Representing Actions in Extended Logic Programming

Michael Gelfond
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968

Vladimir Lifschitz
Department of Computer Sciences
and Department of Philosophy
University of Texas at Austin
Austin, TX 78712

Abstract

We represent properties of actions in a logic programming language that uses both classical negation and negation as failure. The method is applicable to temporal projection problems with incomplete information, as well as to reasoning about the past. It is proved to be sound relative to a semantics of action based on states and transition functions.

1 Introduction

This paper extends the work of Eshghi and Kowalski [4], Evans [5] and Apt and Bezem [1] on representing properties of actions in logic programming languages with negation as failure.

Our goal is to overcome some of the limitations of the earlier work. The existing formalizations of action in logic programming are adequate for only the simplest kind of temporal reasoning—"temporal projection." In a temporal projection problem, we are given a description of the initial state of the world, and use properties of actions to determine what the world will look like after a series of actions is performed. Moreover, the existing formalizations can be used for temporal projection only in the cases when the given description of the initial state is complete. The reason for that is that traditional logic programming languages automatically apply the "closed world assumption" to each predicate.

We are interested here in temporal reasoning of a more general kind, when the values of some fluents[1] in one or more situations are given, and the goal is to derive other facts about the values of fluents. Besides temporal projection, this class of reasoning problems includes, for instance, the cases when we want to use information about the current state of the world for answering questions about the past[2]. The view of logic programming accepted in this paper is strictly declarative. The adequacy of a representation of a body of knowledge in a logic programming language means, to us, adequacy with respect to the declarative semantics of that language. It is interesting to find out, of course, whether any of the currently available query evaluation procedures will actually terminate if used for answering questions on the basis of our representation, and how fast, but those are secondary issues. In fact, the language of "extended logic programs" used in this paper is a subset of the system of default logic from [16], and our work can be viewed as a development of the approach to temporal reasoning based on nonnormal defaults [15].

Two parts of this paper may be of more general interest.

First, we introduce here a simple declarative language for describing actions, called $\mathcal{A}$. Traditionally, ideas on representing properties of actions in classical logic or nonmonotonic formalisms are explained on specific examples, such as the "Yale shooting problem" from [9]. Competing approaches are evaluated and compared in terms of their ability to handle such examples. Sandewall [17] provides a systematic comparison of the most important approaches by applying them to a rather long series of problems of this kind. We propose to supplement the use of examples by a different method. A particular methodology for representing action can be formally described as a translation from $\mathcal{A}$, or from a subset or a superset of $\mathcal{A}$, into a "target language"—for instance, into a language based on classical logic or on circumscription. The soundness and completeness of each particular translation become precise mathematical questions; the possibilities and limitations of each methodology can be described in terms of the "dialects" of $\mathcal{A}$ to which it is applicable. Our method for describing properties of actions in logic programming is presented here as a translation from $\mathcal{A}$ into the language of extended logic programs, and the soundness of this translation is the main technical result of the paper.

Second, the proof of the main theorem depends on a relationship between stable models [7] and signings [11], that may be interesting as a part of the general theory of logic programming.

The language $\mathcal{A}$ is introduced in Section 2, and Section 3 is a brief review of extended logic programs. Our translation from $\mathcal{A}$ into logic programming is defined in Section 4, and the soundness theorem is stated in Section 5. Section 6 contains the lemmas that relate stable models to signings. The proof of the soundness theorem can be found in the complete version of this paper.

2 A Language for Describing Actions

A description of an action domain in the language $\mathcal{A}$ consists of "propositions" of two kinds. A "v-proposition" specifies the value of a fluent in a particular situation—either in the initial situation, or after performing a sequence of actions. An "e-proposition" describes the effect of an action on a fluent.

We begin with two disjoint nonempty sets of symbols, called *fluent names* and *action names*. A *fluent expression* is a fluent name possibly preceded by $\neg$. A *v-proposition* is an expression of the form

$$F \textbf{ after } A_1; \ldots; A_m, \tag{1}$$

where F is a fluent expression, and $A_1, \ldots, A_m$ $(m \geq 0)$ are action names. If $m = 0$, we will write (1) as

$$\textbf{initially } F.$$

An *e-proposition* is an expression of the form

$$A \textbf{ causes } F \textbf{ if } P_1, \ldots, P_n, \tag{2}$$

where A is an action name, and each of $F, P_1, \ldots, P_n$ $(n \geq 0)$ is a fluent expression. About this proposition we say that it *describes the effect of A on F*, and that $P_1, \ldots, P_n$ are its *preconditions*. If $n = 0$, we will drop **if** and write simply

$$A \textbf{ causes } F.$$

A *proposition* is a v-proposition or an e-proposition. A *domain description*, or simply *domain*, is a set of propositions (not necessarily finite).

Example 1. The Fragile Object domain, motivated by an example from [18], has the fluent names *Holding*, *Fragile* and *Broken*, and the action *Drop*. It consists of two e-propositions:

$$\begin{array}{l} \mathit{Drop} \textbf{ causes } \neg \mathit{Holding} \textbf{ if } \mathit{Holding}, \\ \mathit{Drop} \textbf{ causes } \mathit{Broken} \textbf{ if } \mathit{Holding}, \mathit{Fragile}. \end{array}$$

Example 2. The Yale Shooting domain, motivated by the example from [9] mentioned above, is defined as follows. The fluent names are *Loaded* and *Alive*; the action names are *Load*, *Shoot* and *Wait*. The domain is characterized by the propositions

$$\begin{array}{l} \textbf{initially } \neg \mathit{Loaded}, \\ \textbf{initially } \mathit{Alive}, \\ \mathit{Load} \textbf{ causes } \mathit{Loaded}, \\ \mathit{Shoot} \textbf{ causes } \neg \mathit{Alive} \textbf{ if } \mathit{Loaded}, \\ \mathit{Shoot} \textbf{ causes } \neg \mathit{Loaded}. \end{array}$$

Example 3. The Murder Mystery domain, motivated by an example from [2], is obtained from the Yale Shooting domain by substituting

$$\neg \mathit{Alive} \textbf{ after } \mathit{Shoot}; \mathit{Wait} \tag{3}$$

for the proposition **initially** $\neg \mathit{Loaded}$.

Example 4. The Stolen Car domain, motivated by an example from [10], has one fluent name *Stolen* and one action name *Wait*, and is characterized by two propositions:

$$\begin{array}{l} \textbf{initially } \neg \mathit{Stolen}, \\ \mathit{Stolen} \textbf{ after } \mathit{Wait}; \mathit{Wait}; \mathit{Wait}. \end{array}$$

To describe the semantics of $\mathcal{A}$, we will define what the "models" of a domain description are, and when a v-proposition is "entailed" by a domain description.

A *state* is a set of fluent names. Given a fluent name F and a state σ, we say that F *holds* in σ if $F \in \sigma$; $\neg F$ *holds* in σ if $F \notin \sigma$. A *transition function* is a mapping Φ of the set of pairs (A, σ), where A is an action name and σ is a state, into the set of states. A *structure* is a pair (σ_0, Φ), where σ_0 is a state (the *initial state* of the structure), and Φ is a transition function.

For any structure M and any action names $A_1, \ldots, A_m$, by $M^{A_1;\ldots;A_m}$ we denote the state

$$\Phi(A_m, \Phi(A_{m-1}, \ldots, \Phi(A_1, \sigma_0)\ldots)),$$

where Φ is the transition function of M, and σ_0 is the initial state of M. We say that a v-proposition (1) is *true* in a structure M if F holds in the state $M^{A_1;\ldots;A_m}$, and that it is *false* otherwise. In particular, a proposition of the form **initially** F is true in M iff F holds in the initial state of M.

A structure (σ_0, Φ) is a *model* of a domain description D if every v-proposition from D is true in (σ_0, Φ), and, for every action name A, every fluent name F, and every state σ, the following conditions are satisfied:

(i) if D includes an e-proposition describing the effect of A on F whose preconditions hold in σ, then $F \in \Phi(A, \sigma)$;

(ii) if D includes an e-proposition describing the effect of A on $\neg F$ whose preconditions hold in σ, then $F \notin \Phi(A, \sigma)$;

(iii) if D does not include such e-propositions, then $F \in \Phi(A, \sigma)$ iff $F \in \sigma$.

It is clear that there can be at most one transition function Φ satisfying conditions (i)–(iii). Consequently, different models of the same domain description can differ only by their initial states. For instance, the Fragile Object domain (Example 1) has 8 models, whose initial states are the subsets of

$$\{\mathit{Holding}, \mathit{Fragile}, \mathit{Broken}\};$$

in each model, the transition function is defined by the equation

$$\Phi(Drop, \sigma) = \begin{cases} \sigma \setminus \{Holding\} \cup \{Broken\}, & \text{if } Fragile \in \sigma, \\ \sigma \setminus \{Holding\}, & \text{otherwise.} \end{cases}$$

A domain description is *consistent* if it has a model, and *complete* if it has exactly one model. The Fragile Object domain is consistent, but incomplete. The Yale Shooting domain (Example 2) is complete; its only model is defined by the equations

$$\begin{aligned} &\sigma_0 = \{Alive\}, \\ &\Phi(Load, \sigma) = \sigma \cup \{Loaded\}, \\ &\Phi(Shoot, \sigma) = \begin{cases} \sigma \setminus \{Loaded, Alive\}, & \text{if } Loaded \in \sigma, \\ \sigma, & \text{otherwise,} \end{cases} \\ &\Phi(Wait, \sigma) = \sigma. \end{aligned}$$

The Murder Mystery domain (Example 3) is complete also; it has the same transition function as Yale Shooting, and the initial state $\{Loaded, Alive\}$. The Stolen Car domain (Example 4) is inconsistent.

A v-proposition is *entailed* by a domain description D if it is true in every model of D. For instance, Yale Shooting entails

$$\neg Alive \textbf{ after } Load; Wait; Shoot.$$

Murder Mystery entails, among others, the propositions

$$\textbf{initially } Loaded$$

and

$$\neg Alive \textbf{ after } Wait; Shoot.$$

Note that the last proposition differs from (3) by the order in which the two actions are executed. This example illustrates the possibility of reasoning about alternative "possible futures" of the initial situation.

Although the language $\mathcal{A}$ is adequate for formalizing several interesting domains, its expressive possibilities are rather limited. The only fluents available in $\mathcal{A}$ are propositional ones. It is impossible to say in $\mathcal{A}$ that certain fluents are related in some way (for instance, that $F1$ and $F2$ cannot be simultaneously true); for this reason, actions described in $\mathcal{A}$ have no indirect effects ("ramifications"). Every action is assumed to be executable in any situation. We cannot talk about the duration of actions, or describe actions that are nondeterministic, or are performed concurrently. The inconsistency of the Stolen Car domain illustrates the fact that $\mathcal{A}$ cannot be used for representing "causal anomalies," or "miracles" [12][3]. Defining and studying extensions of $\mathcal{A}$ is a topic for future work.

The entailment relation of $\mathcal{A}$ is nonmonotonic, in the sense that adding an e-proposition to a domain description D may nonmonotonically change the set of propositions entailed by D. (This cannot happen when a v-proposition is added.) For this reason, a modular translation from $\mathcal{A}$ into another declarative language (that is, a translation that processes propositions one by one) can be reasonably adequate only if this other language is nonmonotonic also.

3 Extended Logic Programs

Representing incomplete information in traditional logic programming languages is difficult, because their semantics is based on the automatic application of the closed world assumption to all predicates. Given a ground query, a traditional logic programming system can produce only one of two answers, *yes* or *no*; it will never tell us that the truth value of the query cannot be determined on the basis of the information included in the program.

Extended logic programs, introduced in [8], are, in this sense, different. The language of extended programs distinguishes between negation as failure *not* and classical negation $\neg$. The general form of an extended rule is

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n, \tag{4}$$

where each L_i is a literal, that is, an atom possibly preceded by $\neg$. An extended program is a set of such rules. Here is an example:

$$\begin{array}{l} p, \\ \neg q \leftarrow p, \\ r \leftarrow \neg p, \\ t \leftarrow \neg q, not\ s, \\ u \leftarrow not\ \neg u. \end{array} \tag{5}$$

Intuitively, these rules say:

p is true;
q is false if p is true;
r is true if p is false;
t is true if q is false and there is no evidence that s is true;
u is true if there is no evidence that it is false.

The answers that an implementation of this language is supposed to give to the ground queries are:

p :	*yes*,
q :	*no*,
r :	*unknown*,
s :	*unknown*,
t :	*yes*,
u :	*yes*.

The semantics of extended logic programs defines when a set of ground literals is an *answer set* of a program [8]. For instance, the program (5) has one answer set, $\{p, \neg q, t, u\}$.

The answer sets of a program can be easily characterized in terms of default logic. We will identify the rule (4) with the default

$$L_1 \wedge \ldots \wedge L_m \;:\; \overline{L_{m+1}}, \ldots, \overline{L_n} \,/\, L_0 \tag{6}$$

($\overline{L}$ stands for the literal complementary to L). Thus every extended program can be viewed as a default theory. The answer sets of a program are simply its extensions in the sense of default logic, intersected with the set of ground literals ([8], Proposition 3).

4 Describing Actions by Logic Programs

Now we are ready to define the translation π from $\mathcal{A}$ into the language of extended programs.

About two different e-propositions we say that they are *similar* if they differ only by their preconditions. Our translation method is defined for any domain description that does not contain similar e-propositions. This condition prohibits, for instance, combining in the same domain such propositions as

$$\begin{array}{l} \textit{Shoot}\ \textbf{causes}\ \neg \textit{Alive}\ \textbf{if}\ \textit{Loaded}, \\ \textit{Shoot}\ \textbf{causes}\ \neg \textit{Alive}\ \textbf{if}\ \textit{VeryNervous}. \end{array}$$

(*VeryNervous* refers to the victim, of course—not to the gun.)

Let D be a domain description without similar e-propositions. The corresponding logic program πD uses variables of three sorts: *situation* variables $s, s', \ldots$, *fluent* variables $f, f', \ldots$, and *action* variables $a, a', \ldots$[4]. Its only situation constant is $S0$; its fluent constants and action constants are, respectively, the fluent names and action names of D. There are also some predicate and function symbols; the sorts of their arguments and values will be clear from their use in the rules below.

The program πD will consist of the translations of the individual propositions from D and the four standard rules:

$$\begin{array}{l} \textit{Holds}(f, \textit{Result}(a, s)) \leftarrow \textit{Holds}(f, s), \textit{not Noninertial}(f, a, s), \\ \neg \textit{Holds}(f, \textit{Result}(a, s)) \leftarrow \neg \textit{Holds}(f, s), \textit{not Noninertial}(f, a, s), \end{array} \tag{7}$$

$$\begin{array}{l} \textit{Holds}(f, s) \leftarrow \textit{Holds}(f, \textit{Result}(a, s)), \textit{not Noninertial}(f, a, s), \\ \neg \textit{Holds}(f, s) \leftarrow \neg \textit{Holds}(f, \textit{Result}(a, s)), \textit{not Noninertial}(f, a, s). \end{array} \tag{8}$$

These rules are motivated by the "commonsense law of inertia," according to which the value of a fluent after performing an action is normally the same as before. The rules (7) allow us to apply the law of inertia in reasoning "from the past to the future": the first—when a fluent is known to be true in the past, and the second—when it is known to be false. The rules (8) play the same role for reasoning "from the future to the past." The auxiliary predicate *Noninertial* is essentially an "abnormality predicate" [13].

Now we will define how π translates v-propositions and e-propositions. The following notation will be useful: For any fluent name F,

$$|F| \text{ is } F, \quad |\neg F| \text{ is } F,$$

and, if t is a situation term, $Holds(\neg F, t)$ stands for $\neg Holds(F,t)$. The last convention allows us to write $Holds(F,t)$ even when F is a fluent name preceded by $\neg$. Furthermore, if $A_1, \ldots, A_m$ are action names, $[A_1; \ldots; A_m]$ stands for the term

$$Result(A_m, Result(A_{m-1}, \ldots, Result(A_1, S0) \ldots)).$$

It is clear that every situation term without variables can be represented in this form.

The translation of a v-proposition (1) is

$$Holds(F, [A_1; \ldots; A_m]). \tag{9}$$

For instance, π(**initially** *Alive*) is

$$Holds(Alive, S0),$$

and $\pi(\neg$*Alive* **after** *Shoot*) is

$$\neg Holds(Alive, Result(Shoot, S0)).$$

The translation of an e-proposition (2) consists of $2n+2$ rules. The first of them is

$$Holds(F, Result(A, s)) \leftarrow Holds(P_1, s), \ldots, Holds(P_n, s). \tag{10}$$

It allows us to prove that F will hold after A, if the preconditions are satisfied. The second rule is

$$Noninertial(|F|, A, s) \leftarrow not\ \overline{Holds(P_1, s)}, \ldots, not\ \overline{Holds(P_n, s)} \tag{11}$$

($\overline{Holds(P_i, s)}$ is the literal complementary to $Holds(P_i, s)$.) It disables the inertia rules (7), (8) in the cases when f can be affected by a. Without this rule, the program would be contradictory: We would prove, using a rule of the form (10), that an unloaded gun becomes loaded after the action *Load*, and also, using the second of the rules (7), that it remains unloaded!

Note the use of *not* in (11). We want to disable the inertia rules not only when the preconditions for the change in the value of F are known to hold, but whenever *there is no evidence that they do not hold.* If, for instance, we do not know whether *Loaded* currently holds, then we do not want to conclude by inertia that the value of *Alive* will remain the same after *Shoot*. We cannot draw any conclusions about the new value of *Alive*. If we replaced the body of (11) by $Holds(P_1, s), \ldots, Holds(P_n, s)$, the translation would become unsound.

Besides (10) and (11), the translation of (2) contains, for each i ($1 \leq i \leq n$), the rules

$$Holds(P_i, s) \leftarrow \overline{Holds(F, s)}, Holds(F, Result(A, s)) \tag{12}$$

and

$$\begin{aligned}\overline{Holds(P_i, s)} \leftarrow\ & \overline{Holds(F, Result(A, s))},\\ & Holds(P_1, s), \ldots, Holds(P_{i-1}, s),\\ & Holds(P_{i+1}, s), \ldots, Holds(P_n, s).\end{aligned} \tag{13}$$

The rules (12) justify the following form of reasoning: If the value of F has changed after performing A, then we can conclude that the preconditions were satisfied when A was performed. These rules would be unsound in the presence of similar propositions. The rules (13) allow us to conclude that a precondition was false from the fact that performing an action did not lead to the result described by an effect axiom, while all other preconditions were true.

We will illustrate the translation process by applying it to Yale Shooting (Example 2). The translation of that domain includes, in addition to (7) and (8), the following rules:

$Y1$. $\neg Holds(Loaded, S0)$.

$Y2$. $Holds(Alive, S0)$.

$Y3$. $Holds(Loaded, Result(Load, s))$.

$Y4$. $Noninertial(Loaded, Load, s)$.

$Y5$. $\neg Holds(Alive, Result(Shoot, s)) \leftarrow Holds(Loaded, s)$.

$Y6$. $Noninertial(Alive, Shoot, s) \leftarrow$ *not* $\neg Holds(Loaded, s)$.

$Y7$. $Holds(Loaded, s) \leftarrow Holds(Alive, s), \neg Holds(Alive, Result(Shoot, s))$.

$Y8$. $\neg Holds(Loaded, s) \leftarrow Holds(Alive, Result(Shoot, s))$.

$Y9$. $\neg Holds(Loaded, Result(Shoot, s))$.

$Y10$. $Noninertial(Loaded, Shoot, s)$.

It is instructive to compare this set of rules with the formalization of Yale Shooting given by Apt and Bezem [1], who were only interested in temporal projection problems, and did not use classical negation. Instead of our four inertia rules, they have one, corresponding to the first of the rules (7). In addition, their program includes counterparts of $Y2$, $Y3$, $Y5$ and $Y6$. It does not tell us whether *Loaded* holds in the initial situation, but the negative answer to this question follows by the closed world assumption. Their rule corresponding to $Y5$ does not have $\neg$ in the head, of course; instead, the new fluent *Dead* is used. In their counterpart of $Y6$, the combination *not* $\neg$ is missing; this does not lead to any difficulties, because the closed world assumption is implicitly postulated.

5 Soundness Theorem

We say that a ground literal L is *entailed* by an extended logic program, if it belongs to all its answer sets (or, equivalently, to all its extensions in the sense of default logic). Using this notion of entailment and the entailment relation for the language $\mathcal{A}$ introduced in Section 2, we can state a result expressing the soundness of the translation π.

Soundness Theorem. *Let D be a domain description without similar e-propositions. For any v-proposition P, if πD entails πP, then D entails P.*

For an inconsistent D, the statement of the soundness theorem is trivial, because such D entails every v-proposition. For consistent domain descriptions, the statement of the theorem is an immediate consequence of the following lemma which will be proved elsewhere:

Soundness Lemma. *Let D be a consistent domain description without similar e-propositions. There exists an answer set Z of πD such that, for any v-proposition P, if $\pi P \in Z$ then D entails P.*

Note that the lemma asserts the possibility of selecting Z uniformly for all P; this is more than is required for the soundness theorem.

The set Z from the statement of the lemma is obviously consistent, because a consistent domain description cannot entail two complementary v-propositions. Consequently, if D is consistent and does not include similar v-propositions, then πD has a consistent answer set.

The converse of the soundness theorem does not hold, so that the translation π is incomplete. This following simple counterexample belongs to Thomas Woo (personal communication). Let D be the domain with one fluent name F and one action name A, characterized by two propositions:

$$A \textbf{ after } F,$$
$$A \textbf{ causes } F \textbf{ if } F.$$

It is clear that D entails **initially** F. But the translation of this proposition, $Holds(F, S0)$, is not entailed by πD. Indeed, it is easy to verify that the set of all positive ground literals other than $Holds(F, S0)$ is an answer set of πD.

6 Answer Sets and Signings

To prove the soundness lemma, we need the following definition. Let Π be a general logic program (that is, an extended program that does not contain classical negation). A *signing* for Π is any set S of ground atoms such that, for any ground instance

$$B_0 \leftarrow B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$$

of any rule from Π, either

$$B_0, B_1, \ldots, B_m \in S,\ B_{m+1}, \ldots, B_n \notin S$$

or

$$B_0, B_1, \ldots, B_m \notin S,\ B_{m+1}, \ldots, B_n \in S.^5$$

For example, $\{p\}$ is a signing for the program

$$p \leftarrow not\ q, \quad q \leftarrow not\ p, \quad r \leftarrow q.$$

In this section we show that the answer sets of a general program Π which has a signing S can be characterized in terms of the fixpoints of a monotone operator. Specifically, for any set X of ground atoms, let θX be the symmetric difference of X and S:

$$\theta X = (X \setminus S) \cup (S \setminus X).$$

Obviously, θ is one to one. Moreover, it is clear that θ is an involution:

$$\begin{aligned} \theta^2 X &= \{[(X \setminus S) \cup (S \setminus X)] \setminus S\} \cup \{S \setminus [(X \setminus S) \cup (S \setminus X)]\} \\ &= (X \setminus S) \cup (S \cap X) \\ &= X. \end{aligned}$$

We will define a monotone operator ϕ such that any X is an answer set of Π if and only if θX is a fixpoint of ϕ.

Recall that, for general logic programs, the notion of an answer set (or "stable model") can be defined by means of the following construction [7]. Let Π be a general logic program, with every rule replaced by all its ground instances. The *reduct* Π^X of Π relative to a set X of ground atoms is obtained from Π by deleting

(i) each rule that has an expression of the form *not* B in its body with $B \in X$, and

(ii) all expressions of the form *not* B in the bodies of the remaining rules.

Clearly, Π^X is a positive program, and we can consider its "minimal model"—the smallest set of ground atoms closed under its rules. If this set coincides with X, then X is an *answer set* of Π.

This condition can be expressed by the equation $X = \alpha\Pi^X$, where α is the operator that maps any positive program to its minimal model.

Let S be a signing for Π. The operator ϕ is defined by the equation

$$\phi X = \theta\alpha\Pi^{\theta X}.$$

Lemma 1. *A set X of ground atoms is an answer set of Π iff θX is a fixpoint of ϕ.*

Proof. By the definition of ϕ, θX is a fixpoint of ϕ iff

$$\theta\alpha\Pi^{\theta^2 X} = \theta X.$$

Since θ is one-to-one and an involution, this is equivalent to

$$\alpha\Pi^X = X.$$

Note that, since θ is an involution, Lemma 1 can be also stated as follows: X is an answer set of Π iff $X = \theta Y$ for some fixpoint Y of ϕ.

Lemma 2. *The operator ϕ is monotone.*

Proof. Let Π_1 be the set of all rules from Π whose heads belong to S, and let Π_2 be the set of all remaining rules. Clearly, for any X,

$$\Pi^X = \Pi_1^X \cup \Pi_2^X.$$

Since S is a signing for Π, all atoms occurring in Π_1^X belong to S, and all atoms occurring in Π_2^X belong to the complement of S. Consequently, Π_1^X and Π_2^X are disjoint, and

$$\alpha\Pi^X = \alpha\Pi_1^X \cup \alpha\Pi_2^X.$$

Furthermore, for any expression of the form *not* B occurring in Π_1, B does not belong to S; consequently,

$$\Pi_1^X = \Pi_1^{X\setminus S}.$$

Similarly, for any expression of the form *not* B occurring in Π_2, B belongs to S, so that

$$\Pi_2^X = \Pi_2^{X\cap S}.$$

Consequently, for every X,

$$\alpha\Pi^X = \alpha\Pi_1^{X\setminus S} \cup \alpha\Pi_2^{X\cap S}.$$

In particular,

$$\alpha\Pi^{\theta X} = \alpha\Pi_1^{\theta X\setminus S} \cup \alpha\Pi_2^{\theta X\cap S}.$$

It is clear from the definition of θ that

$$\theta X \setminus S = X \setminus S,$$

$$\theta X \cap S = S \setminus X.$$

We conclude that

$$\alpha\Pi^{\theta X} = \alpha\Pi_1^{X\setminus S} \cup \alpha\Pi_2^{S\setminus X}.$$

By the choice of Π_1 and Π_2, $\alpha\Pi_1^{X\setminus S}$ is contained in S, and $\alpha\Pi_2^{S\setminus X}$ is disjoint with S. Consequently,

$$\alpha\Pi^{\theta X} \setminus S = \alpha\Pi_2^{S\setminus X},$$

$$S \setminus \alpha\Pi^{\theta X} = S \setminus \alpha\Pi_1^{X\setminus S}.$$

Hence

$$\phi X = \theta\alpha\Pi^{\theta X} = (\alpha\Pi^{\theta X} \setminus S) \cup (S \setminus \alpha\Pi^{\theta X}) = \alpha\Pi_2^{S\setminus X} \cup (S \setminus \alpha\Pi_1^{X\setminus S}).$$

Since α is monotone, and the reduct operators $X \mapsto \Pi_i^X$ are antimonotone, it follows that ϕ is monotone.

Having proved Lemmas 1 and 2, we can use properties of the fixpoints of monotone operators given by the Knaster-Tarski theorem [20] to study the answer sets of a program with a signing. The Knaster-Tarski theorem asserts, for instance, that every monotone operator has a fixpoint; this gives a new, and more direct, proof of the fact that every general program with a signing has at least one answer set.[6] Moreover, it asserts that a monotone operator has a least fixpoint, which is also its least pre-fixpoint. (A *pre-fixpoint* of ϕ is any set X such that $\phi X \subset X$.) This characterization of the least fixpoint of ϕ is used in the proof of the soundness lemma.

Footnotes

1. A *fluent* is something that may depend on the situation, as, for instance, the location of a moveable object [14]. In particular, *propositional* fluents are assertions that can be true or false depending on the situation.

2. One possible way to represent reasoning about the past in the framework of logic programming is to interpret it as "explanation" and "abduction" [19]. Our approach is more symmetric; we treat both reasoning about the future and reasoning about the past as "deductive." The precise relationship between the two approaches is a subject of further investigation

3. Our preferred approach to causal anomalies is to view them as evidence of unknown events that occur concurrently with the given actions and contribute to the properties of the new situation.

4. Using a sorted language implies, first of all, that all atoms in the rules of the program are formed in accordance with the syntax of sorted predicate

logic. Moreover, when we speak of an *instance* of a rule, it will be always assumed that the terms substituted for variables are of appropriate sorts.

5. This is slightly different from the original definition [11].

6. The existence of answer sets for such programs, and for programs of some more general types, was established by Phan Minh Dung [3] and François Fages [6].

Acknowledgements

We would like to thank G. N. Kartha and Norman McCain for comments on a draft of this paper, Kenneth Kunen for directing us to his paper on signings, and Thomas Woo for the counterexample reproduced in Section 5. This research was supported in part by NSF grants IRI-9101078 and IRI-9103112.

References

[1] Krzysztof Apt and Marc Bezem. Acyclic programs. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 617–633, 1990.

[2] Andrew Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49:5–23, 1991.

[3] Phan Minh Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, 1992. To appear.

[4] Kave Eshghi and Robert Kowalski. Abduction compared with negation as failure. In Giorgio Levi and Maurizio Martelli, editors, *Logic Programming: Proc. of the Sixth Int'l Conf.*, pages 234–255, 1989.

[5] Chris Evans. Negation-as-failure as an approach to the Hanks and McDermott problem. In *Proc. of the Second Int'l Symp. on Artificial Intelligence*, 1989.

[6] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of logic in computer science*, 1992. To appear.

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

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

[9] Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33(3):379–412, 1987.

[10] Henry Kautz. The logic of persistence. In *Proc. of AAAI-86*, pages 401–405, 1986.

[11] Kenneth Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[12] Vladimir Lifschitz and Arkady Rabinov. Miracles in formal theories of actions. *Artificial Intelligence*, 38(2):225–237, 1989.

[13] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986.

[14] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[15] Paul Morris. The anomalous extension problem in default reasoning. *Artificial Intelligence*, 35(3):383–399, 1988.

[16] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[17] Erik Sandewall. Features and fluents. Technical Report LiTH-IDA-R-91-29, Linköping University, 1992.

[18] Lenhart Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

[19] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proc. of IJCAI-89*, pages 1055–1060, 1989.

[20] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.