From: AAAI Technical Report SS-01-01. Compilation copyright © 2001, AAAI (www.aaai.org). All rights reserved.

On the Foundations of Answer Set Programming

V.W. Marek
Department of Computer Science
University of Kentucky
J.B. Remmel
Department of Mathematics
University of California, San Diego

Abstract

Schlipf (Schlipf 1995) proved that the Stable Logic Programming solves all NP decision problems. We extend Schlipf's result to all search problems in the class NP. Moreover, we do this in a uniform way as defined in (Marek & Truszczyński 1999). Specifically, we show that there is a single DATALOG$^{\neg}$ program P_{Trg} so that for every Turing machine T, every polynomial with nonnegative coeeficients p, every positive integer n and an input σ of size at most n over a fixed alphabet Σ there is a polynomial-time encoding of the machine M and the input as an extensional database $edb_{M,p,\sigma}$ so that there is a one-to-one correspondence between the stable models of $edb_{M,p,\sigma} \cup P_{Trg}$ and accepting computations of the machine M that reach the final state in at most $p(n)$ steps. The decoding of computations form stable models is done in polynomial (in fact linear), time as well.

Introduction

The main motivation for this paper comes from recent developments in Knowledge Representation, especially the appearance of a new generation of systems (Niemelä & Simons 1996; Eiter & al. 1998; Cholewiński & al. 1999) based on the so-called Answer Set Programming (ASP) paradigm (Niemelä 1998; Cadoli & Palipoli 1998; Marek & Truszczyński 1999; Lifschitz 1998; 1999; East & Truszczyński 2000). In particular, these 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 on one particular ASP formalism, namely, the Stable Semantics for Logic Programs (SLP) (Gelfond & Lifschitz 1988) and related systems. We note that the underlying methods of ASP are similar to those used in Logic Programming (Colmerauer & al. 1973; Apt 1990) and Constraint Programming (Jaffar & Maher 1994; Marriott & 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 the individual elements of Herbrand base of the program while the constraint clauses in ASP act more globally in that they place restrictions on what subsets of the Herbrand base can be acceptable answers for the ASP 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 a stable model of P through this decoding. The program P is then submitted to the ASP engine such as *smodels* (Niemelä & Simons 1996), `dlv` (Eiter & al. 1998) or DeReS (Cholewiński & al. 1999) which computes the stable models of the program P. Thus the ASP engine finds the stable models of the program (if any exists) and we read-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 (T. Soininen & I. Niemelä 1999), combinatorial optimization problems (Cholewiński & al. 1999; Niemelä 1998) and other domains.

It is a well known fact that the semantics of existing Logic Programming systems such as Prolog have serious problems. For instance, the unification algorithm used by most dialects of Prolog do not enforce the occur check and hence these systems can produce incorrect results (Apt & Pellegrini 1994). Moreover, the processing strategies of Prolog and similar languages have the effect that correct logic programs can be nonterminating (Apt & Pedreschi 1994). While good programming techniques can overcome these problems, it is clear that such deficiencies have been limiting the appeal of the Logic Programming systems for ordinary programmers or system analysts. The promise of ASP and, in particular, of SLP and its extensions, e.g. Disjunctive Logic Programming, (Eiter & al. 1998), 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 occur 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, namely, the term formed from function symbols, variables and constants, is not available in SLP. Thus SLP systems require the programmer to explicitly construct many data structures. In this sense, SLP is similar to languages such as Perl (Wall & al. 2000) where finite functions and arrays are used to build data structures. In SLP programming, predicates are used to construct the required data structures and constraint type clauses are used to ensures 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 set of stable models. In addition, the order of the clauses of the program does not affect the set of stable models of the program.[1]. Finally the stable semantics of logic programs is well understood so that SLP programs have clear semantics.

We note that restriction that ASP programs do not have function symbols is absolutely crucial. First, it is well known that once one allows function symbols in a logic program P, then Herbrand base becomes infinite. Moreover, the stable models of logic programs with function symbols can be immensely complex. The first result which showed that the set of stable models can be complex can be found in (Apt & Blair 1990). For stratified logic programs (Apt & al. 1988; Przymusiński 1986), the perfect model is the unique stable model of that program (Gelfond & Lifschitz 1988). Apt and Blair (Apt & 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 is a powerful result limiting hopes for any meaningful practical programming with stratified programs if we allow function symbols. The result of (Apt & Blair 1990) was extended in (Blair & al. 1995) where it is shown that the stable models of a locally stratified program can capture any set in the hyperarithmetic hierarchy. In (Marek & al. 1994) it is shown that there is a uniform translation of trees contained in $\omega^{<\omega}$ into logic programs so that given an infinitely branching recursive tree $T \subseteq \omega^{<\omega}$, there is a finite predicate logic program P_T such that there is an effective one-to-one degree preserving correspondence between stable models of P and infinite branches through T. In effect, this result shows that the problem of finding a stable model for a finite predicate logic program is equivalent to finding an infinite path through an infinitely branching recursive tree. One consequence of this result is that the problem of determining whether a finite predicate logic program has a stable model is a Σ_1^1-complete. Moreover, the result of (Marek & al. 1994) gives us some idea about the structure of the family of stable models of the program. A complete characterization of families of stable models of logic programs in terms of inverse-Scott topology has been found in (Ferry 1994). More results on the structure of the family of stable models of the programs can be found in (Cenzer & Remmel 1999).

All the results mentioned in the previous paragraph show that stable semantics for logic programs admitting function symbols can be used only in a very limited setting. This is precisely what the XSB system does (Rao & al. 1997). The ASP systems such as those mentioned above, propose a more radical solution to the problem of complexity of stable models of logic programs with function symbols, namely, we should abandon function symbols entirely. Once this is accepted, the semantics of logic program can be defined in two stages. First, we assume that the program is interpreted over the Herbrand universe of the program, although, in principle, such a restriction is not necessary. Thus, given a logic program P, we assume, as in standard Logic Programming, that the universe is determined by the constants explicitly occurring in P. The set of these constants is finite. The program is grounded in these constants and the resulting program P_g is a propositional logic program. In the second stage, we compute stable models of the program P_g and those are the stable models of P. The process of grounding is performed by a separate grounding engine such as *lparse* (Niemelä & Simons 1996) and the result is passed to the engine computing the stable models. It is now clear that the features of SLP mentioned above, i.e., the absence of occur 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 occur 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 (although the processing, of course, does).

The language of logic programming without function symbols has been studied by the database community in expectation that it could lead to new, more powerful, database languages (Ullman 1988). It is called DATALOG and some database systems such as DB2 implement Horn part of DATALOG. The fact that admitting negation in the bodies of clauses leads to multiple stable models was unacceptable from the database perspective. Hence the semantics of DATALOG with negation used in database community was either a well-founded semantics (Van Gelder & al. 1991) (that is the

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

least three-valued stable model (Przymusiński 1990)), or other somewhat non-intuitive semantics such as the inflationary semantics (Abiteboul & 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. As we will see in a moment, the question itself requires some fine-tuning. First, let us look at the case of finite propositional programs. Here the situation is simple. Let At be a set of propositional atoms. Let $\mathcal{F}$ be a finite antichain of subsets of At (i.e. whenever $X, Y \in \mathcal{F}$, $X \subseteq Y$ then $X = Y$). Then there is a logic program $P_{\mathcal{F}}$ so that $\mathcal{F}$ is precisely the class of all stable models of $P_{\mathcal{F}}$ (Marek & Truszczyński 1993). Moreover, the family of stable models of any program P forms an antichain. Thus in the finite propositional logic program case, we have a complete characterization. However, this result does not tell us anything about the uniformity and the effectiveness of the construction. The problem of whether there exists a stable model of a propositional program is NP-complete (Marek & Truszczyński 1991). For the DATALOG (with negation), an analogous result has been obtained in (Schlipf 1995).

To refine, our question about what can be computed by logic programs without functions symbols under the stable model semantics, we first introduce the notion of *search problem* (Garey & Johnson 1979) and of a *uniform* logic program (Marek & Truszczyński 1999). A search problem is a set $\mathcal{S}$ of finite instances (Garey & Johnson 1979). For example, the search problem may be to find Hamiltonian paths in a graph so that the set of instances of the problem is the set of all finite graphs. Given an instance $I \in \mathcal{S}$, we have a set S_I of solutions to $\mathcal{S}$ for instance I. For example, in our Hamiltonian path problem, given a finite graph I, S_I is the set of all Hamiltonian paths of I. An algorithm solves the problem $\mathcal{S}$ if it returns a solution $s \in S_I$ whenever S_I is non-empty, and returns "empty" otherwise. Decision problems are special cases of search problems. Now consider a search problem $\mathcal{S}$. Assume that there exists

1. a polynomial time encoding $edb_{\mathcal{S}}$ under which every instance I of $\mathcal{S}$ is mapped to a finite set of atoms and
2. a *single* logic program $P_{\mathcal{S}}$ such that there is a polynomial time computable function $sol_{\mathcal{S}}(\cdot,\cdot)$ such that for every instance I of $\mathcal{S}$, $sol_{\mathcal{S}}(I,\cdot)$ maps the set of stable models of the $edb_{\mathcal{S}}(I) \cup P$ onto the set of solutions S_I of I.

In such a situation, we say that the search problem $\mathcal{S}$ is solved by a uniform logic program. Schlipf (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.

The goal of our paper is to prove the strengthening of this result. Namely we will prove that the Schlipf result remains true if we deal with search problems. We shall exhibit a *single* logic program P that is capable to simulating polynomial time nondeterministic Turing Machines. That is, given any polynomial time nondeterministic Turing machine M and any input σ, there is a set of facts $C_{M,\sigma}$ such that a stable model of $P \cup C_{M,\sigma}$ codes an accepting computation of M started with input σ and and any accepting computation of M started on input σ is coded by some stable model of $P \cup C_{M,\sigma}$. This results will show that logic programs without function symbols under the stable logic semantics can capture precisely the set of NP-search problems. Our proof of this result uses essentially the same technique used by Cook (Cook 1971) in his proof of the NP-completeness of the satisfiability problem. That is, given the run-time polynomial bound $p(x)$ for M, stable models of $P \cup C_{M,\sigma}$ encodes the sequence of steps of length $p(n)$ which occur in an accepting computation of M started on σ if σ has length n. More precisely, give a nondeterministic polynomial time Turing machine M, the clauses $C_{M,x}$ will incorporate the description of M as the part of the instance description (extensional database) to the program P. The input σ will also be a part of the extensional data base. Call this extensional database $edb_{M,p,\sigma}$. The size of $edb_{M,p,\sigma}$ is polynomial in the size of M, $|\sigma|$, and the exponent of p. There will be a one-to-one polynomial time correspondence between the computations of the machine M that reach the final accepting state in at most $p(n)$ steps and the stable models of $P \cup edb_{M,p,\sigma}$. Thus all search problems in class NP can be solved by a uniform logic program. The converse implication is, of course, obvious.

We shall also show that we can apply the same techniques to construct a metainterpreter for stable logic programming. Specifically, we show that the search problem of finding a stable models for propositional logic program can also be solved by means of a uniform logic program. Specifically, we represent any propositional logic program by a set of propositional atoms that serve as an extensional database for a metainterpreter. In fact, we can construct two such metainterpreters with different properties. In this paper, we construct a simple one that computes stable models for so-called 0-2 programs. A result due to Blair (Blair & al. 1999) shows that every propositional program can be semirepresented in the sense of (Marek & Truszczyński 1993) by a 0-2 logic program and that the representation can be computed in linear time. A more complex metainterpreter that directly computes stable models can be constructed along the lines of the proof of Theorem 2 below. Due to the space limitations, we shall not give any details about this second metainterpreter.

Technical preliminaries

In this section we formally introduce several notions that will be needed for the proof of our result.

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

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

where $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are atoms, possibly with variables and/or constants. A program is a finite set P of clauses of the form (1). Each program determines its language (based on the predicates occurring in the program), Herbrand universe U (set of all constant terms occurring in the program), and Herbrand base H_P (the set of all ground atoms of the language). Since there is no function symbols in our programs, both Herbrand universe and Herbrand base are finite.

A ground instance of the clause C of the form (1) is the result of a simultaneous substitution of constants for variables occurring in C. Given a program P, P_g is the propositional program consisting of all ground substitutions of clauses of P.

Given a propositional program P and a set $M \subseteq H_P$, the Gelfond-Lifschitz transform of P by means of M, $GL(P, M)$ is the program $GL(P, M)$ arising from P as follows. First, eliminate all clauses C in P such that for some j, $1 \leq j \leq n$, $r_j \in M$. Finally, in any remaining clauses, we 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, we say that M is a stable model of a DATALOG$^\neg$ program P, if M is a stable model of the propositional program P_g.

A nondeterministic Turing Machine is a structure of the form $M = (Q, \Sigma, \Gamma, D, \delta, s, f)$ where Q is a finite set of states, Σ is a finite alphabet of input symbols. We will assume that B is an additional symbol such that $B \notin \Sigma$. The symbol B will denote "Blank". The set $\Gamma = \Sigma \cup \{B\}$ is the set of tape symbols. The set D of move directions will consist of elements L, R, and λ (L is a "move left" symbol, R is a "move right" symbol, and λ is "stay put" symbol). The function $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times D)$ is the transition function of the machine M. The state $s \in Q$ is the start state, and $f \in Q$ is the final state. To visualize the behavior of the machine M we talk about the read-write head of the machine. The read-write head of the machine is always in some state from set Q. As usual, a Turing machine M operates on a tape. The tape could be thought as starting at the cell with the index 1 and containing infinitely many cells i for $i > 0$ to the right. However, when we consider a polynomial time nondeterministic Turing machine, M, with run time polynomial $p(x)$, then on any input σ of length n, an accepting computation 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 can implicitly assume that that the tape is finite.

A runtime polynomial p of a nondeterministic Turing machine M is a polynomial of the form $p(x) = a_0 + a_1x + \cdots + a_kx^k$ where each $a_i \in N = \{0, 1, 2, \ldots\}$ and $a_k \neq 0$. If p is the runtime polynomial for M, then on any input of size n, M computes an answer in at most we reach the final step in at most $p(n)$ steps for any input of length n. Given a nondetrministic Turing machine M with a runtime polynomial p, we modify the usual 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 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.

Uniform coding of Turing Machines by a Logic Program

In this section we describe a logic program used for the uniform coding of the operation of Turing machines. This program P is the core part of our encoding of NP-search problems. The encoding of the input, specific machine, and the run-time bound will all be encoded by a collection of facts, i.e. of ground atoms.

Firstly, we need to define the language (i.e. a signature) of the program P. Here is the collection of atoms (predicate symbols) that we use for description of P. We also explain the use for each symbol.

$time(X)$ for "X is a time step".

$lasttime(X)$ for "X is the last time step".

$cell(X)$ for "X is a cell number".

$symb(X)$ for "X is a symbol".

$state(S)$ for "S is a state".

$action(X)$ for "$X \in \{L, R, \lambda\}$".

$rwct(Y, T)$ for "the read-write head is reading cell Y at time T".

$rwst(X, T)$ for "machine M is in state X at time T".

$otherrwst(X, T)$ for "machine M is in state other than X at time T".

$syct(X, Y, T)$ for "s symbol X is in cell Y at time T".

$othersyct(X, Y, Z)$ for "symbol other than X is in cell Y at time T".

$delta(X, Y, X1, Y1, Z)$ for "the triple $(X1, Y1, Z)$ is an executable instruction when the read-write head is in state X and reads Y." (This five-place relation represents the transition function δ).

$nogood(X, Y)$ for "the read-write head is in state X and reads the symbol Y, but there is no instruction to execute."

$inst(X, Y, Z)$ for "(X, Y, Z) is an instruction".

$exinst(X, Y, Z, T)$ for "machine M executes the instruction (X, Y, Z) at time T". Here we assume that instruction (X, Y, Z) is executed at the start of any given time step $T > 1$ and that the instruction (X, Y, Z) was chosen by M at the previous time step.

$candidateexinst(X, Y, Z, T)$ for "(X, Y, Z) is one of the instructions that could be executed at time T".

$otherexinst(X,Y,Z,T)$ for "the instruction (X,Y,Z) is not executed at time T".

$succ(X,Y)$ for "$Y = X+1$"

$neq(X,Y)$ for "$X \neq Y$"

A is a propositional letter.

When we encode a specific machine, input and runtime, we will need constants. Notice that in the program P that we will write, there should be no constants. We will not be absolutely strict in this respect in this paper. In the program P described below, we use five constants l, r, λ, f and 1. These constants can easily be eliminated by replacing the explicit use of such constants by new one-place unary predicates *pred-lambda*, *pred-r* etc. Then, in the extensional database, we need only ensure that *pred-lambda* is satisfied only by λ, *pred-r* is only satisfied by r, etc.

The following will be the constants that will be used in our description of time, cell numbers, cell contents and specific machines. The last two families of constants will be "machine-dependent", since we did not specify any restrictions on the finite sets Q and Σ. (1) constant symbols $0, 1, \ldots, p(n)$ where n is the length of the input σ, p is the runtime polynomial. (2) $q_0, \ldots, q_{p+1}$, where $Q = \{q_0, \ldots, q_{p+1}\}$, and we will assume that $s = q_0$ and $f = q_{p+1}$. (3) $a_1, \ldots, a_{r+1}$ where $\Sigma = \{a_1, \ldots, a_r\}$, and B (blank symbol) is a_{r+1} (4) r, l, λ. We are now ready to write the program P.

1. Our first three clauses will ensure that in a stable model, the read-write head is in exactly one state at any given time t.

(1.1) $A \leftarrow time(T), state(X), state(Y), rwst(X,T), rwst(Y,T), neq(X,Y), \neg A.$

(1.2) $rwst(X,T) \leftarrow time(T), state(X), \neg otherrwst(X,T).$

(1.3) $otherrwst(X,T) \leftarrow time(T), state(X), state(Y), rwst(Y,T), neq(X,Y).$

2. Our next three clauses will ensure that in a stable model, there is exactly one symbol in cell y at any given time t.

(2.1) $A \leftarrow time(T), cell(Y), symb(X), symb(X1), syct(X,Y,T), syct(X1,Y,T), neq(X,X1), \neg A.$

(2.2) $syct(X,Y,T) \leftarrow time(T), cell(Y), symb(X), \neg othersyct(X,Y,T)$

(2.3) $othersyct(X,Y,T) \leftarrow time(T), cell(Y), symb(X), symb(X1), syct(X1,Y,T), neq(X,X1).$

3. These clauses describe the operation of the Turing machine.

Case 1. The read-write head is in the final state f.

(3.1.1) $rwst(f,W) \leftarrow time(T), time(W), succ(T,W), rwst(f,T)$

(3.1.2) $rwct(Y,W) \leftarrow time(T), time(W), succ(T,W), cell(Y), rwct(Y,T), rwst(f,T)$

(3.1.3) $syct(X,Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), syct(X,Y,T), rwst(f,T)$

(3.1.4) $exinst(f,X,\lambda,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), rwct(Y,T), syct(X,Y,T), rwst(f,X).$

Case 2. The read-write state is q_i at time t where $q_i \neq f$ and we are reading symbol a_j at time t, but $\delta(q_i, a_j) = \emptyset$. We have two clauses that ensure that we can not get into this situation in a stable model.

(3.2.1) $A \leftarrow time(T), cell(Y), symb(X), state(Z), rwst(X,T), syct(X,Y,T), nogood(X,Z), \neg A$

Case 3. The read-write state is q_i at time t where $q_i \neq f$ and $t \neq p(n)$, we are reading symbol a_j at time t, and

$$\delta(q_i, a_j) = \{(q_{i_0}, a_{j_0}, \gamma_{k_0}), \ldots, (q_{i_d}, a_{j_d}, \gamma_{k_d})\}$$

Our next clauses are designed to ensure that there is a unique instruction to be executed at time t.

(3.3.1) $candidateexinst(S1,X1,Z,W) \leftarrow state(S), state(S1), symb(X), symb(X1), action(Z), time(T), time(W), succ(T,W), rwct(Y,T), syct(X,Y,T), rwst(S,T), delta(S,X,S1,X1,Z).$

(3.3.2) $exinst(S,X,Z,W) \leftarrow state(S), symb(X), action(Z), time(W), candidateexinst(S,X,Z,W), \neg otherexinst(S,X,Z,W).$

(3.3.3) $otherexinst(S,X,Z,W) \leftarrow state(S), state(S1), symb(X), symb(X1), action(Z), action(Z1), exinst(S1,X1,Z1,W), neq(S,S1).$

(3.3.4) $otherexinst(S,X,Z,W) \leftarrow state(S), state(S1), symb(X), symb(X1), action(Z), action(Z1), exinst(S1,X1,Z1,W), neq(X,X1).$

(3.3.5) $otherexinst(S,X,Z,W) \leftarrow state(S), state(S1), symb(X), symb(X1), action(Z), action(Z1), exinst(S1,X1,Z1,W), neq(Z,Z1).$

4. Executing the next instruction. These clauses describe the computation process, i.e. the change in time, the state of the read-write head, the content of tape, and the position of the read-write head.

(4.1.1) (Inertia rule - dealing with the content of cells that do not change)
$syct(X,Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), cell(Z), neq(Y,Z), rwct(Z,T), syct(X,Y,T).$

(4.1.2) (Content of the cell changes according to instruction)
$syct(X,Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), state(S), action(Z), exinstr(S,X,Z,W), rwct(Y,T).$

(4.2) (Changing the state of the read-write head according to instruction)
$rwst(S,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), state(S), action(Z), exinstr(S,X,Z,W), rwct(Y,T).$

(4.3.1) (Moving the read-write head left, if at a cell with index that is different from 1)
$rwct(Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), cell(Z), neq(Z,1), succ(Y,Z), exinstr(S,X,l,W), rwct(Z,T).$

(4.3.2) (Blocking an attempt to move left the read-write head when pointing to the cell of index 1)
$A \leftarrow time(T), time(W), succ(T,W), symb(X), state(S), exinstr(S,X,l,W), rcwt(1,T), \neg A.$

(4.4) (Moving the read-write head right).
$rwct(Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), cell(Z), succ(Z,Y), exinstr(S,X,r,W), rwct(Z,T).$

(4.5) (Leaving the read-write head where it is).
$rwct(Y,W) \leftarrow time(T), time(W), succ(T,W), symb(X), cell(Y), exinstr(S,X,\lambda,W), rwct(Y,T).$

5. Our next set of clauses ensures that in a stable model, the read-write head is over exactly one cell at time t.

(5.1) $A \leftarrow time(T), cell(Y), cell(Y1), neq(Y,Y1), rwct(Y,T), rwct(Y1,T), \neg A.$

(5.2) $rwct(Y,T) \leftarrow time(T), cell(Y), \neg otherrwct(Y,T).$

(5.3) $otherrwct(Y,T) \leftarrow time(T), cell(Y), cell(Y1), neq(Y,Y1), rwct(Y1,T).$

6. This group consists of a single clause. That clause ensures that in a stable model, the read-write head ends in state $f = q_{p+1}$ at time $t = p(n)$.)
$A \leftarrow lasttime(T), state(X), neq(X,f), rwst(X,T), \neg A$

We call the DATALOG$^{\neg}$ program consisting of all clauses in groups (1)-(6), Turing program, P_{Trg}.

Extensional database – coding of the machine, input and the run-time

We will define now the extensional database $ext_{M,p,\sigma}$. Given initial input $\sigma = \sigma_1 \ldots \sigma_n$ and runtime $p(n)$, we let $I_{(\sigma,p(n))}$ consist of the following set of facts.

1. Basic descriptive facts
   (a) $state(q_i) \leftarrow$ for $i = 0, \ldots, p+1$.
   (b) $symb(a_i) \leftarrow$ for $i = 1, \ldots, r+1$.
   (c) $delta(s, x, s1, x1, d) \leftarrow$ for every pair (s,x) and every triple $(s1, x1, d) \in \delta(s,x)$.
   (d) $nogood(s,x)$ for every pair (s,x) such that $\delta(s,x) = \emptyset$.
   (e) $succ(i, i+1) \leftarrow$ for $0 \leq i < p(n)$.
   (f) $neq(a,b) \leftarrow$ for all $a, b \in \{0, \ldots, p(n)\}$ with $a \neq b$.
   (g) $time(i) \leftarrow$ for $0 \leq i \leq p(n)$.
   (h) $lasttime(p(n)-1)$.
   (i) $cell(i) \leftarrow$ for $1 \leq i \leq p(n)$.
   (j) $syct(\sigma_i, i, 0) \leftarrow$ for $i = 1, \ldots, n$ and $syct(B, i, 0) \leftarrow$ for $i = n+1, \ldots, p(n)$.
   (k) $action(l)$, $action(r)$, $action(\lambda)$.
2. Beginning state of the Read-Write Head
   $rwst(q_0, 0) \leftarrow$ (recall $q_0 = s$ is the start state.)
3. $rwct(1,0) \leftarrow$. This is initial read-write head cell condition.

Main Results

The main results of this paper can be summed up in the following three results. Our first proposition immediately follows from our construction.

Theorem 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 equal to $q(|M|, |\sigma|, p(|\sigma|)$.*

We will prove that for any nondeterministic Turing Machine M, runtime polynomial $p(x)$, and input σ of length n, that 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 2 *The mapping of Turing machines to DATALOG$^{\neg}$ programs defined by $M \mapsto edb_{M,p,\sigma} \cup P_{Trg}$ 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 computations of M of the length $p(n)$ ending in the state f.*

Corollary 1 *A search S problem can be solved by means of a uniform logic program if and only if S is an NP-search problem.*

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

Metainterpreters for logic programs

We will now discuss metainterpreters, i.e. programs that process programs. Our goal is to build a single DATALOG$^{\neg}$ program P that given a coding edb_Q of a propositional logic program Q, the stable models of $P \cup edb_Q$ codes the stable models of Q. Such program is analogous to the classical 'vanilla metainterpreter' of (Bowen & Kowalski 1982).

In this paper, we will describe a metainterpreter for the class 0-2 programs as described in the introduction. A propositional program P is a *0-2 program* if for every clause C of P has either no positive literal in the body, or exactly 2 positive literals in the body. Blair proved that 0-2 programs semirepresent all propositional programs. (See (Marek & Truszczyński 1993), Ch. 5, for the discussion of semirepresentability). This result is implicit in (Blair & al. 1999). Specifically we have the following.

Theorem 1 (Blair) *There is a linear-lime computable function f that assigns to each propositional program P a 0-2 program $f(P)$ and such that there is a one-to-one projection from the family of stable models of $f(P)$ to the family of stable models of P.*

To put Blair's result in proper perspective, we note that there is a more restrictive class of programs with an even stronger property. Namely, call a program P *purely negative* if the clauses of P have no positive literals in the body. Dung and Kanchanasut (Dung & Kanchanasut 1989) proved that every propositional program is representable by a purely negative program. However, the purely negative program of Dung and Kanchanasut can not be computed in polynomial time from the original program. Thus the cost of this type of more extensive preprocessing would be prohibitive.

We shall now construct our metainterpreter *Meta1* that computes stable models of 0-2 propositional programs. To this end we need a data structure that expresses the given 0-2 program Q. The extensional predicates that occur in the extensional database edb_Q are as follows:

1. $atom(\cdot)$ for listing atoms,
2. $clause(\cdot)$ for listing clauses,
3. $head(\cdot,\cdot)$ for specifying the head of a clause,
4. $neg(\cdot,\cdot)$ for specifying that an atom occurs negatively in the body of a clause,
5. $first(\cdot,\cdot)$ for specifying that an atom is the first of the two positive atoms occurring in the body of a clause, and
6. $second(\cdot,\cdot)$ for specifying that an atom is the second of two positive atoms occurring in the body of a clause.

Next we list the intentional predicates that also occur in *Meta1*.

1. $nempty(\cdot)$ for specifying that there are atoms occurring positively in the body of a clause.
2. $empty(\cdot)$ for specifying that there are no atoms occurring positively in the body of a clause.
3. $in(\cdot)$ for specifying the atoms which occur in the stable model of the input program.
4. $out(\cdot)$ for specifying the atoms which do not occur in the stable model of the input program.
5. $unusable(\cdot)$, for specifying the clauses do not occur in the Gelfond-Lifschitz reduct relative to the stable model.
6. $usable(\cdot)$ for specifying the clauses that occur in the Gelfond-Lifschitz reduct relative to the stable model.
7. $computed(\cdot)$ for specifying the atoms which are computed to be in the stable model.
8. f a propositional atoms used to enforce constraints.

The extensional database edb_Q for a propositional program Q consists of the atoms $atom(a)$ for all atoms a occurring in Q, the atoms $clause(c)$ for all clauses c in Q, the atoms $head(a,c)$ for the head a of clause c, the atoms $first(a,c)$ and $second(b,c)$ where a,b are first and the second atom in the body of c, respectively. We denote this collection of atoms edb_Q.

We now describe the metainterpreter itself.

1. Generating the candidate for a model
   (a) $in(A) \leftarrow atom(A), \neg out(A)$
   (b) $out(A) \leftarrow atom(A), \neg in(A)$
2. Computing Gelfond-Lifschitz reduct
   (a) $unusable(C) \leftarrow clause(C), atom(A), neg(A,C), in(A)$
   (b) $usable(C) \leftarrow clause(C), \neg unusable(C)$
3. Classifying clauses
   (a) $nempty(C) \leftarrow clause(C), atom(A), first(A,C)$
   (b) $empty(C) \leftarrow clause(C), \neg nempty(C)$
4. Computation process
   (a) $computed(A) \leftarrow clause(C), empty(C), usable(C), head(A,C)$
   (b) $computed(A) \leftarrow clause(C), first(A1,C), second(A2,C), computed(A1), computed(A2), head(A,C)$
5. Constraints
   (a) $f \leftarrow atom(A), in(A), \neg computed(A), \neg f$
   (b) $f \leftarrow atom(A), out(A), computed(A), \neg f$

We now have the following result

Theorem 3 *For every propositional 0-2 program Q, there is a one-to-one correspondence between set of stable models of Q and the set of stable models of Meta1 $\cup$ edb_Q. Moreover, we can recover a stable model of Q from its corresponding stable model of Meta1 $\cup$ edb_Q in linear time.*

Thus, in order to compute stable models of a propositional program P, first process P using Proposition 1, getting a program Q. Next, apply Proposition 3, get a stable model of *Meta1* $\cup$ edb_Q and project it twice. First to get a stable model of Q and then of P.

We note that the metainterpreter *Meta1* does distinguish between the stable and supported semantics. That is, the supported models of *Meta1* $\cup$ edb_Q do not necessarily determine stable models of Q. Finally, we note that by using techniques similar to those used to construct our program P_{Trg}, we can write another metainterpreter *Meta2* that does not require a particular form for the input propositional program. This requires a more complicated construction where we directly simulate the iterations of the operator $T_{GL_M(Q)}$ in the computation of a stable model M from it Gelfond-Lifschitz reduct $GL_M(Q)$. We can not supply details here due to lack of space. However, we will provide such details in the full version of the paper.

References

K.R. Apt Logic Programming. in *Handbook of Theoretical Computer Science*, pp. 475–574. Elsevier, 1990.

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

K. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of deductive databases and logic programming*, pages 89–142, Los Altos, CA, 1988. Morgan Kaufmann.

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

K.R. Apt and A. Pellegrini. On the occur-check free pure Prolog programs. *ACM Toplas* 16:687–726, 1994.

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

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

H.A. Blair, F. Dushin, D.W. Jackel, A.J. Rivera and M. Sezgin, Continuous Models of Computation for Logic Programs: Importing Continuous Mathematics into Logic Programming's Algorithmic Foundations, *The Logic Programming Paradigm*, pp. 231-255, Series *Artificial Intelligence*, Springer-Verlag, 1999.

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

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

D. Cenzer and J.B. Remmel Π_1^0 Classes in Mathematics. In: *Handbook of Recursive Mathematics* pp. 623–821, Elsevier 1999.

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

A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Technical report, University of Marseille, 1973.

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

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

East, D., and Truszczyński, M. 2000. Datalog with constraints. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, 163–168, 2000.

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

A. Ferry, *Topological Characterizations for Logic Programming Semantics*, Ph.D. Dissertation, University of Michigan, 1994.

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

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

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

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

V. Lifschitz. Answer set planning. In *Logic programming and nonmonotonic reasoning*, volume 1730 of *Lecture Notes in Computer Science*, pages 373–374. Springer-Verlag, 1999.

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

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

W. Marek and M. Truszczyński. *Nonmonotonic Logic - Context-Dependent Reasoning.* Series *Artificial Intelligence*, Springer-Verlag, 1993.

W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. *The Logic Programming Paradigm*, pp. 375–398. Series *Artificial Intelligence*, Springer-Verlag, 1999.

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

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

I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96.* MIT Press, 1996.

T. Przymusiński. On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann.

T. Przymusinski. The Well-Founded Semantics Coincides With The Three-Valued Stable Semantics, *Fundamenta Informaticae*, 13:445–464, 1990.

P. Rao, I.V. Ramskrishnan, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of LP-NMR'97*, pages 430–440, Lecture Notes in Computer Science, 1265, Springer-Verlag, 1997.

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

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

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

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

L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*, Third edition, O'Reilly, 2000.