229.

Bonatti, P. A. 2001. Reasoning with infinite stable models. *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 603–608. Morgan Kaufmann.

Bowen, K. A. and Kowalski, R. A. 1982. Amalgamating language and metalanguage in lLogic programming. *Logic Programming*, pp. 153–172. Academic Press.

Cadoli, M., Eiter, T. and Gottlob, G. 1997. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering 9*, 448–463.

Cadoli. M. and Palipoli, L. 1998. Circumscribing Datalog: expressive power and complexity. *Theoretical Computer Science 193*, 215–244.

Cenzer, D. and Remmel, J. B. 1999. $\prod_1^0$ Classes in Mathematics. *Handbook of Recursive Mathematics*, pp. 623–821. Elsevier.

Cholewiński, P., Marek, W., Mikitiuk, A. and Truszczyński, M. 1999. Programming with default logic. *Artificial Intelligence Journal 112*, 105–146.

Cholewiński, P., Marek, W. and Truszczyński, M. 1996. Default reasoning system DeReS. *Proceedings of Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pp. 518–528. Morgan Kaufmann.

Cook, S. 1971. The complexity of theorem-proving procedures. *Proceedings of Third Annual ACM Symposium on Theory of Computing*, pp. 151–158.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys 33*, 374–425.

Dung, P. M. and Kanchanasut, K. 1989. On the generalized predicate completion of non-Horn programs. *Logic programming, Proceedings of the North American Conference*, pp. 587–603. MIT Press.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. 1997. A deductive system for non-monotonic reasoning. *Proceedings 4th International Conference on Logic Programming and Nonmonotonic Reasoning: Lecture Notes in Computer Science 1265*, pp. 363–374. Springer-Verlag.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. and Scarcello, F. 1998. The KR System dlv: Progress Report, Comparisons, and Benchmarks. *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pp. 406–417. Morgan Kaufmann.

Garey, M. R. and Johnson, D. S. 1979. *Computers and intractability; a guide to the theory of NP-completeness*. W. H. Freeman.

Gelfond, M. and Lifschitz, V. 1988. The stable semantics for logic programs. *Proceedings 5th International Symposium on Logic Programming*, pp. 1070–1080. MIT Press.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Jaffar, J. and Maher, M. J. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19*(20), 503–581.

Lifschitz, V. 1999. Action languages, answer sets and planning. *The Logic Programming Paradigm*, pp. 357–373. Springer-Verlag.

Lifschitz, V. 1999. Answer set planning. *Proceedings 5th International Conference on Logic Programming and Nonmonotonic Reasoning: Springer Lecture Notes in Computer Science 1730*, pp. 373–374.

Marek, W., Nerode, A. and Remmel, J. B. 1994. The stable models of predicate logic programs. *Journal of Logic Programming 21*(3), 129–154.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press

Marek, V. W. and Remmel, J. B. 2001. On the foundations of answer set programming. *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pp. 124–131. AAAI Press.

Marek, W. and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM 38*, 588–619.

Marek, V. W. and Truszczyński, M. 1993. *Nonmonotonic Logic – Context-Dependent Reasoning*. Springer-Verlag.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, pp. 375–398. Springer-Verlag.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints: An Introduction*. MIT Press.

Niemelä, I. 1998. Logic programs with stable model semantics as a constraint programming paradigm. *Proceedings Workshop on Computational Aspects of Nonmonotonic Reasoning*, pp. 72–79.

Niemelä, I. and Simons, P. 1996. Efficient implementation of the well-founded and stable model semantics. *Proceedings Joint International Conference and Symposium on Logic Programming*, pp. 289–303. MIT Press.

Niemelä, I. and Simons, P. 1997. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs, *Proceedings 4th International Conference on Logic Programming and Nonmonotonic Reasoning: Springer Lecture Notes in Computer Science 1265*, pp. 420–429.

Przymusiński, T. 1988. On the declarative semantics of deductive databases and logic programs. *Foundations of deductive databases and logic programming*, pp. 193–216. Morgan Kaufmann.

Rao, P., Ramskrishnan, I. V., Sagonas, K., Swift, T., Warren, D. S. and Freire, D. S. 1997. XSB: A system for efficiently computing well-founded semantics. *Proceedings 4th International Conference on Logic Programming and Nonmonotonic Reasoning: Springer Lecture Notes in Computer Science 1265*, pp. 430–440.

Schlipf, J. 1995. The expressive powers of the logic programming semantics. *Journal of the Computer Systems and Science 51*, 64–86.

Syrjänen, T. 2001. Manual of Lparse version 1.0, `http://saturn.tcs.hut.fi/Software/smodels`

Soininen, T. and Niemelä, I. 1999. Developing a declarative rule language for applications in product configuration. *Proceedings First International Workshop in Practical Aspects of Declarative Languages*, pp. 305–319. Springer-Verlag.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM 38*, 620–650.

https://doi.org/10.1017/S1471068403001777 Published online by Cambridge University Press