This paper was selected by a process of
anonymous peer reviewing for presentation at

COMMONSENSE 2007

8th International Symposium on Logical Formalizations of Commonsense Reasoning

Part of the AAAI Spring Symposium Series, March 26-28 2007,
Stanford University, California

Further information, including follow-up notes for some of the
selected papers, can be found at:

www.ucl.ac.uk/commonsense07

Learning Action Descriptions with A-Prolog: Action Language $\mathcal{C}$

Marcello Balduccini
Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
marcello.balduccini@ttu.edu

Abstract

This paper demonstrates how A-Prolog can be used to solve the problem of non-monotonic inductive learning in the context of the learning of the behavior of dynamic domains. Non-monotonic inductive learning is an extension of traditional inductive learning, characterized by the use of default negation in the background knowledge and/or in the clauses being learned. The importance of non-monotonic inductive learning lies in the fact that it allows to learn theories containing defaults and ultimately to help automate the complex task of compiling commonsense knowledge bases.

Introduction

To formalize commonsense knowledge, one needs suitable languages, whose definition has proven to be extremely difficult, and is still to a large extent an open problem. Research on the existing formalisms has also shown that, even when a suitable language is available, the task of compiling the commonsense knowledge about a particular domain is far from trivial. A possible way to simplify this task consists in the adoption of learning techniques, and in particular in the use of inductive learning.[1]

However, as argued in (Sakama 2005), there is a contrast between the nature of inductive learning problems, which assume incompleteness of information, and the languages used in inductive logic programming (ILP), which are not sufficiently expressive to deal with various forms of incomplete and commonsensical knowledge.

From a knowledge representation standpoint, various types of incomplete and commonsense knowledge can be represented by means of *defaults* (statements describing what is *typically* true, as opposed to *always* true). Default negation, when combined with a suitable semantics for logic programs, has been successfully used to encode sophisticated forms of defaults, in particular together with classical negation (see e.g. (Gelfond 2002)). Defaults also make it possible to write *elaboration tolerant* programs (a program is elaboration tolerant when small modifications in the specification yield small modifications in the program). Unfortunately, traditional ILP methods cannot be applied directly to logic programs with default negation, which poses substantial limits on the use of ILP to learn commonsense knowledge.

Some authors have attempted to overcome the problem by defining reductions of normal programs to negation-free programs, allowing to apply ILP methods (e.g. (Otero 2003; 2005)). Other authors have instead developed techniques that do not rely on the traditional methods (Otero 2001; Sakama 2005). The latter techniques are often referred to as *non-monotonic ILP* (NMILP) (Sakama 2001; 2005).

The aim of this paper is to show that A-Prolog (Gelfond & Lifschitz 1988; 1991), a powerful formalism for knowledge representation and reasoning, can be used (besides planning and diagnosis) for non-monotonic inductive learning tasks, and, ultimately, to learn commonsense knowledge. Differently from others, *our approach allows not only the addition of new laws, but also the modification of existing ones*. To demonstrate A-Prolog's ability to deal with normal and extended logic programs, we focus on the task of learning action descriptions in action language $\mathcal{C}$ (Giunchiglia & Lifschitz 1998). In fact, the translation of $\mathcal{C}$ to logic programming makes a heavily use of default and classical negation. For this reason, it is unlikely that a reduction to monotonic methods can be used. Moreover, the fact that learning an action description typically requires generating laws that match *several* sample transitions, does not allow a direct application of NMILP approaches such as (Sakama 2005). Finally, our approach to building learning modules has the added value of being *entirely declarative*.

The paper is organized as follows. We begin with an introduction on A-Prolog. Next, we describe the two main parts of our approach: the encoding of the action description and the A-Prolog learning module. Finally, we compare with other NMILP approaches and draw conclusions. Examples that were not included in this paper because of space restrictions are available at www.ucl.ac.uk/commonsense07/papers/notes/.

[1] Although our work is also related to update/revision of knowledge bases, we view it to be closer to inductive learning because of the stress on the generality of the learned laws. We plan to discuss the link with update/revision techniques in an extended version of this paper.

A-Prolog

A-Prolog is a knowledge representation language that allows the formalization of various forms of commonsense knowledge and reasoning. The language is one of the products of the research aimed at defining a formal semantics for logic programs containing default negation (Gelfond & Lifschitz 1988), and was later extended to allow also classical negation (Gelfond & Lifschitz 1991).

A *(regular) rule* r of A-Prolog is a statement of the form:

$$h \leftarrow l_1, l_2, \ldots l_m, \text{not } l_{m+1}, \text{not } l_{m+2}, \ldots, \text{not } l_n. \quad (1)$$

where h and l_i's are literals and "not" is the default negation symbol. The informal meaning of the statement, in terms of the set of beliefs of an agent complying with it, is "if you believe $l_1, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then you must believe h." We call h the *head* of the rule ($head(r)$) and $l_1, \ldots, \text{not } l_n$ the *body* ($body(r)$).

The semantics of A-Prolog programs is defined first for programs not containing default negation (*definite* programs). Let Π be a definite program, and S a consistent set of literals. We say that S is closed under a rule w if $head(w) \in S$ whenever $body(w) \subseteq S$. If S is the set-theoretically minimal set of literals closed under the rules of Π, then S is the *answer set of* Π.

If Π is an arbitrary program, we first reduce it to a definite program. The *reduct of an A-Prolog program* Π *with respect to a set of literals* S is denoted by Π^S and obtained from Π by deleting each rule, r, such that $neg(r) \setminus S \neq \emptyset$, and by removing all expressions of the form not l from the bodies of the remaining rules. A consistent set S of literals is an answer set of a program Π if it is an answer set of Π^S.

To allow a more compact representation of some types of knowledge, we introduce the following abbreviation. A *choice macro* is a statement of the form:

$$\{p_1(\vec{X}), \ldots, p_n(\vec{X})\} \leftarrow \Gamma. \quad (2)$$

where $\vec{X}$ is a (possibly empty) list of terms, and $p_1, \ldots, p_n$ do not occur in Γ. The statement informally says that any $\vec{X}$ can have any property p_i, and stands for the set of rules $\{p_i(\vec{X}) \leftarrow \text{not } \neg p_i(\vec{X}), \Gamma. \ , \ \neg p_i(\vec{X}) \leftarrow \text{not } p_i(\vec{X}), \Gamma.\}$, where i ranges over $1 \ldots n$.

The choice macro is inspired to (Niemela & Simons 2000). Its use allows to keep a compact representation of knowledge, later in the paper, without committing to a particular extension of A-Prolog and to its corresponding inference engine.

Step 1: Encoding the Action Description

Our approach is based on two components: the definition of an encoding of the action description suitable for A-Prolog based learning, and the specification of the A-Prolog learning module. In this section we define the encoding.

In this paper, the signature of an action language consists of sets of constant, variable, fluent, and action symbols. *Fluents* (used to represent relevant properties of the domain whose truth value changes over time) are expressions of the form $f(t_1, \ldots, t_n)$ where f is a fluent symbol and t_i's are constants or variables. Similarly, *elementary actions* are expressions of the form $e(t_1, \ldots, t_n)$, where e is an action symbol. A fluent or action is *ground* if all of its arguments are constants, and is *non-ground* otherwise. Fluents and their negations (i.e. $\neg f$, where f is a fluent) are called *fluent literals* (or simply *literals*, whenever no confusion is possible). Sets of elementary actions (intended for concurrent execution) are called *compound actions*. The term actions denotes both elementary and compound actions.

Recall that a *definite action description* of language $\mathcal{C}$ consists of *static laws* of the form

$$\text{caused } r \text{ if } l_1, \ldots, l_n \quad (3)$$

and *dynamic laws* of the form

$$\text{caused } r \text{ if } l_1, \ldots, l_n \text{ after } p_1, \ldots, p_m \quad (4)$$

where r (the *head*) is a literal or $\bot$, l_i's (the *if-preconditions*) are literals, and p_i's (the *after-preconditions*) are literals or elementary actions (the definition, with a somewhat different terminology, can be found in (Lifschitz & Turner 1999)).

A translation from $\mathcal{C}$ to logic programming can be found in (Lifschitz & Turner 1999). That is a case of "direct" translation, where each law is mapped into one rule. For example, a static law of the form (3) is encoded by:

$$h(r, T) \leftarrow \text{not } h(\overline{l_1}, T), \ldots, \text{not } h(\overline{l_n}, T). \quad (5)$$

(By $\bar{l}$ we denote the literal complementary to l.) In some cases, it is more convenient to use an "indirect" encoding, where laws are encoded by collections of facts, together with a *general schema*, describing the semantics of the law. A possible fact-based encoding of the law above is:

$$\begin{array}{l} s\_law(s_1). \quad head(s_1, r). \\ if(s_1, \langle l_1, \ldots, l_n \rangle). \end{array} \quad (6)$$

and the semantics is encoded by a rule:

$$\begin{array}{l} h(H, T) \leftarrow s\_law(W), head(W, H), \\ \qquad all\_if\_h(W, T). \end{array} \quad (7)$$

where $all\_if\_h(W, T)$ intuitively means that all the preconditions following the "if" keyword hold.[2] Notice that we view the above facts containing tuples as macros. For instance, fact $if(s_1, \langle l_1, \ldots, l_n \rangle)$ above stands for the collection of facts $if(s_1, 1, l_1)$, ..., $if(s_1, n, l_n)$. We also assume the existence of suitable relations len and nth that allow to retrieve respectively the length and each element of the tuple. In the example above, $len(if(s_1), n)$ holds, as well as $nth(if(s_1), 1, l_1)$, $nth(if(s_1), 2, l_2)$, etc.

It is worth stressing that *the explicit use of default negation in the specification of the semantics of the laws of* $\mathcal{C}$ *makes it difficult to adapt the traditional ILP techniques to the language.* We will come back to this issue later.

Notice the importance of the role of the law's name (specified by $s\_law(s_1)$ in (6)) when encoding laws that contain variables. For example, a law stating that object O is wet when it is in water, can be encoded by facts:

$$\begin{array}{l} s\_law(s_2(O)). \quad head(s_2(O), wet(O)). \\ if(s_2(O), \langle inWater(O) \rangle). \end{array} \quad (8)$$

[2] The formal definition of these and other auxiliary relations is omitted to save space.

Fact-based encodings are particularly convenient for A-Prolog based learning. Intuitively, once the semantics of an action language has been described with general schemas such as (7), it is possible to reduce the task of learning to that of finding collections of facts such as (8). If the laws are not allowed to contain variables, then the goal can be simply accomplished in A-Prolog by using choice macros. For example, the generation of the precondition list of a law can be roughly[3] obtained by means of a rule:

$$\{if(W, N, L)\} \leftarrow s\_law(W). \tag{9}$$

where N and L range respectively over the positions in the precondition list and the (ground) fluent literals. From a knowledge representation standpoint, the intuitive meaning of the rule is that any fluent literal L can occur in any position of the precondition list of law W.

When variables are allowed, unfortunately, (9) may yield unintended results. Consider static law $s_2(O)$ above, and let O takes on values o_1 and o_2. Intuitively, we consider the law as a single statement. However, recall that in A-Prolog non-ground rules are semantically equivalent to the set of their ground instances. Hence, rule (9) acts *separately* on each ground instantiation of law $s_2(O)$ (such as instance $s_2(o_1)$, whose effect is $wet(o_1)$). It is not difficult to see that this may cause the addition of some precondition p to one ground instance of the law, but not to another. To deal with laws containing variables, the encoding must be extended. The key step consists in defining ground names for non-ground literals and actions, as follows.

Let $\iota = l(t_1, \ldots, t_k)$ be a fluent literal, and $V = \langle X_1, \ldots, X_j \rangle$ be a tuple of variables such that all the variables from ι are in V. For every variable t_i, the expression $V \downarrow t_i$ denotes the index p in V such that $X_p = t_i$. For example, $\langle X, Y, Z \rangle \downarrow Y = 2$. The *groundification of* $l(t_1, \ldots, t_k)$ *w.r.t.* V is the expression $l(g_1, \ldots, g_k)$, where g_i is $\nu(V \downarrow t_i)$ (ν does not belong to the signature of AD) if t_i is a variable, and $g_i = t_i$ otherwise. We denote the groundification of a literal l w.r.t. V by l^V. For example, given $V = \langle X, Y, Z \rangle$ and $l = p(a, Z, b, c, Y)$, l^V is $p(a, \nu(3), b, c, \nu(2))$. In a similar way we define the groundification of an elementary action. When we need to distinguish between literals (or actions), and their groundifications, we will call the former *regular* literals (resp., actions) and the latter *groundified* literals (resp., actions). Given a law $w(X_1, \ldots, X_j)$, we call the tuple $\langle X_1, \ldots, X_j \rangle$ the *variable-list of the law*, and w the *prefix of the law*. We denote the variable-list of w by $w^{\mathcal{V}}$ and its prefix by $w^{\mathcal{P}}$. Clearly, for each l and a occurring in some law w, their groundifications w.r.t. $w^{\mathcal{V}}$ are defined. We denote them respectively by l^w and a^w. Given a law w, the association between a literal l and its groundification w.r.t. $w^{\mathcal{V}}$ is encoded by fact[4] $gr(l, \lambda(w^{\mathcal{V}}), l^w)$ (similarly for actions). For example, the association between $wet(O)$ and $wet(\nu(1))$ w.r.t. $s_2(O)$ is represented as $gr(wet(O), \lambda(O), wet(\nu(1)))$.

[3] Some constraints are also needed to suitably restrict the generation.

[4] Relation gr can also be defined more compactly.

The encoding of a static law w is then:

$$s\_law(w^{\mathcal{P}}). \quad head(w^{\mathcal{P}}, r^w).$$
$$vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}})). \quad if(w^{\mathcal{P}}, \langle l_1^w, l_2^w, \ldots, l_n^w \rangle).$$

The semantics of static laws becomes:

$$\begin{aligned} h(H, T) \leftarrow\ & s\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T). \end{aligned} \tag{10}$$

Notice that the conclusion of the rule is still a regular literal. This makes it possible to use the new encoding to replace directly the original one. The encoding of the law from (8) is:

$$s\_law(s_2). \quad head(s_2, wet(\nu(1))).$$
$$vlist(s_2, \lambda(O)). \quad if(s_2, \langle inWater(\nu(1)) \rangle). \tag{11}$$

Groundified literals and actions are mapped to their regular counterparts, when needed, by means of relation gr. For example, the definition of $all\_if\_h(W, T)$ is:

$$all\_if\_h(W, VL, T) \leftarrow all\_if\_from(W, VL, T, 1).$$

$$all\_if\_from(W, VL, T, N+1) \leftarrow len(if(W), N).$$

$$\begin{aligned} all\_if\_from(W, VL, T, N) \leftarrow\ & nth(if(W), N, L_g), \\ & gr(L, VL, L_g), \text{not } h(\overline{L}, T), \\ & all\_if\_from(W, VL, T, N+1). \end{aligned}$$

Intuitively, $all\_if\_from(W, VL, T, N)$ means that all the if-preconditions of W (w.r.t. variable list VL) with index N or greater hold at step T.

A dynamic law w of the form (4) is encoded by:

$$d\_law(w^{\mathcal{P}}). \quad head(w^{\mathcal{P}}, r^w). \quad vlist(w^{\mathcal{P}}, \lambda(w^{\mathcal{V}})).$$
$$if(w^{\mathcal{P}}, \langle l_1^w, \ldots, l_n^w \rangle). \quad after(w^{\mathcal{P}}, \langle p_1^w, \ldots, p_m^w \rangle).$$

and the semantics is defined by:

$$\begin{aligned} h(H, T+1) \leftarrow\ & d\_law(W), vlist(W, VL), \\ & head(W, H_g), gr(H, VL, H_g), \\ & all\_if\_h(W, VL, T), all\_after\_h(W, VL, T). \end{aligned}$$

The set of rules[5] defining the semantics of $\mathcal{C}$ is denoted by $Sem(\mathcal{C})$. An *action description* AD consists of the union of $Sem(\mathcal{C})$ and the sets of atoms encoding the laws.[6]

In reasoning about dynamic domains, an important role is played by the observation of the domain's behavior. Here, we use observations to encode the examples for the learning task. Observations are encoded by statements of the form $obs(l, t)$, meaning that literal l was observed to hold at step t, and $hpd(a, t)$, meaning that action a happened at t. The *history of the domain up to step* cT is denoted by H^{cT} and consists of a collection of statements $obs(l, t)$ and $hpd(a, t)$, where $0 \leq t \leq cT$ for the former and $0 \leq t < cT$ for the latter. A domain description is a pair $DD = \langle AD, H^{cT} \rangle$. Given such a domain description DD, by *program* DD we mean the program $AD \cup H^{cT} \cup \Pi_r$, where Π_r is:

$$\Pi_r \left\{ \begin{array}{l} h(L, 0) \leftarrow obs(L, 0). \\ o(A, T) \leftarrow hpd(A, T). \\ \leftarrow h(L, T), obs(\overline{L}, T). \end{array} \right.$$

[5] A few rules were omitted to save space.

[6] Although not required by our approach, to simplify the presentation we assume completeness of knowledge about the initial situation and the actions performed.

Intuitively, the use of Π_r ensures that the possible evolutions of the domain identified by the answer sets of DD match the the history H^{cT} (see (Balduccini & Gelfond 2003a) for more details). In the next section we discuss how we use the history to detect the need for learning, and how the (possibly empty) action description is modified to match the examples provided.

Step 2: Modifying the Action Description

When given a history, we expect a rational agent capable of learning to perform two steps: (1) check if the history can be explained by the action description, and (2) modify the action description accordingly if the history cannot be explained. Notice that we talk about *modifying* the action description, rather than learning an action description. That is because our approach is also capable of *incremental* learning of action descriptions: by default, the existing laws can be modified, as well as new laws created.[7]

Central to the reasoning required to check if the history can be explained by the action description is the notion of *symptom*. Given a domain description $DD = \langle AD, H^{cT} \rangle$, H^{cT} is said to be a *symptom* if (program) DD is inconsistent (that it, it has no answer sets). It can be shown that the history is explained by the action description iff H^{cT} is a symptom.

Next, we define what it means to modify an action description. We provide an implementation-independent definition that can be used to verify properties, such as soundness and completeness, of the A-Prolog learning module described later.

A *modification* of an action description AD is a collection of *modification statements* of the form: $d\_law(\omega)$, $s\_law(\omega)$, $vlist(\omega, \lambda(X_1, \ldots, X_k))$, $head(\omega, hg)$, $if(\omega, \eta, lg)$, $after(\omega, \eta, ag)$, where ω is a constant, X_i's are variables, η is a positive integer, lg is a groundified literal, and ag is a groundified action. Intuitively, a modification $\mathcal{M}$ is *valid w.r.t.* AD if AD together with the $\mathcal{M}$ describe valid dynamic laws and state constraints. More precisely, $\mathcal{M}$ is valid if:

- For every $d\_law(\omega) \in \mathcal{M}$, we have $s\_law(\omega) \notin AD \cup \mathcal{M}$ and $vlist(\omega, \Lambda) \in AD \cup \mathcal{M}$ for some Λ. Similarly for $s\_law(\omega)$.
- $head(\omega, hg) \in \mathcal{M}$ iff either $d\_law(\omega)$ or $s\_law(\omega)$ is in $\mathcal{M}$.
- For every $if(\omega, \eta, lg) \in \mathcal{M}$, either $d\_law(\omega)$ or $s\_law(\omega)$ is in $AD \cup \mathcal{M}$.
- $after(\omega, \eta, ag) \in \mathcal{M}$ implies $d\_law(\omega) \in AD \cup \mathcal{M}$.
- For every $if(\omega, \eta, lg) \in \mathcal{M}$ and $vlist(\omega, \Lambda) \in AD \cup \mathcal{M}$, lg is a valid groundification w.r.t. Λ.[8] Similarly for ag from $after(\omega, \eta, ag)$.
- For every ω, the indexes η_i from all the statements of the form $if(\omega, \eta_i, lg)$ from $AD \cup \mathcal{M}$ must form a complete sequence of integers starting from 1.[9] Similarly for $after(\omega, \eta_i, ag)$.
- For every $if(\omega, \eta, lg) \in \mathcal{M}$ and $lg' \neq lg$, it must hold that $if(\omega, \eta, lg') \notin AD \cup \mathcal{M}$. Similarly for $after(\omega, \eta, ag)$.

According to the definition, given an empty action description, $\{d\_law(\omega), vlist(\omega, \lambda(X_1, \ldots, X_k))\}$ is not valid modifications, but $\{d\_law(\omega), vlist(\omega, \lambda(X_1, \ldots, X_k)), head(\omega, hg)\}$ is.

The learning task is reduced to finding a valid modification that explains the symptom. More precisely:

Definition 1 *For every $DD = \langle AD, H^{cT} \rangle$, an* inductive correction of AD for symptom H^{cT} *is a valid modification $\mathcal{M}$ such that $DD \cup \mathcal{M}$ is consistent.*

Recall that, if H^{cT} is a symptom, then DD itself is inconsistent. To better understand the definition, consider an action description containing (11) and laws stating that the literals formed by $inWater$ and wet are inertial (see (Lifschitz & Turner 1999)). Let $H^{cT} = \{obs(\neg wet(o_1), 0), obs(\neg inWater(o_1), 0), hpd(putInWater(o_1), 0), obs(wet(o_1), 1)\}$. It is easy to check that H^{cT} is a symptom. In fact, $obs(\neg wet(o_1), 0)$ and the first rule of Π_r imply $h(\neg wet(o_1), 0)$. Because $wet(O)$ is an inertial fluent, $h(\neg wet(o_1), 1)$ also holds. This conclusion and $obs(wet(o_1), 1)$ satisfy the body of the constraint from Π_r, making DD inconsistent. Now consider $\mathcal{M}_1 = \{d\_law(d_1), vlist(d_1, \lambda(O)), head(d_1, inWater(\nu(1))), after(d_1, \langle putInWater(\nu(1)) \rangle)\}$. It is not difficult to see that $\mathcal{M}_1$ is an inductive correction. In fact, from H^{cT} and Π_r we obtain $o(putInWater(o_1), 0)$. This allows to apply d_1 and conclude $h(wet(o_1), 1)$. Hence, the body of the constraint in Π_r is false, and $DD \cup \mathcal{M}_1$ is consistent.

Next, we show how inductive corrections can be computed using A-Prolog. Let Π_{ps} be a set of rules of the form $available(w)$ and $avail\_vlist(w, \lambda(X_1, \ldots, X_{w_j}))$. We call Π_{ps} the *prefix store*. Intuitively, the purpose of Π_{ps} is to provide fresh symbols and suitable variable lists for the definition of new laws. The *learning module* $\mathcal{L}$ consists of the following rules:

1. $\{if(W, N, Lg)\}$.
2. $\leftarrow if(W, N, Lg_1), if(W, N, Lg_2), Lg_1 \neq Lg_2$.
3. $\leftarrow has\_if(W, N), N > 1,$ not $has\_if(W, N-1)$.
4. $\leftarrow if(W, N, Lg),$ not $valid\_gr(W, N, Lg)$.

5. $\{d\_law(W), s\_law(W)\} \leftarrow available(W)$.
6. $\leftarrow d\_law(W), s\_law(W)$.

7. $\{head(W, Hg)\} \leftarrow newly\_defined(W)$.
8. $\leftarrow newly\_defined(W),$ not $has\_head(W)$.
9. $\leftarrow head(W, Hg_1), head(W, Hg_2)$.
10. $\leftarrow head(W, Hg),$ not $valid\_gr(W, N, Hg)$.

[7] Although it is not difficult to force our learning module to act in a non-incremental fashion, or to only modify certain laws, we will not go into details in this paper.

[8] That is, the arguments of the $\nu(N)$ terms must be valid indexes for the tuple defined by Λ.

[9] For example, if η_i is 2 and η_j from some $prec(\omega, \eta_j, lg')$ is 4, there must be some $prec(\omega, \eta_k, lg'')$ such that $\eta_k = 3$.

11. $\{after(W, N, Ag)\} \leftarrow d\_law(W)$.
12. $\leftarrow after(W, N, Ag_1), after(W, N, Ag_2), Ag_1 \neq Ag_2$.
13. $\leftarrow has\_after(W, N), N > 1, \text{not } has\_after(W, N-1)$.
14. $\leftarrow after(W, N, Ag), \text{not } valid\_gr(W, N, Ag)$.

15. $vlist(W, VL) \leftarrow newly\_defined(W), avail\_vlist(W, VL)$.

where W ranges over law prefixes, N ranges over positive integers, Ag (possibly indexed) denotes a groundified action or literal, and L_g (possibly indexed) and H_g denote groundified literals. $\mathcal{L}$ can be viewed as composed of two modules: rules (1), (5), (7), (11), and (15) roughly generate modifications, while the rest of $\mathcal{L}$ guarantees that the modification encoded by each answer set is valid. The process of finding inductive corrections is completed by Π_r, which ensures that every answer set explains the observations.

Let us now look at $\mathcal{L}$ is more detail. Rule (1) intuitively says that any Lg may be specified as N^{th} if-precondition of W. Rule (2) guarantees that only one groundified literal is selected for each position in the if-precondition list. Rule (3) guarantees that there are no "holes" in the assignment of the indexes: relation $has\_if(W, N)$ (definition omitted) holds if W has an if-precondition with index N, and can be trivially defined from $if(W, N, Lg)$. Rule (4) states that Lg must be a valid groundified literal for W. For example, $lit(\nu(1))$ is a valid groundified literal for $d_2(N)$, but $lit(\nu(3))$ is not. Relation $valid\_gr(W, N, X)$ (definition omitted) is defined to hold if there exists a literal or action of which X is the groundification w.r.t. W, and can be easily defined from relation gr. Rule (5) intuitively says that any available constant may be used as prefix of a new dynamic law or state constraint. Rule (6) ensures that the same constant is not used as prefix of both a dynamic law and a state constraint. Rule (7) says that any Hg may be head (i.e. the effect) of a newly defined law W: relation $newly\_defined(W)$ (definition omitted) is true if both $available(W)$ (defined in Π_{ps}) and one of $d\_law(W)$, $s\_law(W)$ hold. Rules (8)-(10) ensure that every newly defined law has exactly one head Hg, and that Hg is a valid groundified literal for W. Relation $has\_head(W)$ (definition omitted) is defined similarly to $has\_if(W, N)$ above. Rule (11) intuitively says that any Ag may be specified as N^{th} after-precondition of a dynamic law W. Rules (12)-(14) state that only one after-precondition is associated with each index, that there are no "holes" in the assignments of indexes ($has\_after(W, N)$ is defined similarly to $has\_if(W, N)$ above), and that every Ag is a valid groundified action or literal for W. Finally, rule (15) says the variable-list of each newly defined law is taken from the prefix store, Π_{ps}. Notice that *the learning module is substantially independent of the semantics of the language.* $\mathcal{L}$ only depends on the predicates used for the fact-based encoding, and it is not difficult to see that the changes required to support languages other than $\mathcal{C}$ are conceptually simple.

Intuitively, the computation of the inductive corrections of a domain description DD is reduced to finding the answer sets of the program $DD \cup \mathcal{L}$:

Theorem 1 (Soundness and Completeness) *For every $DD = \langle AD, H^{cT} \rangle$, there exists a prefix store Π_{ps} such that the inductive corrections of AD for H^{cT} and the answer sets of $DD \cup \Pi_{ps} \cup \mathcal{L}$ are in one-to-one correspondence.*

Proof. *(Sketch)* Left-to-right. *Let $\mathcal{M}$ be an inductive correction of AD. Using the Splitting Set Lemma, split $\Pi = DD \cup \Pi_{ps} \cup \mathcal{L}$ in Π_{ps}, Π_d, consisting of all the rules and facts used to encode laws of $\mathcal{AL}^+$ (including e.g. rule (1) of $\mathcal{L}$), Π_s, consisting of all the other rules with a non-empty head from Π, and Π_c, consisting of the constraints of Π. It is not difficult to show from Definition 1 that $\mathcal{M}$ is contained in some answer set A of $\Pi_{ps} \cup \Pi_d \cup \Pi_s$. The reasoning can also be extended to show that A satisfies the constraints of $\Pi_c \cap DD$. Finally, from the definition of valid modification, we conclude that A also satisfies $\Pi_c \setminus DD$.*
Right-to-left. *Let A be an answer set of Π and $\mathcal{M}$ be the set of modification statements from A. By Definition 1 we need to show that $\Pi' = DD \cup \mathcal{M}$ is consistent. Let us split Π as above and Π' into Π'_d, Π'_s, and Π'_c, following the same technique used for Π. Let $A' \subseteq A$ be an answer set of $\Pi_{ps} \cup \Pi_d$ (existence follows from the Splitting Set Lemma). It can be shown that $A' \setminus \Pi_{ps}$ is an answer set of Π'_d. Notice now that $\Pi'_s \cup \Pi'_c = \Pi_s \cup \Pi_c$: from the Splitting Set Lemma (and the fact that Π is consistent) it follows that Π' is consistent.* □

An inductive correction can be obtained from the corresponding answer set A of $DD \cup \Pi_{ps} \cup \mathcal{L}$ by extracting the modification statements from A.

It is not difficult to convince oneself that $\mathcal{M}_1$ from the previous example can be generated by $DD \cup \Pi_{ps} \cup \mathcal{L}$. In fact, given $\Pi_{ps} = \{available(d_1), avail\_vlist(d_1, \lambda(O))\}$, the choice macros of $\mathcal{L}$ can obviously generate $\mathcal{M}_1$. By inspection of the constraints of $\mathcal{L}$, it is possible to see that $\mathcal{M}_1$ defeats all of their bodies. Finally, with the same reasoning used in the previous example, we can conclude that the body of constraint in Π_r is also never satisfied.

Related Work

To the best of our knowledge, ours is the first work investigating the use of A-Prolog to implement learning modules. It is also the first attempt at defining a *declarative solution* to the problem of learning normal logic programs.

Various attempts have been made at characterizing learning of normal logic programs, some of them based on the answer set semantics. Because of space restrictions, we will focus on the ones that are most relevant to our research. In (Otero 2003; 2005; Otero & Varela 2006), the authors describe an interesting method to simplify learning problems based on an action language similar to $\mathcal{AL}$ (Balduccini & Gelfond 2003a), so that the traditional ILP approaches are applicable. Differently from our approach, this technique targets a particular action language, as the required manipulations of the examples depend on the semantics of the language. Because of the use of default negation in the translation of $\mathcal{C}$, it is unclear whether a reduction to traditional ILP approaches may exist. Differently from (Otero & Varela

2006), our inductive corrections are by no means limited to planning (for example, they can be applied for diagnostic reasoning as defined in (Balduccini & Gelfond 2003a)). Interestingly, according to preliminary experimental results, the performance of a simple implementation of our approach for language $\mathcal{AL}$ appears to be reasonably close to that of Iaction (Otero & Varela 2006): for the experiment with 5 narratives of 4 blocks and 6 actions described in that paper, the solution is found using our approach in 14 sec on a Pentium 4, 3.2GHz with 1.5 GB RAM running cmodels 3.59 (Lierler & Maratea 2004), which is fairly comparable to the time of 36 sec taken by Iaction on a somewhat slower machine (Pentium 4, 2.4GHz).

In (Sakama 2005), a method for learning normal logic programs is presented, which does not rely on traditional ILP approaches. The main differences with our approach are: (1) The target predicate of positive examples is not allowed to occur in the background knowledge. Hence, the method cannot be applied directly when an observation about a literal f is given, and some law for f already exists; (2) If multiple examples are given, a solution may be returned that correctly covers only the last example, even when a solution covering all of them exists; (3) At most one rule can be learned for each example (it can be seen from $\mathcal{L}$ that there is no limit to the number of laws that can be learned with our approach).

In (Otero 2001), a logical characterization of the general problem of induction of normal logic programs is given, based on the answer set semantics. The work does not seem to be affected by the limitations of (Sakama 2005), but a thorough comparison is difficult because (Otero 2001) does not contain a complete definition of an algorithm.

Differently from all of the approaches above, the declarative nature of our technique makes it relatively simple to introduce various minimization criteria on the solutions. For example, set-theoretically minimal inductive corrections can be found by replacing the choice macros in $\mathcal{L}$ by cr-rules of CR-Prolog (Balduccini & Gelfond 2003b) as follows:

rule 1: $if(W, N, Lg) \stackrel{+}{\leftarrow}$.
rule 5: $d\_law(W)$ OR $s\_law(W) \stackrel{+}{\leftarrow} available(W)$.
rule 7: $head(W, Hg) \stackrel{+}{\leftarrow} newly\_defined(W)$.
rule 11: $after(W, N, Ag) \stackrel{+}{\leftarrow} d\_law(W)$.

Other types of minimization can be similarly obtained using CR-Prolog or other extensions of A-Prolog.

Acknowledgments

This work was supported in part by NASA contract NASA-NNG05GP48G and ATEE/DTO contract ASU-06-C-0143.

References

Balduccini, M., and Gelfond, M. 2003a. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)* 3(4–5):425–461.

Balduccini, M., and Gelfond, M. 2003b. Logic Programs with Consistency-Restoring Rules. In Doherty, P.; McCarthy, J.; and Williams, M.-A., eds., *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, 9–18.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, 1070–1080.

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

Gelfond, M. 2002. Representing Knowledge in A-Prolog. In Kakas, A. C., and Sadri, F., eds., *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, 413–451. Springer Verlag, Berlin.

Giunchiglia, E., and Lifschitz, V. 1998. An Action Language Based on Causal Explanation: Preliminary Report. In *Proceedings of the 15th National Conference of Artificial Intelligence (AAAI'98)*.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: SAT-based Answer Sets Solver Enhanced to Non-tight Programs. In *Proceedings of LPNMR-7*.

Lifschitz, V., and Turner, H. 1999. Representing transition systems by logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99)*, number 1730 in Lecture Notes in Artificial Intelligence (LNCS), 92–106. Springer Verlag, Berlin.

Niemela, I., and Simons, P. 2000. *Extending the Smodels System with Cardinality and Weight Constraints*. Logic-Based Artificial Intelligence. Kluwer Academic Publishers. 491–521.

Otero, R., and Varela, M. 2006. Iaction, a System for Learning Action Descriptions for Planning. In *Proceedings of the 16th International Conference on Inductive Logic Programming, ILP 06*.

Otero, R. 2001. Induction of Stable Models. In *Proceedings of 11th Int. Conference on Inductive Logic Programming, ILP-01*, number 2157 in Lecture Notes in Artificial Intelligence (LNCS), 193–205.

Otero, R. 2003. Induction of the effects of actions by monotonic methods. In *Proceedings of the 13th International Conference on Inductive Logic Programming, ILP 03*, number 2835 in Lecture Notes in Artificial Intelligence (LNCS), 299–310.

Otero, R. 2005. Induction of the indirect effects of actions by monotonic methods. In *Proceedings of the 15th International Conference on Inductive Logic Programming, ILP 05*, number 3625 in Lecture Notes in Artificial Intelligence (LNCS), 279–294.

Sakama, C. 2001. Nonmonotonic inductive logic programming. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, 62–80.

Sakama, C. 2005. Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic* 6(2):203–231.