Answer Set Planning

Vladimir Lifschitz
University of Texas
Austin, TX 78712, USA
vl@cs.utexas.edu

Abstract

In "answer set programming," solutions to a problem are represented by answer sets, and not by answer substitutions produced in response to a query, as in conventional logic programming. Instead of Prolog, answer set programming uses software systems capable of computing answer sets. This paper is about applications of this idea to planning.

1 Introduction

Over the years, the declarative semantics of negation as failure has attracted the attention of many researchers. Take the one-line program

$$p \leftarrow not\ q. \tag{1}$$

Given this program, a logic programming system is expected to answer *yes* to query p. Can we explain why—without referring to query evaluation procedures, in declarative terms?

Each of many existing theories of negation as failure answers this question in its own way. One can say, for instance, that p logically follows from the "completion" of this program [1]; or that p is true in its "well-founded model" [22]; or that p belongs to its "answer set" [7].

For some logic programs, theories of negation as failure do not describe a *yes*-part of the set of ground queries as definitely as they do for program (1). There may be a "gray area" where neither *yes* nor *no* would be an appropriate answer. This happens, for instance, for the program consisting of two rules

$$\begin{aligned} p &\leftarrow not\ q, \\ q &\leftarrow not\ p. \end{aligned} \tag{2}$$

The completion of (2) does not entail p, but it does not entail the negation of p either. The truth value assigned to p in the well-founded model of (2) is neither *true* nor *false*; it is *undefined*. Furthermore, (2) has two answer sets. One of them is $\{p\}$; it suggests that p should be viewed as a *yes*-query. But the second answer set $\{q\}$ suggests that that would be incorrect. The situation with query q is similar. Both p and q are in the gray area of program (2). Accordingly, the evaluation of these queries by SLDNF resolution would not terminate.

One might think that programs with large gray areas are pretty much useless. But such programs play an important part in the work of the proponents of *answer set programming*[1] [14, 16] who represent solutions to a problem by answer sets, and not by answer substitutions produced in response to a query, as in conventional logic programming. Instead of Prolog, answer set programming requires a software system capable of computing answer sets. Four such systems were demonstrated at the Workshop on Logic-Based AI held in June of 1999 in Washington, DC: SMODELS, DLV, DeReS and CCALC.[2] Their creators keep learning from each other and from the experience accumulated by users, and the performance of the systems keeps improving with every new release.

Two examples of answer set programming are given in Section 3 below. A potentially important application of this programming style is described in [19]. Answer set programming has been also used as a planning method, which is the main subject of this paper.

2 Answer Sets

First we will we review the definition of an answer set. Consider a set of propositional symbols, called *atoms*. A *literal* is an expression of the form A or $\neg A$, where A is an atom. (We call the symbol $\neg$ "classical negation," to distinguish it from the symbol *not* used for negation as failure.[3]) A *rule element* is an expression of the form L or *not* L, where L is a literal. A *rule* is an ordered pair

$$Head \leftarrow Body \tag{3}$$

where *Head* and *Body* are finite sets of rule elements. If

$$Head = \{L_1, \dots, L_k, not\ L_{k+1}, \dots, not\ L_l\}$$

and

$$Body = \{L_{l+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n\}$$

($n \geq m \geq l \geq k \geq 0$) then we will write (3) as

$$L_1; \dots; L_k; not\ L_{k+1}; \dots; not\ L_l \leftarrow L_{l+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n. \tag{4}$$

A rule (3) is a *constraint* if $Head = \emptyset$. A *program* is a set of rules.

The notion of an answer set is defined first for programs that do not contain negation as failure ($l = k$ and $n = m$ in every rule (4) of the program). Let Π be such a program, and let X be a consistent set of literals.[4] We say that X is *closed* under Π if, for every rule (3) in Π, $Head \cap X \neq \emptyset$ whenever $Body \subseteq X$. We say that X is an *answer set* for Π if X is minimal among the sets closed under Π (relative to set inclusion).

For instance, the program

$$\begin{array}{c} p; q \leftarrow \\ \neg r \leftarrow p \end{array} \tag{5}$$

has two answer sets: $\{p, \neg r\}$ and $\{q\}$. If we add the constraint

$$\leftarrow q$$

to (5), we will get a program whose only answer set is $\{p, \neg r\}$.

The last example illustrates a general property of constraints: adding a constraint to a program affects its collection of answer sets by eliminating the answer sets that "violate" this constraint. This is true for programs with negation as failure also.

To extend the definition of an answer set to programs with negation as failure, take any program Π, and let X be a consistent set of literals. The *reduct* Π^X of Π relative to X is the set of rules

$$L_1; \ldots; L_k \leftarrow L_{l+1}, \ldots, L_m$$

for all rules (4) in Π such that X contains all the literals $L_{k+1}, \ldots, L_l$ but does not contain any of $L_{m+1}, \ldots, L_n$. Thus Π^X is a program without negation as failure. We say that X is an *answer set* for Π if X is an answer set for Π^X.

For instance, the reduct of (1) relative to $\{p\}$ consists of one rule $p \leftarrow$. Since set $\{p\}$ is an answer set for this reduct, it is an answer set for (1). The reduct of (1) relative to $\{p, q\}$ is the empty set. Since set $\{p, q\}$ is not an answer set for this reduct, it is not an answer set for (1).

3 Programming with Answer Sets

As an example, we will show how answer set programming can be used to find all Hamiltonian cycles in a given finite directed graph G. To this end, we will turn G into a program whose answer sets correspond to the Hamiltonian cycles in G. (The program is reproduced, with minor changes, from [14].)

Let the vertices of G be $0, \ldots, n$. The set of atoms of the program consists of the expressions $in(u, v)$ for all edges $\langle u, v\rangle$ of G and the expressions $reachable(u)$ for all vertices u. The first group of rules is

$$in(u, v); \neg in(u, v) \leftarrow \tag{6}$$

for all edges $\langle u, v\rangle$. The answer sets for this part of the program are in a 1–1 correspondence with the subsets of the edges of G: an answer set X corresponds to the subset $\{\langle u, v\rangle \;:\; in(u, v) \in X\}$. We will now add constraints that eliminate all subsets other than the Hamiltonian cycles.

This is done in two steps. First, the constraints

$$\leftarrow in(u, v), in(u, w) \tag{7}$$

for all edges $\langle u, v\rangle$ and $\langle u, w\rangle$ with $v \neq w$ and the constraints

$$\leftarrow in(u, w), in(v, w) \tag{8}$$

```
                 1          2
1   2            2          1
-------        -----      -----
```

Figure 1: Configurations of blocks for $n = 2$

for all edges $\langle u, w\rangle$ and $\langle v, w\rangle$ with $u \neq v$ are added. These constraints prohibit edges with a common endpoint. It remains to ensure that, by starting at 0 and following the *in*-edges, one can visit all vertices of G and come back to 0. To this end, we define

$$\begin{array}{l} \mathit{reachable}(u) \leftarrow \mathit{in}(0, u) \\ \mathit{reachable}(v) \leftarrow \mathit{reachable}(u), \mathit{in}(u, v) \end{array} \tag{9}$$

and add the constraints

$$\leftarrow \mathit{not}\ \mathit{reachable}(u). \tag{10}$$

Our second example is a favorite toy world of AI researchers—the blocks world. Each of n blocks can be somewhere on the table or on top of exactly one other block. Each block can have at most one other block immediately on top of it. Figure 1 shows all possible configurations of blocks for $n = 2$. If $n = 3$, the number of possible configurations is 13. How to find all possible configurations?

The answer sets for program (11) below correspond to the possible configurations of a given set of blocks. The range of the "block variables" b, b', b'' consists of the names of blocks; the range of the "location variables" l, l' includes both the names of blocks and the symbol *table*.

$$\begin{array}{ll} \mathit{on}(b, l); \neg \mathit{on}(b, l) \leftarrow & \\ \leftarrow \mathit{on}(b, l), \mathit{on}(b, l') & (l \neq l') \\ \leftarrow \mathit{on}(b, b''), \mathit{on}(b', b'') & (b \neq b') \\ \mathit{supported}(b) \leftarrow \mathit{on}(b, \mathit{table}) & \\ \mathit{supported}(b) \leftarrow \mathit{supported}(b'), \mathit{on}(b, b') & \\ \leftarrow \mathit{not}\ \mathit{supported}(b) & \end{array} \tag{11}$$

4 Systems DLV and SMODELS

We will now briefly describe some possibilities of two of the systems mentioned in the introduction. System DLV computes answer sets for finite programs without negation as failure in the heads of rules ($l = k$ in every rule (4) of the program). For instance, given the input file

```
p :- not q.
q :- not p.
```

it will return the answer sets for program (2). Given the input file

```
p; q.
-r :- p.
```

it will return the answer sets for program (5). Schematic rules with variables can be used to specify sets of rules obtained by instantiation. Variables are assumed to range over the object constants used in the program (function symbols are not allowed in the current version), and "built-in" predicates can be used to express restrictions on the values of variables. For instance, program (6)–(10) can be turned into an input for DLV as follows:

```
in(U,V); -in(U,V) :- edge(U,V).

:- in(U,V), in(U,W), V!=W.
:- in(U,W), in(V,W), U!=V.

reachable(U) :- in(0,U).
reachable(V) :- reachable(U), in(U,V).

:- not reachable(U), #int(U).
```

Appended to this should be the description of graph G in the form of a definition of `edge/2`. Built-in predicate `#int/1` expresses membership in an initial segment of nonnegative integers; the length of the segment is specified on the command line when DLV is invoked.

The input language of system SMODELS is less expressive in that the program is supposed to be nondisjunctive (besides $l = k$, $k \leq 1$ in every rule (4) is required). Also, the current version of SMODELS does not support classical negation.[5]

For some purposes, the unavailability of disjunctive rules is an essential limitation. But a disjunctive rule of the form

$$p; \neg p \leftarrow$$

can be always replaced by a pair of rules similar to (2):

$$\begin{array}{l} p \leftarrow not\ \neg p, \\ \neg p \leftarrow not\ p \end{array}$$

(see [3]). Furthermore, the use of classical negation can be always avoided, because a negative literal $\neg p$ can be replaced by a new atom p' provided that the constraint

$$\leftarrow p, p' \tag{12}$$

is added to the program. Using these two tricks, programs (6)–(10) and (11) can be turned into programs containing neither ; nor $\neg$ (in these two cases, adding constraints (12) is not even necessary). Then SMODELS can be used

```
Initial state:              Goal:

                             3   6
   1   3   5                 2   5
   2   4   6                 1   4
-------------              ---------
```

Figure 2: A planning problem in the blocks world

instead of DLV to find Hamiltonian cycles and configurations of blocks. Actually, SMODELS allows the user to specify such modified programs in a concise way, without introducing the new atoms explicitly. The same can be done with the planning program discussed in Section 5.

5 Planning

In a planning problem, we look for a sequence of actions that leads from a given initial state to a given goal state.[6] Turning one configuration of blocks into another (Figure 2) is a planning problem. We will use this example here to illustrate some ideas of answer set planning.

To specify a planning problem completely, we need to say which actions are allowed in a plan. When a block is "clear" (there is nothing on top of it), we assume that it can be placed on top of any tower of blocks or on the table. Thus moving a block located in the middle of a tower, along with everything on top of it, is prohibited. This is because we think of moving performed by a robot that grasps blocks from above.

To make the blocks world more interesting, let us assume that the robot has two grippers that can move blocks independently. For instance, in the initial state shown in Figure 2, blocks 1 and 3 can be moved concurrently. Since a block can be moved only when it is clear, moving blocks 1 and 2 concurrently is impossible. (That would require a better synchronization between the grippers than the robot is supposed to have.) Besides, we assume that the robot is unable to move b onto b' if b' is being moved also. Even with these restrictions, the planning problem above can be solved in 3 steps:

1. Place blocks 1 and 3 on the table.
2. Place block 2 on block 1 and block 5 on block 4.
3. Place block 3 on block 2 and block 6 on block 5.

Research on planning has two components: representation (design of declarative languages for specifying planning problems) and search (design of planning algorithms).

An important class of planning algorithms is based on the idea of reducing a planning problem to the problem of finding a satisfying interpretation for a set of propositional formulas. This is known as satisfiability planning [9]. The latest satisfiability planning system created by the inventors of this method is available at `http://www.research.att.com/~kautz/blackbox` . Answer set planning differs from satisfiability planning in that it uses logic programming rules instead of propositional formulas. An important advantage of answer set planning is that the representation of properties of actions is easier when logic programs are used instead of classical logic, in view of the nonmonotonic character of negation as failure.[7]

The idea of answer set planning is due to Subrahmanian and Zaniolo [20], and the results of computational experiments that use SMODELS to compute answer sets are reported in [2, 16]. The logic programming representation of the blocks world below follows [3] and is based on some of the ideas of [15, 21, 10, 11].

The key element of answer set planning is the representation of a dynamic domain such as the blocks world in the form of a "history program"—a program whose answer sets represent possible "histories," or evolutions of the system, over a fixed time interval. In the next section we show how to write such a program for the blocks world. The answer sets for that program represent possible evolutions of the blocks world over the time interval $0, \dots, T$ for a fixed positive integer T. A history of the blocks world is characterized by the truth values of atoms of two kinds: $on(b, l, t)$ ("block b is on location l at time t") and $move(b, l, t)$ ("block b is moved to location l between times t and $t + 1$"). Here b ranges over blocks and l ranges over locations, as in Section 3; the time variable t ranges over the time instants $0, \dots, T$, except that the atoms $move(b, l, t)$ are introduced only for $t \neq T$.

When such a program is available, we can find a plan of length T that solves a given planning problem, or establish that there is no such plan, in the following way. The history program is extended by the constraints representing the initial state and the goal state of the problem. In the example above, $T = 3$ and the constraints are

$$\begin{array}{l} \leftarrow not\ on(1,2,0) \\ \leftarrow not\ on(2,table,0) \\ \leftarrow not\ on(3,4,0) \\ \dots \end{array} \tag{13}$$

$$\begin{array}{l} \leftarrow not\ on(1,table,3) \\ \leftarrow not\ on(2,1,3) \\ \leftarrow not\ on(3,2,3) \\ \dots \end{array} \tag{14}$$

The answer sets for the extended program correspond to the plans of length T that lead from the initial state to the goal state. A planner would invoke a system for computing answer sets to find an answer set X for the extended program, and then return the list of all atoms in X that represent actions (in the case of the blocks world, the atoms beginning with $move$).

To find a plan consisting of the smallest possible number of steps, the planner would invoke a system for computing answer sets several times, with different values of T, using binary search if desired. (A similar process is used in satisfiability planning.)

6 A History Program for the Blocks World

Since actions in the blocks world domain are deterministic, a history is completely defined by its starting point—the truth values of atoms $on(b,l,0)$—and the actions that have been executed—the truth values of $move(b,l,t)$. We begin the construction of the history program by allowing these atoms to have arbitrary values:

$$on(b,l,0); \neg on(b,l,0) \leftarrow \ , \tag{15}$$

$$move(b,l,t); \neg move(b,l,t) \leftarrow \qquad (t \neq T). \tag{16}$$

Some of the other rules will encode what we know about the effects of moving blocks by characterizing the truth values of the atoms $on(b,l,t)$ with $t \neq 0$. We will also have rules expressing what we know about the executability of actions and about possible configurations of blocks.

The main rule tells us that moving a block b onto a location l causes b to be on l:

$$on(b,l,t+1) \leftarrow move(b,l,t) \qquad (t \neq T). \tag{17}$$

But this is not all that needs to be said about the configuration of blocks at time $t+1$. Two things are still missing.

First, we need to say that the blocks that are not being moved remain where they were. In other words, we need to solve the frame problem for the blocks world. This can be done by postulating

$$on(b,l,t+1) \leftarrow on(b,l,t), not\ \neg on(b,l,t+1) \qquad (t \neq T). \tag{18}$$

The idea of this solution to the frame problem goes back to [17].

Second, we need to ensure that when a block is moved to a new location, it disappears from the old one. An elegant way to do this does not even mention moving blocks. We can use the rule

$$\neg on(b,l,t) \leftarrow on(b,l',t) \qquad (l \neq l') \tag{19}$$

which is similar to the second line of (11), except that now *on* has a time argument, and that this rule is not a constraint.[8]

The history program includes also the constraints expressing the assumptions about the executability of *move* stated in Section 5. A block can be moved only when it is clear:

$$\leftarrow move(b,l,t), on(b',b,t) \qquad (t \neq T) \tag{20}$$

and when the target is not moving:

$$\leftarrow move(b, b', t), move(b', l, t) \qquad (t \neq T). \tag{21}$$

The robot has only two grippers:

$$\begin{array}{l}\leftarrow move(b, l, t), move(b', l', t), move(b'', l'', t) \\ \quad (\langle b, l\rangle \neq \langle b', l'\rangle,\ \langle b, l\rangle \neq \langle b'', l''\rangle,\ \langle b', l'\rangle \neq \langle b'', l''\rangle,\ t \neq T).\end{array} \tag{22}$$

Finally, the history program includes the last two of the three constraints on possible configurations of blocks from (11), applied to every time instant:

$$\leftarrow on(b, b'', t), on(b', b'', t) \qquad (b \neq b'), \tag{23}$$

$$\begin{array}{l}supported(b, t) \leftarrow on(b, table, t) \\ supported(b, t) \leftarrow supported(b', t), on(b, b', t) \\ \leftarrow not\ supported(b, t).\end{array} \tag{24}$$

There is no need to include the counterpart (26) of the first of those constraints, because we have already included a "stronger" rule (19).

History program (15)–(24) can be used to solve planning problems in the blocks world as described at the end of the previous section.[9] Information on the performance of several answer set solvers on a few planning problems in the blocks world can be found at `http://www.cs.utexas.edu/users/esra/experiments/experiments.html` .

7 Expressiveness

As a formalism for representing properties of actions, logic programming is remarkably expressive. Here are three examples of how it handles relatively complex cases—all of them are beyond the expressive capabilites of the classical STRIPS language [5].

Conditional Effects. After toggling a switch, the light in the room is on if it was off before, and the other way around:

$$\begin{array}{l}light\text{-}on(t+1) \leftarrow toggle(t), \neg light\text{-}on(t), \\ \neg light\text{-}on(t+1) \leftarrow toggle(t), light\text{-}on(t).\end{array}$$

Implicit Effects. When I walk to another place, this action indirectly affects the locations of the objects that I have in my pockets. Any other action that changes my location—running, or riding a bus, or driving a car—has the same implicit effect. Describing implicit effects of actions is known as the "ramification problem." In logic programming, we can solve the ramification problem by writing rules that do not mention actions, such as (19) or

$$at(x, l, t) \leftarrow at(I, l, t), in\text{-}my\text{-}pocket(x, t).$$

Implicit Preconditions. Rules of the history program above specify three cases when moving blocks is impossible: when the block to be moved is not clear (20), when the target is a block that is being moved (21), and when more than two blocks are being moved at once (22). Is it possible, in the initial state shown in Figure 2, to place block 1 on top of block 6? That would not violate any of the three explicitly stated preconditions. Nevertheless, the answer is no, unless block 5, which was initially on top of block 6, is being moved at the same time. This fact is a consequence of constraint (23): two blocks cannot be located on top of the same block.

The expressiveness of logic programming as a language for describing actions is illustrated also by the fact that any definite action description in action language $\mathcal{C}$ [8] can be translated into logic programming [11].

8 Why Answer Sets?

In the introduction, the notion of an answer set was mentioned as one of three mathematical concepts that can be used to give a declarative meaning to negation as failure, the other two being completion and the well-founded model. Can any of the other two concepts serve as a basis for a logic programming style similar to answer set programming?

The well-founded model classifies ground queries into three groups: true, false, and undefined. It is useful as a specification for query answering systems that are similar to Prolog but have better termination properties, such as XSB (`http://www.cs.sunysb.edu/~sbprolog/xsb-page.html`). Given program (2), and either p or q as a query, such a system can be expected to terminate and return the answer *undefined*. But the information about a program provided by its well-founded model does not seem to be adequate for anything similar to answer set programming. It does not allow us, for instance, to distinguish between programs (2) and

$$\begin{array}{l} p \leftarrow not\ p, \\ q \leftarrow not\ q \end{array} \tag{25}$$

—both have the same well-founded model. The answer set approach to negation as failure recognizes a big difference between the two programs: the latter has no answer sets.

With the completion semantics, the situation is different. The definition of completion is not applicable to disjunctive programs, nor to programs with two kinds of negation. But, as discussed at the end of Section 4, these are not very severe limitations. In particular, classical negation can be always eliminated at the cost of adding new atoms and constraints (12). Also, completion has a counterpart defined for programs with classical negation—"literal completion."[10]

Consider a finite nondisjunctive program Π without classical negation, such as (2). For simplicity, assume also that Π does not contain constraints.

Two basic facts are known about the relationship between the answer sets for Π and the models of the completion of Π (see [4]). If we identify an interpretation in the sense of classical logic with the set of atoms that are true in that interpretation then

- every answer set for Π is a model of the completion of Π, and
- if Π is tight[11] then, conversely, every model of the completion of Π is an answer set for Π.

For instance, program (2) is tight, and its answer sets are the models of its completion $p \equiv q$. Program (25) is tight also; its completion $p \equiv \neg p$, $q \equiv \neg q$ is inconsistent.

It follows that answer sets for a tight program can be computed by finding models of its completion. This is, essentially, how CCALC (see Introduction) operates when used to find answer sets for a logic program. It forms the (literal) completion of the given program Π and submits the result to a propositional solver. Thus CCALC can be used to find answer sets for Π only if Π is tight, or if we know in some other way that every model of the completion of Π is an answer set. For such programs, "completion programming" differs from answer set programming only in that it uses a different computational mechanism for computing answer sets.

On the other hand, if the completion of a program has a model that is not an answer set then this model is usually "unwanted"—it does not reflect the intended meaning of the program. This is the case, actually, for all three examples of answer set programming given in this paper. Generally, the completion of (the result of eliminating disjunctive rules and classical negation from) program (6)–(10) has models that do not correspond to Hamiltonian paths. The completion of (11) has models that do not correspond to valid configurations of blocks. The completion of the history program from Section 6 has models that do not correspond to possible evolutions of the blocks world. In this sense, "completion programming" is not a good idea.

It is interesting that, in each of the last two examples, a small modification of the part of the program that involves *supported* would help us get rid of the unwanted models of completion. In the case of the history program, it is sufficient to replace (24) by the rules

$$\begin{array}{l} above(b,l,t) \leftarrow on(b,l,t) \\ above(b,l,t) \leftarrow above(b,b',t), above(b',l,t) \\ \leftarrow above(b,b,t) \\ \leftarrow not\ \ above(b,table,t). \end{array}$$

Although the resulting program is not tight, every model of its completion is an answer set (Norman McCain and Hudson Turner, personal communication). This modification may be helpful when CCALC is used for planning in the blocks world.

Acknowledgements

This report is an outcome of many enjoyable discussions with Esra Erdem, Michael Gelfond, Nicola Leone, Victor Marek, Norman McCain, Ilkka Niemelä, Teodor Przymusinski, Miroslaw Truszczyński and Hudson Turner. This work was partially supported by National Science Foundation under grant IIS-9732744.

Notes

1. Answer set programming is also known as "stable model programming." The term "stable model" was used in place of "answer set" in a preliminary publication on the answer set semantics [6]. That old terminology stems from the view that a rule in logic programming is merely a clause in the sense of classical logic, with one of its positive literals designated as the "head." If so, we can talk about models of a logic program in the sense of classical logic, and a semantics of negation as failure can be defined by specifying an additional condition on models—"stability." For instance, program (1), according to this view, is simply the disjunction $p \vee q$ written in the logic programming notation. The usual truth table for disjunction shows that this formula has 3 models: $\{p\}$, $\{q\}$ and $\{p, q\}$. The first model is "stable"; the other two are not.

This view does not extend, however, to programs with two kinds of negation that are defined in [7] and are used in this paper. It is better to think of *inference rules*, rather than clauses, as the classical logic counterpart of logic programming rules. A rule with negation as failure is similar to a default in the sense of [17] which combines an inference rule with a list of "justifications." This is actually more than an analogy: there is a precise relationship between answer sets for programs and extensions for default theories [13, 7].

2. See

`http://www.tcs.hut.fi/Software/smodels/`
`http://www.dbai.tuwien.ac.at/proj/dlv/`
`http://www.cs.engr.uky.edu/~lpnmr/DeReS.html`
`http://www.cs.utexas.edu/users/mccain/cc`

3. Some authors call the second kind of negation "strong" or "explicit." It is natural to think of this negation as "taken from classical logic" in connection with the relationship between logic programs and default theories mentioned at the end of Note 1. Generally, the conclusion of a default is a formula of classical logic. If we require that the conclusion be a literal, and limit the syntactic form of the other parts of a default (prerequisite and justifications) in a similar way, we will arrive at the class of defaults corresponding to nondisjunctive logic programs.

4. Some expositions of the answer set semantics allow answer sets to be inconsistent, but different papers treat this case in different ways. It is best to eliminate inconsistent answer sets altogether.

5. On the other hand, SMODELS permits function symbols in schematic rules, which is often convenient. System CCALC permits function symbols as well. The problem with the use of function symbols in answer set programming is that it can make the set of ground atoms infinite. The user of a system for computing answer sets should be careful not to provide an input that has an infinite answer set—such a set cannot be returned as output!

6. More generally, a planning problem may involve a set of initial conditions instead of a specific initial state, and a set of goal conditions instead of a specific goal state. Instead of a sequence of actions, we may be looking for a "strategy" built from actions using conditional operators and loops.

7. In fact, the emergence of formal nonmonotonic reasoning twenty years ago was motivated to a large degree by the need to solve a knowledge representation problem related to properties of actions—the frame problem. A history of attempts to solve the frame problem, in their relation to nonmonotonic logics and logic programming, is described in [18].

8. The difference between (19) and the constraint

$$\leftarrow on(b,l',t), on(b,l,t) \qquad (l \neq l') \tag{26}$$

is similar to the difference between "ramification constraints" and "qualification constraints" in the sense of [12].

9. In the extended program described at the end of Section 5, the part consisting of disjunctive rules (15) and constraints (13) can be equivalently replaced by the initial conditions represented as facts:

$$\begin{array}{l} on(1,2,0) \leftarrow \\ on(2,table,0) \leftarrow \\ on(3,4,0) \leftarrow \\ \dots \end{array}$$

More importantly, the program can be further extended by constraints that limit the set of executable sequences of actions without changing the class of solvable planning programs. In this way, domain-dependent control information can be conveyed to the answer set solver, which may help it find a solution faster. For instance, we can tell the solver that *if a block and all blocks under it are in their final positions then that block should not be moved* by adding a constraint that makes such a move impossible.

10. Literal completion was defined for a language of causal theories in [15] and extended to nondisjunctive logic programs in [11]. It does not introduce new atoms. The models of the literal completion of a program are

similar to its *complete* answer sets—those that contain one element of every complementary pair of literals.

11. We say that Π is *tight* if there exists a function λ from atoms to ordinals such that, for every rule

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$$

in Π,

$$\lambda(A_1), \ldots, \lambda(A_m) < \lambda(A_0).$$

This condition does not impose any restrictions on $A_{m+1}, \ldots, A_n$. In [4], such programs are called "positive-order-consistent."

References

[1] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[2] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In *Proc. European Conf. on Planning 1997*, pages 169–181, 1997.

[3] Esra Erdem and Vladimir Lifschitz. Transformations of logic programs related to causality and planning. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf.*, 1999. To appear.

[4] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[5] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[6] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[7] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[8] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630, 1998.

[9] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.

[10] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.

[11] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf.*, 1999. To appear.

[12] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4:655–678, 1994.

[13] Victor Marek and Mirosław Truszczyński. Stable semantics for logic programs and default theories. In *Proc. North American Conf. on Logic Programming*, pages 243–256, 1989.

[14] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[15] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[16] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 1999.

[17] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[18] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia.* MIT Press, 1997.

[19] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proc. First Int'l Workshop on Practical Aspects of Declarative Languages (Lecture Notes in Computer Science 1551)*, pages 305–319. Springer-Verlag, 1998.

[20] V.S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proc. ICLP-95*, 1995.

[21] Hudson Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.

[22] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, pages 620–650, 1990.