Representing Knowledge in A-Prolog

Michael Gelfond

Department of Computer Science
Texas Tech University
Lubbock, TX 79409, USA
mgelfond@cs.ttu.edu
http://www.cs.ttu.edu/ ~mgelfond

Abstract. In this paper, we review some recent work on declarative logic programming languages based on stable models/answer sets semantics of logic programs. These languages, gathered together under the name of A-Prolog, can be used to represent various types of knowledge about the world. By way of example we demonstrate how the corresponding representations together with inference mechanisms associated with A-Prolog can be used to solve various programming tasks.

1 Introduction

Understanding the basic principles which can serve as foundation for building programs capable of learning and reasoning about their environment is one of the most interesting and important challenges faced by people working in Artificial Intelligence and Computing Science. Frequently search for these principles is centered on finding efficient means of human-computer communication, i.e. on programming languages[1]. Such languages differ according to the type of information their designers want to communicate to computers. There are two basic types of languages - *algorithmic* and *declarative*. Programs in algorithmic languages describe sequences of actions for a computer to perform while declarative programs can be viewed as collections of statements describing objects of a domain and their properties. A semantic of a declarative program Π is normally given by defining its models, i.e. possible states of the world compatible with Π. Statements which are true in all such models constitute the set of valid consequences of Π. Declarative programming consists in representing knowledge, about the domain relevant to the programmer's goals, by a program Π (often called a *knowledge base*) and in reducing various programming tasks to finding models or computing consequences of Π. Normally, models are found and/or consequences are computed by general purpose reasoning algorithms often called *inference engines.* There are a number of requirements which should be satisfied by a declarative programming language. Some of these requirements are common

[1] In this paper by programming we mean a process of refining specifications. Consequently, the notion of a programming language is understood broadly and includes specification languages which are not necessarily executable.

to all programming languages. For instance, there is always a need for a simple syntax and a clear definition of the meaning of a program. Among other things such a definition should provide the basis for the development of mathematical theory of the language. It is also important to have a programming methodology to guide a programmer in the process of finding a solution to his problem and in the design and implementation of this solution on a computer. These and other general requirements are well understood and frequently discussed in the programming language community. There are, however, some important requirements which seems to be pertinent mainly to declarative languages. We would like to mention those which we believe to be especially important and which will play a role in our discussion.

• A declarative language should allow construction of *elaboration tolerant* knowledge bases, i.e. the bases in which small modifications of the informal body of knowledge correspond to small modifications of the formal base representing this knowledge. It seems that this requirement is easier satisfied if we use languages with *nonmonotonic* consequence relation. (A consequence relation $\models_{\mathcal{L}}$ is called nonmonotonic if there are formulas A, B and C in $\mathcal{L}$ such that $A \models C$ but $A, B \not\models C$, i.e. addition of new information to the knowledge base of a reasoner may invalidate some of his previous conclusions [59]). This property is especially important for representing common-sense knowledge about the world. In common-sense reasoning, additions to the agent's knowledge are frequent and inferences are often based on the absence of knowledge. Modeling such reasoning in languages with a non-monotonic consequence relation seem to lead to simpler and more elaboration tolerant representations.

• Inference engines associated with the language should be sufficiently general and efficient. Notice however that, since some of the relations one needs to teach a computer about are not enumerable, such systems cannot in general be complete. Language designers should therefore look for the 'right' balance between the expressive power of the language and computational efficiency of its inference engine.

We are not sure that it is possible (and even desirable) to design a knowledge representation language suitable for all possible domains and problems. The choice of the language, its semantics and its consequence relation depends significantly on the types of statements of natural language used in the informal descriptions of programming tasks faced by the programmer.

In this paper we discuss A-Prolog – a language of logic programs under answer set (stable model) semantics [30],[31]. A-Prolog can be viewed as a purely declarative language with roots in logic programming [42, 43, 85], syntax and semantics of standard Prolog [18], [22], and in the work on nonmonotonic logic [73], [62]. It differs from many other knowledge representation languages by its ability to represent *defaults*, i.e. statements of the form "*Elements of a class C normally satisfy property P*". One may learn early in life that parents normally love their children. So knowing that Mary is a mother of John he may conclude that Mary loves John and act accordingly. Later one can learn that Mary is an exception to the above default, conclude that Mary does not really like John, and use this new knowledge to change his behavior. One can argue that a substantial part of our education consists in learning various defaults, exceptions to these defaults, and the ways of using this information to draw reasonable conclusions about the world and the consequences of our actions. A-Prolog provides a powerful logical model of this process. Its syntax allows simple representation of defaults and their exceptions, its consequence relation characterizes the corresponding set of valid conclusions, and its inference mechanisms allow a program to find these conclusions in a "reasonable" amount of time.

There are other important types of statements which can be nicely expressed in A-Prolog. This includes the causal effects of actions (" statement F becomes true as a result of performing an action a"), statements expressing the lack of information ("It is not known if statement P is true or false"), various completeness assumptions, "Statements not entailed by the knowledge base are false", etc. On the negative side, A-Prolog in its current form is not adequate for reasoning with real numbers and for reasoning with complex logical formulas - the things classical logic is good at.

There is by now a comparatively large number of inference engines associated with A-Prolog. There are well known conditions which guarantee that the traditional SLDNF-resolution based goal-oriented methods of "classical" Prolog and its variants are sound with respect to various semantics of logic programming [86, 27, 45, 1]. All of these semantics are sound with respect to the semantics of A-Prolog, i.e. if a program Π is consistent under the answer set semantics and Π entails a literal l under one of these semantics then Π entails l in A-Prolog. This property allows the use of SLDNF based methods for answering A-Prolog's queries. Similar observations hold for bottom up methods of computation used in deductive databases. The newer methods (like that of [14]) combine both, bottom-up and top-down, approaches. A more detailed discussion of these matters can be found in [48]. In the last few years we witnessed the coming of age of inference engines aimed at computing answer sets (stable models) of programs of A-Prolog [65, 66, 21, 17]. The algorithms implemented in these engines have much in common with more traditional satisfiability algorithms. The additional power comes from the use of techniques from deductive databases, good understanding of the relationship between various semantics of logic programming and other more recent discoveries (see for instance [44]) These engines are of course applicable only to programs with finite Herbrand universes. Their efficiency and power combined with so called *answer sets programming paradigm* [64], [56] lead to the development of A-Prolog based solutions for various problems in several knowledge intensive domains [77, 7, 25].

This paper is an attempt to introduce the reader to some recent developments in theory and practice of A-Prolog. In section 2 we briefly review the syntax and the semantics of the basic version of A-Prolog. In section 3 A-Prolog will be used to gradually construct a knowledge base which will demonstrate some knowledge of the notion of *orphan*. The main goal of this, rather long, example is to familiarize the reader with basic methodology of representing knowledge in A-Prolog. Section 4 contains some recent results from the mathematical theory of the language. The selection, of course, strongly reflects personal taste of the author and the limitations of space and time. Many first class recent results are not even mentioned. I hope however that the amount of material is sufficient to allow the reader to form a first impression and to get some appreciation of the questions involved. Section 5 contains a brief introduction to two extensions of the basic language: A-Prolog with disjunction and A-Prolog with sets. The latter is the only part of this paper which was neither published nor discussed in a broad audience. Again the purpose is primarily to illustrate the power of the basic semantics and the ease of adding extensions to the language. Finally, section 6 deals with more advanced knowledge representation techniques and more complex reasoning problems. There are several other logical languages and reasoning methods which can be viewed as alternatives to A-Prolog (see for instance [1, 12, 40]). They were developed in approximately the same time frame as A-Prolog share the same roots and a number of basic ideas. The relationship and mutual fertilization between these approaches is a fascinated subject which goes beyond the natural boundaries of this paper.

2 Syntax and Semantics of the Language

In this section we give a brief introduction to the syntax and semantics of a comparatively simple variant of A-Prolog. Two more powerful dialects will be discussed in sections 5. The syntax of the language is determined by a signature σ consisting of types, $types(\sigma) = \{\tau_0, \ldots, \tau_m\}$, object constants $obj(\tau, \sigma) = \{c_0, \ldots, c_m\}$ for each type τ, and typed function and predicate constants $func(\sigma) = \{f_0, \ldots, f_k\}$ and $pred(\sigma) = \{p_0, \ldots, p_n\}$. We will assume that the signature contains symbols for integers and for the standard relations of arithmetic. Terms are built as in typed first-order languages; positive literals (or atoms) have the form $p(t_1, \ldots, t_n)$, where t's are terms of proper types and p is a predicate symbol of arity n; negative literals are of the form $\neg p(t_1, \ldots, t_n)$. In our further discussion we often write $p(t_1, \ldots, t_n)$ as $p(\bar{t})$. The symbol $\neg$ is called *classical* or *strong* negation.[2] Literals of the form $p(\bar{t})$ and $\neg p(\bar{t})$ are called contrary. By $\bar{l}$ we denote a literal contrary to l. Literals and terms not containing variables are called *ground*. The sets of all ground terms, atoms and literals over

[2] Logic programs with two negations appeared in [31] which was strongly influenced by the epistemic interpretation of logic programs given below. Under this view $\neg p$ can be interpreted as "believe that p is false" which explains the term "classical negation" used by the authors. Different view was advocated in [67, 87] where the authors considered logic programs without negation as failure but with $\neg$. They demonstrated that in this context logic programs can be viewed as theories of a variant of intuitionistic logic with strong negation due to [63]. For more recent work on this subject see [68]. I believe that both views proved to be fruitful and continue to play an important role in our understanding of A-Prolog. A somewhat different view on the semantics of programs with two negations can be found in [1].

σ will be denoted by $terms(\sigma)$, $atoms(\sigma)$ and $lit(\sigma)$ respectively. For a set P of predicate symbols from σ, $atoms(P, \sigma)$ ($lit(P, \sigma)$) will denote the sets of ground atoms (literals) of σ formed with predicate symbols from P. Consistent sets of ground literals over signature σ, containing all arithmetic literals which are true under the standard interpretation of their symbols, are called *states* of σ and denoted by $states(\sigma)$.

A rule of A-Prolog is an expression of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{1}$$

where $n \geq 1$, l_i's are literals, l_0 is a literal or the symbol $\perp$, and *not* is a logical connective called *negation as failure* or *default negation*. An expression *not* l says that there is no reason to believe in l. An *extended literal* is an expression of the form l or *not* l where l is a literal. A rule (1) is called a *constraint* if $l_0 = \perp$.

Unless otherwise stated, we assume that the $l's$ in rules (1) are ground. Rules with variables (denoted by capital letters) will be used only as a shorthand for the sets of their ground instantiations. This approach is justified for the so called closed domains, i.e. domains satisfying the domain closure assumption [72] which asserts that *all objects in the domain of discourse have names in the language of Π*. Even though the assumption is undoubtedly useful for a broad range of applications, there are cases when it does not properly reflect the properties of the domain of discourse. Semantics of A-Prolog for open domains can be found in [8], [41].
A pair $\langle \sigma, \Pi \rangle$ where σ is a signature and Π is a collection of rules over σ is called a *logic program.* (We often denote such pair by its second element Π. The corresponding signature will be denoted by $\sigma(\Pi)$.)

The following notation will be useful for the further discussion. A set
$not\ l_i, \ldots, not\ l_{i+k}$ will be denoted by $not\ \{l_i, \ldots, l_{i+k}\}$. If r is a rule of type (1) then $head(r) = \{l_0\}$, $pos(r) = \{l_1, \ldots, l_m\}$, $neg(r) = \{l_{m+1}, \ldots, l_n\}$, and $body(r) = pos(r), not\ neg(r)$. The head, $\perp$, of a constraint rule will be frequently omitted. Finally, $head(\Pi) = \bigcup_{r \in \Pi} head(r)$. Similarly, for *pos* and *neg*.

We say that a literal $l \in lit(\sigma)$ is $true$ in a state X of σ if $l \in X$; l is $false$ in X if $\bar{l} \in X$; Otherwise, l is unknown. $\perp$ is false in X.

The answer set semantics of a logic program Π assigns to Π a collection of *answer sets* – consistent sets of ground literals over signature $\sigma(\Pi)$ corresponding to beliefs which can be built by a rational reasoner on the basis of rules of Π. In the construction of these beliefs the reasoner is assumed to be guided by the following informal principles:

- He should satisfy the rules of Π, understood as constraints of the form: *If one believes in the body of a rule one must belief in its head.*
- He cannot believe in $\perp$ (which is understood as falsity).
- He should adhere to the *rationality principle* which says that *one shall not believe anything he is not forced to believe.*

The precise definition of answer sets will be first given for programs whose rules do not contain default negation. Let Π be such a program and let X be a state of $\sigma(\Pi)$. We say that X is *closed* under Π if, for every rule $head \leftarrow body$ of Π, head is true in X whenever *body* is true in X. (For a constraint this condition means that the body is not contained in X.)

Definition 1. *(Answer set – part one)*
A state X of $\sigma(\Pi)$ is an *answer set* for Π if X is minimal (in the sense of set-theoretic inclusion) among the sets closed under Π.

It is clear that a program without default negation can have at most one answer set. To extend this definition to arbitrary programs, take any program Π, and let X be a state of $\sigma(\Pi)$. The *reduct*, Π^X, of Π relative to X is the set of rules

$$l_0 \leftarrow l_1, \ldots, l_m$$

for all rules (1) in Π such that $l_{m+1}, \ldots, l_n \notin X$. Thus Π^X is a program without default negation.

Definition 2. *(Answer set – part two)*
A state X of $\sigma(\Pi)$ is an answer set for Π if X is an answer set for Π^X.

(The above definition differs slightly from the original definition in [31], which allowed the inconsistent answer set, $lit(\sigma)$. Answer sets defined in this paper correspond to consistent answer sets of the original version.)

Definition 3. *(Entailment)*
A program Π entails a literal l ($\Pi \models l$) if l belongs to all answer sets of Π.
The Π's answer to a query l is *yes* if $\Pi \models l$, *no* if $\Pi \models \bar{l}$, and *unknown* otherwise.

Consider for instance a logic program[3]

$$\Pi_0 \begin{cases} p(a) \ \leftarrow not\ q(a). \\ p(b) \ \leftarrow not\ q(b). \\ q(a). \end{cases}$$

It has one answer set $\{q(a), p(b)\}$ and thus answers *yes* and *unknown* to queries $q(a)$ and $q(b)$ respectively. If we expand Π_0 by a rule

$$\neg q(X) \leftarrow not\ q(X). \tag{2}$$

the resulting program Π_1 would have the answer set $S = \{q(a), \neg q(b), p(b)\}$ and hence its answer to the query $q(b)$ would be *no*.

Rule (2), read as "*if there is no reason to believe that X satisfies q then it does not*" is called the *closed world assumption* for q [72]. It guarantees that the reasoner's beliefs about q are complete, i.e. for any ground term t and every answer set S of the corresponding program, $q(t) \in S$ or $\neg q(t) \in S$.

[3] Unless otherwise specified we assume that signature of a program consists of symbols occurring in it.

The programs may have one, many, or zero answer sets. It is easy to check for instance that programs

$$\Pi_3 = \{p \leftarrow not\ p\} \text{ and } \Pi_4 = \{p. \quad \neg p.\}$$

have no answer sets while program Π_5

```
    $e(0).$
    $e(s(s(X))) \leftarrow not\ e(X).$
    $p(s(X))$     $\leftarrow e(X),$
                  $not\ p(X).$
    $p(X)$        $\leftarrow e(X),$
                  $not\ p(s(X)).$
```

has an infinite collection of them.

In some cases a knowledge representation problem consists in representing a (partial) definition of new relations between objects of the domain in terms of the old, known relations. Such a definition can be mathematically described by logic programs viewed as functions from states of some input signature σ_i (given relations) into states of some output signature σ_i (defined relations).[4] More precisely [10].

Definition 4. *(lp-functions)*
A four-tuple $f = \langle \Pi(f), \sigma_i(f), \sigma_o(f), dom(f) \rangle$ where

1. $\Pi(f)$ is a logic program (with some signature σ),
2. $\sigma_i(f)$ and $\sigma_o(f)$ are sub-signatures of σ, called the input and output signatures of f respectively,
3. $dom(f)$ is a collection of states of $\sigma_i(f)$

is called *lp-function* if for any $X \in dom(f)$ program $\Pi(f) \cup X$ is *consistent*, i.e., has an answer set.

For any $X \in dom(f)$, $f(X) = \{l : l \in lit(\sigma_o(f)), \Pi(f) \cup X \models l\}$.

We finish our introduction to A-Prolog by recalling the following propositions which will be useful for our further discussion. To the best of my knowledge Proposition 1 first appeared in [54].

Proposition 1. For any answer set S of a logic program Π:

(a) For any ground instance of a rule of the type (1) from Π,

if $\{l_1, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$ then $l_0 \in S$.

(b) If $l_0 \in S$, then there exists a ground instance of a rule of the type (1) from Π such that $\{l_1, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$.

[4] This view is similar to that of databases where one of the most important knowledge representation problems consists in defining the new relations (views) in terms of the basic relations stored in the database tables. Unlike our case, however, databases normally assume the completeness of knowledge and hence only need to represent positive information. As a result, database views can be defined as functions from sets of atoms to sets of atoms.

The next proposition (a variant of a similar observation from [31]) shows how programs of A-Prolog can be reduced to *general logic programs*, i.e. programs containing neither $\neg$ nor $\bot$. We will need the following notation:

For any predicate p occurring in Π, let p' be a new predicate of the same arity. The atom $p'(\bar{t})$ will be called the *positive form* of the negative literal $\neg p(\bar{t})$. Every positive literal is, by definition, its own positive form. The positive form of a literal l will be denoted by l^+. Π^+ stands for the general logic program obtained from Π by replacing each rule (1) by

$$l_0^+ \leftarrow l_1^+, \ldots, l_m^+, not\ \ l_{m+1}^+, \ldots, not\ \ l_n^+$$

and adding the rules

$$\leftarrow p(\bar{t}), p'(\bar{t})$$

for every atom $p(\bar{t})$ of $\sigma(\Pi)$. For any set S of literals, S^+ stands for the set of the positive forms of the elements of S.

Proposition 2. A consistent set $S \subset lit(\sigma(\Pi))$ is an answer set of Π if and only if S^+ is an answer set of Π^+.

Proposition 2 suggests the following simple way of evaluating queries in A-Prolog. To obtain an answer for query p, run queries p and p' on the program Π^+. If Π^+'s answer to p is *yes* then Π's answer to p is *yes*. If Π^+'s answer to p' is *yes* then Π's answer to p is *no*. Otherwise the answer to p is *unknown*. (The method of course works only if the corresponding inference engine terminates).

3 Defining Orphans - a Case Study

In this section we give a simple example of representing knowledge in A-Prolog. We will be dealing with a class of "personnel" systems whose background knowledge consist of collections of personal records of people. Such collections will be referred to as *databases*. There are multiple ways of designing such records. To keep a presentation concise we fix an artificially simple signature σ_i containing names of people, a special constant *nil* (read as *unknown person*), and the predicate symbols $person(P)$, $father(F, P)$, $mother(M, P)$, $child(P)$, $dead(P)$. We assume that *every person in the domain has a database record not containing false information, names of the parents of the live people are known and properly recorded, while unknown parents are represented by nil*, and that *the death records and children's records are complete.*The set of databases satisfying these assumptions will be denoted by C_0. Typical records of a database from C_0 look as follows:

$person(john)$.	$person(mike)$.	$person(kathy)$.
$father(mike, john)$.	$father(sam, mike)$.	$father(nil, kathy)$.
$mother(kathy, john)$.	$mother(mary, mike)$.	$mother(pat, kathy)$.
	$dead(mike)$.	$dead(kathy)$.
$child(john)$.		

The first record describes a child, John, whose parents are Mike and Kathy. Since the death of John is not recorded he must be alive. Similarly, we can conclude that Mike and Kathy were adults when they died, and that the name of Kathy's father is unknown.

Let us assume that we are confronted with a problem of expanding databases from C_0. In particular we need to familiarize the system with a notion of an *orphan* - a child whose parents are dead. In slightly more precise terms we need to define a function which takes a database $X \in C_0$ describing personal records of people from some domain and returns the set of the domain's orphans.

The problem can be solved by introducing an lp-function f_0 with $dom(f_0) = C_0$, $\Pi(f_0)$ consisting of rules:

$$\Pi(f_0) \begin{cases} r1. & orphan(P) & \leftarrow child(P), \\ & & \quad not\ dead(P), \\ & & \quad parents\_dead(P). \\ & & \\ r2. & parents\_dead(P) & \leftarrow father(F, P). \\ & & \quad mother(M, P), \\ & & \quad dead(F), \\ & & \quad dead(M). \end{cases}$$

with $father$, $mother$, $dead$ and $child$ being predicate symbols of $\sigma_i(f_0)$, $orphan$ being the only predicate symbol of $\sigma_o(f_0)$, and both signatures sharing the same object constants. It is not difficult to convince oneself that, since X contains complete information about the live people of the domain, set $f_0(X)$ consists exactly of the domain's orphans.

Program $\Pi(f_0)$ has many attractive mathematical and computational properties. For instance it is easy to check that, for any database $X \in C_0$, the program $R_0 = \Pi(f_0) \cup X$ is *acyclic* [3], i.e. there is a function $||\ ||$ from ground atoms of $\sigma(R_0)$ to natural numbers [5] such that for any atom l occurring in the body of a rule with the head l_0, $||l_0|| > ||l||$. Acyclic general logic programs have unique answer sets which can be computed by a bottom-up evaluation [3]. Moreover, acyclicity of R_0 together with some results from [4, 79] guarantee that the SLDNF resolution based interpreter of Prolog will always terminate on atomic queries and (under the 'right' interpretation) produce the intended answers. The reference to the 'right' interpretation is of course vague and deserves some comments. Suppose that, according to the database, X_0, containing records about John and Mary, John is an orphan and Mary is not. Given program R_0 the Prolog interpreter will answer queries $orphan(john)$ and $orphan(mary)$ by *yes* and *no* respectively. Since the closed world assumption is built in the semantics of "classical" logic programming, the second answer can be (correctly) interpreted as saying that Mary is not an orphan. It is important to realize however that,

[5] Functions from ground literals to ordinals are called *level mappings*. They often play an important role in characterizing various properties of logic programs.

from the standpoint of the semantics of A-Prolog, this interpretation is incorrect. Since neither $orphan(mary)$ nor $\neg orphan(mary)$ is entailed by the program the answer to the query $orphan(mary)$ should be *unknown*. To get the correct answer we need to complete the rules of Π_0 by explicitly defining non-orphans. This can be done by adding a simple rule encoding the corresponding closed world assumption:

$$\begin{aligned} r3.\ \ \neg orphan(P) \leftarrow\ & person(P), \\ & not\ orphan(P). \end{aligned}$$

It may be instructive at this point to modify our notion of a database X by explicitly defining its negative information. For relations *dead* and *child* and it easy: we just need to explicitly encode the closed world assumptions:

$$\begin{aligned} r4.\ \ \neg child(P) \leftarrow\ & person(P), \\ & not\ child(P). \\ r5.\ \ \neg dead(P) \leftarrow\ & person(P), \\ & not\ dead(P). \end{aligned}$$

Even though we typically have complete information about the parents of people from the database this is not always the case. We can express this fact by the following default with exceptions:

$$\begin{aligned} r6.\ \ \neg father(F,P) \leftarrow\ & person(F), \\ & person(P), \\ & not\ father(F,P), \\ & not\ ab(d(F,P)). \\ r7.\ \ ab(d(F,P)) \quad \leftarrow\ & father(nil,P). \end{aligned}$$

Here $d(F,P)$ is used to name the default; statement $ab(d(F,P))$ says that this default is not applicable to F and P. If we assume that Bob is a person in our database we will be able to use the default to show that Bob is not the father of John. For Kathy, however, the same question will remain undecided.

Rules (r6) and (r7) can be viewed as a result of the application of the general methodology of representing defaults in A-Prolog. More detailed discussion of this methodology can be found in [8]. A more general approach which provides means for specifying priorities between defaults is discussed [36], [23], [39].

Let $X \in C_0$. Since X contains the complete records of parents of every live person p the rules (r6) and (r7) allow us to conclude that for every person r different from the father of P the answer to query $father(r,p)$ will be *no*. For dead people more negative knowledge can be extracted from the database by common-sense rules like:

$$\begin{aligned} r8.\ \ & \neg father(F,P) && \leftarrow mother(F,Q). \\ r9.\ \ & \neg father(F,P) && \leftarrow descendant(F,P). \\ r10.\ \ & descendent(P,P). && \\ r11.\ \ & descendent(D,P) && \leftarrow parent(P,C), \\ & && \quad\ descendant(D,C). \end{aligned}$$

Consider an lp-function

$$g = \langle \Pi(g), \sigma_i(f_0), \sigma_i(f_0), C_0 \rangle$$

where $\Pi(g)$ consists of rules (r4)-(r11), together with the obvious definition of relation *parent* and the rules extracting negative information for mothers. The function computes the *completion* of a database $X \in C_0$ by the corresponding negative information. By $\hat{C}_0$ we denote the collection of completions of elements of C_0. Consider

$$\Pi(f) = \Pi(f_0) \cup (r3)$$

and lp-functions

$$f = \langle \Pi(f_0) \cup (r3), \sigma_i(f_0), \sigma_o(f_0), \hat{C}_0 \rangle$$

and

$$h = \langle \Pi(f) \cup \Pi(g), \sigma_i(f_0), \sigma_o(f_0), C_0 \rangle$$

Using the Splitting Lemma (see the next section) it is not difficult to show that, for any $X \in C_0$, $h(X) = f(g(X))$, i.e. $h = f \circ g$. (Notice that, since $\Pi(g)$ is nonmonotonic, its consequences can be modified by addition of $\Pi(f)$ and so such a proof is necessary. Fortunately, it follows immediately from a fairly general theorem from [28].)

Due to the use of default negation, h is also elaboration tolerant w.r.t. *some* modifications of the background knowledge such as addition of new people and recording of deaths and changes in the adulthood status. The latter for instance can be accomplished by simply removing, say, a record $child(john)$ from the background knowledge X_0 described above, at which point $John$ will seize to be an orphan. A program will continue to work correctly as far as the update of the background knowledge still belongs to the class C_0.

When our knowledge of the domain cannot be captured by databases from C_0 or $\hat{C}_0$ the situation may become substantially more complex. Let us for instance consider a modification of our informal knowledge base by *removing from it the closed world assumption for property of being a child* ($cwa(child)$). Now the record of a person p can contain a statement $child(p)$ or a statement $\neg child(p)$, or no information about p being a child at all. (In the latter case we say that p's age is unknown.) A new class of databases will be denoted by C_1. As before, every database X of C_1 contains atoms $alive(p)$, $father(f,p)$, $mother(m,p)$ for every live person p of the domain. We still have the closed world assumption for *alive* and no false information in the X's records. Our goal is still to teach our knowledge base about the orphans, i.e. to construct an lp-function which takes a database $X \in C_1$ and returns the set of domain's orphans. It is easy to see that completions of databases from C_1 with respect to missing negative information about relations other than *child* can be defined as values of the lp-function g' obtained from g by removing $cwa(child)$ from $\Pi(g)$. We denote the set of all such completions by $\hat{C}_1$. As expected, however, program f does not work correctly with databases from $\hat{C}_1$ – the closed world assumption for orphans will force the program to erroneously conclude that everyone whose age is not known is not an orphan.

The problem of finding a uniform way of modifying logic programs which would reflect the removal of some of its closed world assumptions was addressed in [10], [28]. The authors' approach is based on the notion of *interpolation* of a logic program. To be more precise we will need some additional terminology.

Let F be an lp-function, O be a set of predicate symbols from $\sigma_i(F)$ and $D = dom(F)$ be closed with respect to O, i.e., $X \in D$ contains l or $\bar{l}$ for every literal $l \in lit(O)$. For any set X of input literals we define the set $c(X, O)$ of its *covers* – $\hat{X} \in c(X, O)$ if it satisfies the following properties:

1. $\hat{X} \in D$;
2. $X \subseteq \hat{X}$;
3. for every input literal $l \notin lit(O)$, $l \in X$ iff $l \in \hat{X}$.

By $\tilde{D}$ we denote the set of states of $\sigma_i(F)$ such that

$$X = \bigcap_{\hat{X} \in c(X,O)} \hat{X}$$

Definition 5. We say that an lp-function $\tilde{F}$ is an *O-interpolation* of F if

$$dom(\tilde{F}) = \tilde{D}$$

$$\tilde{F}(X) = \bigcap_{\hat{X} \in c(X,O)} F(\hat{X})$$

$$\sigma_i(F) = \sigma_i(\tilde{F}) \text{ and } \sigma_o(F) = \sigma_o(\tilde{F})$$

Let us go back to function f from our example and consider $O = \{child\}$, program

$$\Pi(\tilde{f}) \begin{cases} 1.\ may\_be\_child(P) & \leftarrow not\ \neg child(P). \\ 2.\ parents\_dead(P) & \leftarrow father(F,P). \\ & \quad mother(M,P), \\ & \quad dead(F), \\ & \quad dead(M). \\ 3.\ orphan(P) & \leftarrow child(P), \\ & \quad not\ dead(P), \\ & \quad parents\_dead(P). \\ 4.\ may\_be\_orphan(P) & \leftarrow may\_be\_child(P), \\ & \quad not\ dead(P), \\ & \quad parents\_dead(P). \\ 5.\ \neg orphan(P) & \leftarrow not\ may\_be\_orphan(P). \end{cases}$$

and lp-function

$$\tilde{f} = \langle \Pi(\tilde{f}), \sigma_i(f), \sigma_o(f), \hat{C}_1 \rangle$$

The rules of $\Pi(\tilde{f})$ are obtained by the general algorithm from [10] which, under certain conditions, translates lp-functions described by general logic programs into their interpolations. In the next section we use the mathematical theory of A-Prolog to prove that $\tilde{f}$ is indeed a $\{child\}$-interpolation of f.

In the conclusion of this section we illustrate how A-Prolog can be used to represent

(a) simple priorities between defaults;

(b) statements about the lack of information.

To do that, let us supply our program with knowledge about some fictitious legal regulations. The first regulation says that *orphans are entitled to assistance according to special government program 1*, while the second says that all *children who are not getting any special assistance are entitled to program 0.* Legal regulations always come with exceptions and hence can be viewed as defaults. We represent both regulations by the following rules:

$$\begin{aligned} entitled(P,1) \leftarrow\ & orphan(P),\\ & not\ ab(d_1(P)),\\ & not\ \neg entitled(P,1).\\[1em] entitled(P,0) \leftarrow\ & child(P),\\ & \neg dead(P),\\ & not\ ab(d_2(P)),\\ & not\ \neg entitled(P,0).\\[1em] ab(d_2(P)) \quad\ \leftarrow\ & orphan(P). \end{aligned}$$

The first two rules are standard representations of defaults. The last rule says that the default d_2 is not applicable to orphans. Notice that if Joe is a child and it is not known whether he is an orphan or not then Joe will receive benefits from program 0 but not from program 1. This case of insufficient documentation can be detected by the following rule:

$$\begin{aligned} check\_status(P) \leftarrow\ & person(P),\\ & not\ \neg orphan(P),\\ & not\ orphan(P). \end{aligned}$$

Though simple, the program above illustrates many interesting features of A-Prolog: recursive rules, the use of default negation for representing defaults with exceptions, the use of both negations in formulating the closed world assumptions, the ability to discriminate between falsity and the absence of information, and to produce conclusions based on such absence. The program can be used together with various inference engines of A-Prolog, thus making it (efficiently) executable. In section 6 we will demonstrate how A-Prolog can be used to represent change and causal relations. First, however, we briefly discuss mathematical theory of A-Prolog.

4 Mathematics of A-Prolog

In this section we review several important properties of programs of A-Prolog. Our goal of course is not to give a serious introduction into the mathematics of A-Prolog. By now the theory is well developed, contains many interesting results, and probably deserves a medium size textbook. Instead we concentrate on a few important discoveries and discuss their relevance to constructing knowledge bases.

4.1 Splitting Lemma

The structure of answer sets of a program Π can sometimes be better understood by "splitting" the program into parts. We say that a set U of literals *splits* a program Π if, for every rule r of Π, $pos(r) \cup neg(r) \subseteq U$ whenever $head(r) \in U$. If U splits Π then the set of rules in Π whose heads belong to U will be called the base of Π (relative to U). We denote the base of Π by $b_U(\Pi)$. The rest of the program (called the top of Π) will be denoted by $t_U(\Pi)$.

Consider for instance a program Π_1 consisting of the rules

$$\begin{array}{l} q(a) \leftarrow not\ q(b), \\ q(b) \leftarrow not\ q(a), \\ r(a) \leftarrow q(a). \\ r(a) \leftarrow q(b) \end{array}$$

Then, $U = \{q(a), q(b)\}$ is a splitting set of Π_1, $b_U(\Pi_1)$ consists of the first two rules while $t_U(\Pi_1)$ consists of the last two.

Let U be a splitting set of a program Π and consider $X \subseteq U$. For each rule $r \in \Pi$ satisfying property

$$pos(r) \cap U \subset X \text{ and } (neg(r) \cap U) \cap X = \emptyset$$

take the rule r' such that

$$head(r') = head(r),\ pos(r') = pos(r) \setminus U,\ neg(r') = neg(r) \setminus U$$

The resulting program, $e_U(\Pi, X)$, is called *partial evaluation* of Π with respect to U and X.

A *solution* to Π with respect to U is a pair $\langle X, Y \rangle$ of sets of literals such that:

- X is an answer set for $b_U(\Pi)$;
- Y is an answer set for $e_U(t_U(\Pi), X)$;
- $X \cup Y$ is consistent.

Lemma 1. *(Splitting Lemma)*
Let U be a splitting set for a program Π. A set S of literals is a consistent answer set for Π if and only if $S = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π w.r.t. U.

The Splitting Lemma has become an important tool for establishing existence and other properties of programs of A-Prolog. To demonstrate its use let us consider a class of finite *stratified* programs. A finite general logic program Π is called stratified if there is a level mapping $|| \; ||$ of Π such that if $r \in \Pi$ then

1. For any $l \in pos(r)$, $||l|| \leq ||head(r)||$;
2. For any $l \in neg(r)$ $||l|| < ||head(r)||$.

This is a special case of the notion of stratified logic program introduced in [2]. The results of that paper together with those from [29] imply that a stratified program has exactly one answer set. For finite stratified programs this can be easily proven by induction on the number of levels of Π with the use of the Splitting Lemma. If Π has one level (i.e. $||l|| = 0$ for every $l \in \sigma(\Pi)$) then Π does not contain default negation and hence, by [85] has exactly one minimal Herbrand model which, by definition, coincides with the Π's answer set. If the highest level of an atom from $\sigma(\Pi)$ is $n+1$ then it suffices to notice that atoms with smaller levels form a splitting set U of Π. By inductive hypothesis, $b_U(\Pi)$ has exactly one answer set, X, $e_U(\Pi, X)$ is a program without *not* and hence has one and only answer set Y. By Splitting Lemma $X \cup Y$ is the only answer set of Π.

The Splitting Lemma can be generalized to programs with a monotone, continuous sequence of splitting sets. This more powerful version can be used to prove the uniqueness of answer set for locally stratified logic programs [69], existence of answer sets for order-consistent logic programs of [26], etc.

The above results, combined with Proposition 2 can be used to establish existence and uniqueness of answer sets of programs with $\neg$. Consider for instance lp-function h from the previous section and a set X of literals from C_0. To show that $\Pi(h) \cup X$ has the unique answer set let us notice that the corresponding general logic program $(\Pi(h) \cup X)^+$ is stratified, and therefore has the unique answer set S^+. To show that the corresponding set S is consistent we need to check that there is no atom $p(\bar{t})$ such that $p(\bar{t}), (\neg p(\bar{t}))^+ \in S^+$. By Proposition 1 we have that this could only happen if for some people f and p, $father(f,p)$ and $mother(f,p)$ or $father(f,p)$ and $descendent(f,p)$ were in S^+. It is not difficult to check that this is impossible since, according to our assumption, C_0 contains correct factual information. This, by Proposition 2, implies that $\Pi(h) \cup X$ has the unique answer set S.

The discussion in this section follows [50], in which authors gave a clear exposition of the idea of splitting in the domain of logic programs. Independently, similar results were obtained in [16]. There is a very close relationship between splitting of logic programs and splitting of autoepistemic and default theories [34, 15, 84].

4.2 Signed Programs

In this section we introduce the notion of *signing* of a program of A-Prolog. The notion of signing for finite general logic programs was introduced by Kunen [46], who used it as a tool in his proof that, for a certain class of program, two different semantics of logic programs coincide. Turner, in [82], extends the definition to the class of logic programs with two kinds of negation and investigates properties of signed programs. We will need some terminology.

The *absolute value* of a literal l (symbolically, $|l|$) is l if l is positive, and $\bar{l}$ otherwise.

Definition 6. A *signing* of logic program Π is a set $S \subseteq atoms(\sigma(\Pi))$ such that

1. for any rule
$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$$
from Π, either
$$|l_0|, \ldots, |l_m| \in S \text{ and } |l_{m+1}|, \ldots, |l_n| \notin S$$
or
$$|l_0|, \ldots, |l_m| \notin S \text{ and } |l_{m+1}|, \ldots, |l_n| \in S$$
;
2. for any atom $l \in S$, $\neg l$ does not appear in Π.

A program is called *signed* if it has a signing. Obviously, programs without default negation are signed with the empty signing. Program

$$\begin{array}{ll} p(a) & \leftarrow not\ q(a). \\ p(b) & \leftarrow not\ q(b). \\ q(a). & \end{array}$$

is signed with signing $\{q(a), q(b)\}$.

Program Π_1 with signature $\sigma = \{\{a, b, c\}, \{p, q, r, ab\}\}$ and the rules

$$\begin{array}{ll} q(X) & \leftarrow p(X), \\ & \quad not\ ab(X). \\ \neg q(X) & \leftarrow r(X). \\ ab(X) & \leftarrow not\ \neg r(X). \end{array}$$

is signed with a signing $atoms(ab, \sigma)$.

Signed programs enjoy several important properties which make them attractive from the standpoint of knowledge representation. In particular,

1. Signed general logic programs are consistent, i.e. have an answer set. Simple consistency conditions can also be given for signed programs with classical negation.

2. If Π is consistent then the set of consequences of the program under answer set semantics coincides with its set of consequences under well-founded semantics [86]. Notice, that this result shows that inference engines such as SLG [14] which compute the well founded semantics of logic programs, can also be used to compute the consequences of such programs under the answer set semantics.

The following theorem gives another important property of signed programs:

An lp-function F is called *monotonic* if for any $X, Y \in dom(F)$, $F(X) \subseteq F(Y)$.

Theorem 1. *(Monotonicity Theorem, Turner)*
If an lp-function F has a signing S such that $S \cap (lit(\sigma_i(F)) \cup lit(\sigma_o(F))) = \emptyset$ then F is monotonic.

Example 1. Consider an lp-function f_1 with $\sigma_i(f_1) = \{\{a, b, c\}, \{p, r\}\}$, $\sigma_o(f_1) = \{\{a, b, c\}, \{q\}\}$, $dom(f_1)$ consisting of consistent sets of literals in σ_i, and program Π_1 above as $\Pi(f_1)$. It is easy to see that lp-function f_1 satisfies the condition of theorem 1, and hence is monotonic. It's worth noticing, that logic program $\Pi(f_1)$ is nonmonotonic. Addition of extra rules (or facts) about ab can force us to withdraw previous conclusions about q. Monotonicity is however preserved for inputs from σ_i.

Discussion of the importance of this property for knowledge representation can be found in [47].

4.3 Interpolation

In the previous section we mentioned the notion of interpolation $\tilde{F}$ of an lp-function F. The switch from F to $\tilde{F}$ reflects the removal from the informal knowledge base represented by $\Pi(F)$ some of its closed world assumptions. The notion of signing plays an important role in the following theorem (a variant of the result from [10]) which facilitates the construction of $\tilde{F}$.

Let F be an lp-function with $\Pi(F)$ not containing $\neg$, O be a set of predicate symbols from $\sigma_i(F)$ and the domain D of F be closed with respect to O, i.e., $X \in D$ contains l or $\bar{l}$ for every literal $l \in lit(O)$. By o we denote the set of predicate symbols of $\Pi(F)$ depending on O. More precisely, o is the minimal set of predicate symbols such that $O \subseteq o$ and if the body of a rule of $\Pi(F)$ with head $p(\bar{t})$ contains an atom formed by a predicate symbol from o, then $p \in o$. To define $\tilde{F}$ we expand the signature σ of $\Pi(F)$ by a new atom, m_p, (read as "may be p") for every predicate symbol p, and consider a mapping, α, from extended literals of $\Pi(F)$ into literals of the new signature $\tilde{\sigma}$:

1. if $p \in o$ then $\alpha(p(\bar{t})) = m_p(\bar{t})$ and $\alpha(not\ p(\bar{t})) = \neg p(\bar{t})$.
2. Otherwise, $\alpha(e) = e$.

If E is a set of extended literals then $\alpha(E) = \{\alpha(e) : e \in E\}$.

If r is a rule of the form

$$l_0 \leftarrow pos, not\ neg$$

then by $\alpha(r)$ we denote the rules:

$$\begin{array}{ll} \alpha(l_0) & \leftarrow \alpha(pos), not\ neg \\ l_0 & \leftarrow pos, \alpha(not\ neg) \end{array}$$

By $\tilde{F}$ we denote the lp-function with $dom(\tilde{F}) = \tilde{dom}(F)$, $\sigma_i(\tilde{F}) = \sigma_i(F)$, $\sigma_o(\tilde{F}) = \sigma_o(F)$, and $\Pi(\tilde{F})$ consisting of the rules:

1. For any predicate symbol $p \in O$ add the rule

$$m\_p(X) \leftarrow not\ \neg p(X).$$

2. For any predicate symbol $p \in o \setminus O$ add the rule

$$\neg p(X) \leftarrow not\ m_p(X).$$

3. Replace every rule $r \in \Pi$ by $\alpha(r)$.

Theorem 2. *(Interpolation theorem)*
Let F and O be as above. If $\Pi(F)$ is signed then $\tilde{F}$ is an interpolation of F.

Let us now demonstrate how these results can be used to prove properties of the lp-functions f and $\tilde{f}$ from section 3.

Proposition 3. $\tilde{f}$ is the interpolation of f.

Proof (sketch).
(a) Let $O = \{child\}$ and $D = dom(f)$. Using definitions of $\hat{C}_1$ and $\tilde{D}$ it is not difficult to show that $\hat{C}_1 = \tilde{D}$. To check the first condition of Definition 5 we need to prove that for any $X \in \tilde{D}$, $(\Pi(\tilde{f}) \cup X)^+$ has an answer set. This follows from the fact that this program is stratified. Moreover stratifiability implies that this answer set is unique. Let us denote it by A^+. Using Proposition 1 we can check that if $orphan(p) \in A^+$ then so is $may\_be\_orphan(p)$ and therefore A^+ does not contain $(\neg orphan(p))^+$. By Proposition 2 we conclude that A is the answer set of $\Pi(\tilde{f}) \cup X$, i.e., $X \in dom(\tilde{f})$.

(b) Let U be a set of literals formed by predicate symbols of the program $\Pi(f)$ different from *orphan* and *child*. Obviously, for any $X \in \tilde{D}$, U is a splitting set of $\Pi(f) \cup X$. The base of this program, consisting of definition of *parents_dead* and and $X \cap U$ contains no default negation. This, together with consistency of X, implies that the base has exactly one answer set. Let us denote it by A_U.

Now consider an lp-function r with

$$\Pi(r) = e_U(\Pi(f), A_U) \tag{3}$$

$\sigma_i(r)$ formed by predicate *child* and objects constants of the domain, $\sigma_o(r) = \sigma\_o(f)$, and the domain consisting of complete and consistent sets of literals formed by *child*. From definition of our operator $\tilde{}$ it is easy to see that

$$\Pi(\tilde{r}) = e_U(\Pi(\tilde{f}), A_U) \tag{4}$$

It is not difficult to check that $\Pi(r)$ is signed with a signing consisting of atoms formed by the predicate symbol "*may_be*" therefore, by Theorem 2,

$$\tilde{r}(Y) = \bigcap_{\hat{Y} \in c(Y,O)} r(\hat{Y}) \tag{5}$$

By Splitting Lemma we can conclude that

$$f(X) = \tilde{r}(Y) \text{ where } Y = X \cap lit(O) \tag{6}$$

and that, for every $\hat{X} \in c(X, O)$,

$$\tilde{f}(\hat{X}) = \tilde{r}(\hat{Y}) \text{ where } \hat{Y} = \hat{X} \cap lit(O) \tag{7}$$

Finally let us notice that

$$\hat{X} \in c(X, O) \text{ iff } \hat{Y} \in c(Y, O) \tag{8}$$

which, together with (5) – (6) implies

$$\tilde{f}(\hat{X}) = \bigcap_{\hat{X} \in c(X,O)} f(\hat{X}) \tag{9}$$

We hope that the discussion in this section will help a reader to get a feel for some of the mathematics of A-Prolog. The following sections contain several other useful mathematical results which may help to better see the variety of questions related to A-Prolog. Meanwhile we turn to the question of the extensions of the basic variant of A-Prolog.

5 Extensions of A-Prolog

There are several important extensions of A-Prolog (see for instance [38, 52, 66]). We will briefly discuss two of such extensions: *disjunctive A-Prolog* (DA-Prolog) [70, 31], and *A-Prolog with sets* (ASET-Prolog). DA-Prolog has been studied for a substantial amount of time. It has a non-trivial theory and efficient implementation. ([61] surveys alternative ways of introducing disjunction in logic programming). ASET-Prolog is still in its developing stage.

5.1 A-Prolog with Disjunction

A program of DA-Prolog consists of rules of the form

$$l_0 \textbf{ or } \ldots \textbf{ or } l_k \leftarrow l_{k+1}, \ldots l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{10}$$

The definition of an answer set of a disjunctive program is obtained by making a small change in the definition of what it means for a set X of literals to be closed under program Π. We now say that X is *closed* under Π if, for every rule *head* $\leftarrow$ *body* of Π, at least one of literals in the head is true in X whenever *body* is true in X. The rest of definitions 1 and 2 remain unchanged. The following simple examples illustrate the definition:
A program Π_1 consisting of the rules:

p_1 **or** p_2.
$q \leftarrow p_1$.
$q \leftarrow p_2$.

has two answer sets, $\{p_1, q\}$ and $\{p_2, q\}$. The program Π_2:

p_1 **or** p_2.
$q \leftarrow not\ p_1$.
$q \leftarrow not\ p_2$.

has the same answer sets. And the program

p_1 **or** p_2.
$q \leftarrow not\ p_1$.
$q \leftarrow not\ p_2$.
$\neg p_1$.

has the answer set $\{\neg p_1, p_2, q\}$.

There are several systems capable of reasoning in DA-Prolog. Some of them use the top-down or bottom-up methods of answering queries similar to those in non-disjunctive logic programs [78, 5]. A different approach is taken by the *dlv* system [20] which takes as an input a program of DA-Prolog with a finite Herbrand Universe and computes the answer sets of this program.

Knowledge Representation in DA-Prolog

The following examples demonstrate the use of disjunction for knowledge representation and reasoning.

Example 2. Let us consider the following scenario: A preliminary summer teaching schedule of a computer science department is described by a relation $teaches(prof, class)$. The preliminary character of the schedule is reflected by the following uncertainty in the database

$teaches(mike, java)$ **or** $teaches(john, java)$.

and by the absence of the closed world assumption for *teaches*. Now assume that in summer semesters the department normally teaches at most one course on computer languages. Intuitively this implies that no course on the C language will be offered. To make such a conclusion possible we expand our database by the following information:

$$\begin{array}{ll}
lang(java). & \\
lang(c). & \\
offered(C) & \leftarrow teaches(P, C). \\
\neg offered(C1) & \leftarrow lang(C1), \\
 & \quad lang(C2), \\
 & \quad offered(C2), \\
 & \quad C1 \neq C2, \\
 & \quad not\ offered(C1).
\end{array}$$

The last statement is a standard representation of a default. The resulting program has two answer sets. In one Java is taught by Mike, in another one by John. In both cases however the C language is not offered. The example demonstrates the ability of DA-Prolog to represent reasoning by cases and to nicely combine disjunction with defaults. (For comparison of these properties with the use of disjunction in Reiter's default logic see [33]).

The next example from [13] demonstrates the expressive power of the language.

Example 3. Suppose a holding owns some companies producing a set of products. Each product is produced by at most two companies. We will use a relation $produced\_by(P, C_1, C_2)$ which holds if a product P produced by companies C_1 and C_2. The holding below consists of four companies producing four products and can be represented as follows:

$$\begin{array}{ll}
produced\_by(p1, b, s). & produced\_by(p2, f, b). \\
produced\_by(p3, b, b). & produced\_by(p4, s, p).
\end{array}$$

This slightly artificial representation, which requires a company producing a unique product to be repeated twice (as in the case of $p3$), is used to simplify the presentation.

Suppose also that we are given a relation $controlled\_by(C_1, C_2, C_3, C_4)$ which holds if companies C_2, C_3, C_4 control company C_1. In our holding, b and s control f, which is represented by

$controlled\_by(f, b, s, s)$

Suppose now that the holding needs to sell some of the companies and that its policy in such situations is to maintain ownership of so called strategic companies, i.e. companies belonging to a minimal (with respect to the set theoretic inclusion) set S satisfying the following conditions:

1. Companies from S produce all the products.
2. S is closed under relation *controlled_by*, i.e. if companies C_2, C_3, C_4 belong to S then so is C_1.

It is easy to see that for the holding above the set $\{b, s\}$ is not strategic while the set $\{b, s, f\}$ is.

Suppose now that we would like to write a program which, given a holding of the above form, computes sets of its strategic companies. This can be done by the rules

$$\begin{array}{lll} 1.\ strat(C_1)\ \mathbf{or}\ strat(C_2) & \leftarrow & produced\_by(P, C_1, C_2) \\ 2.\ strat(C_1) & \leftarrow & controlled\_by(C_1, C_2, C_3, C_4), \\ & & strat(C_2), \\ & & strat(C_3), \\ & & strat(C_4). \end{array}$$

defining the relation $strat(C)$. Let Π be a program consisting of rules (1), (2) and an input database X of the type described above. The first rule guarantees that, for every answer set A of Π and every product p, there is a company c producing p such that an atom $strat(c) \in A$. The second rule ensures that for every answer set of Π the set of atoms of the form $strat(c)$ belonging to this set is closed under the relation *controlled_by*. Minimality of this set follows from the minimality condition in the definition of answer set. It is not difficult to check that answer sets of Π correspond one-to-one to strategic sets of the holding described by an input database. The *dlv* reasoning system can be asked to find an answer set of Π and display atoms of the form *strat* from it.

Complexity and Expressiveness

The above problem can be viewed as an example of a classical search problem P given by a finite collection $dom(P)$ of possible input databases and a function $P(X)$ defining *solutions* of P for every input X from $dom(P)$. An algorithm solves a search problem P if for each $X \in dom(P)$ it returns *no* if $P(X)$ is empty and one of the elements of $P(X)$ otherwise. Solution of the corresponding decision problem requires an algorithm which checks if $P(X)$ is empty or not. This observation suggests the following approach to solving a search and decision problem P:

1. Encode input instances and solutions of P by collections of literals from signatures σ_i and σ_o. (Make sure that the corresponding encoding e is polynomial).
2. Construct a program Π such that for every $X \in dom(P)$ restrictions of answer sets of $\Pi \cup X$ on $lit(\sigma_o)$ correspond to $P(X)$.

If we are successful we say that Π is a *uniform logic programming solution* of P. It is natural to characterize the class of problems which can be solved by this method. First some notation: By FA-Prolog and FDA-Prolog we mean restrictions of A-Prolog and DA-Prolog to languages with finite Herbrand universes.

Theorem 3. *(Complexity results)*

1. The problem of deciding whether a program of FA-Prolog has an answer set is NP-complete [55].

2. A decision problem P can be solved by a uniform program of FA-Prolog iff it is in the class NP [75]
3. A decision problem P can be solved by a uniform program of FDA-Prolog iff it is in the complexity class Σ_2^P [13]

It is interesting to note that the problem from example 3 is Σ_2^P complete [19] and therefore the use of disjunction is essential. The above theorem shows that for decision problems we have a complete answer. The problem remains open for arbitrary search problems but it is clear that both, FA-Prolog and FDA-Prolog can capture a rather large number of such problems. For instance, according to [56], FA-Prolog can solve "all search problems , whose associated decision problems are in NP, that we considered so far".

5.2 A-Prolog with Sets

In this section we introduce a new extension, ASET-Prolog, of A-Prolog which simplifies representation and reasoning with sets of terms and with functions from such sets to natural numbers. The language does not yet have a complete implementation. Fortunately, its semantics is very close to the semantics of *choice* rules of [66], which makes it possible to run a large numbers of programs of ASET-Prolog using *smodels* reasoning system. (In fact, ASET-Prolog is an attempt to simplify and slightly generalize the original work of [66]). We start by defining the syntax and semantics of the language. To simplify the presentation we limit ourself to a language $\mathcal{L}$ without $\neg$ and assume that $\mathcal{L}$ has a finite Herbrand universe. Atoms of $\mathcal{L}$ will be called L-*atoms*. We expand $\mathcal{L}$ by two new types of atoms:

1. An *s-atom* is a statement of the form
$$\{\overline{x} \ : \ p(\overline{x})\} \subseteq \{\overline{x} \ : \ q(\overline{x})\}. \qquad (11)$$
where $\overline{x}$ is the list of all free variables occurring in the corresponding atom. The statement says that p is a subset of q.
2. An *f-atom* is a statement of the form
$$|\{\overline{x} \ : p(\overline{x})\}| \leq n \text{ or } |\{\overline{x} \ : p(\overline{x})\}| = n \qquad (12)$$
where $|\ |$ denotes the cardinality of the corresponding set. (The general description of the language allows other functions on sets except the $|\ |$.)

Let us denote the new language by S.

A *program* of ASET-Prolog (parameterized by a background language S) is a collection of rules of the form
$$l_0 \leftarrow l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n \qquad (13)$$
where $l_1, \dots, l_n$ are atoms of S and l_0 is either L-atom or s-atom of S.

To give a semantics of ASET-Prolog we generalize the notion of stable model of A-Prolog. First we need the following terminology.

Let S be a set of ground atoms of $\mathcal{S}$.

1. An L-atom atom l of $\mathcal{S}$ is true in S if $l \in S$.
2. An s-atom (11) is true in S if for any sequence $\bar{t}$ of ground terms of $\mathcal{L}$, either $p(\bar{t}) \notin S$ or $q(\bar{t}) \in S$.
3. An f-atom (12) is true in S if cardinality of the set $\{\bar{t} \ : \ p(\bar{t}) \in S\}$ satisfies the corresponding condition.

We say that S *satisfies* an atom l of S and write $S \models l$ if l is true in S; *not* l is satisfied by S if $S \not\models l$. As in the definition of stable models, we consider rules of a program Π with free variables to be schemas denoting the set of ground instances of these rules (i.e., the result of replacing free variables of Π by terms of $\mathcal{S}$). Unless stated otherwise Π is assumed to be grounded.

Definition 7. *(Stable models of ASET-Prolog)*
Let S be a collection of ground atoms. By $se(\Pi, S)$ (read as "the set elimination of Π with respect to S") we mean the program obtained from Π by:

1. removing from Π all the rules whose bodies contain s-atoms or f-atoms not satisfied by S;
2. removing all remaining s-atoms and f-atoms from the bodies of the rules;
3. replacing rules of the form $l \leftarrow \Gamma$ where l is an s-atoms not satisfied by S by rules $\leftarrow \Gamma$;
4. Replacing the remaining rules of the form: $\{\overline{x} \ : \ p(\overline{x})\} \subseteq \{\overline{x} \ : \ q(\overline{x}\} \leftarrow \Gamma$ by the rules $p(\bar{t}) \leftarrow \Gamma$ for each $p(\bar{t})$ from S.

We say that S is a stable model of Π if it is a stable model of $se(\Pi, S)$.

Let us now give several examples of the use of ASET-Prolog.

Example 4. (Computing the cardinality of sets)
We are given a complete list of statements of the form $located\_in(C, S)$ (read as "a city C is located in a state S"), e.g.,

$$\begin{array}{l} located\_in(austin, tx). \\ located\_in(lubbock, tx). \\ located\_in(sacramento, ca). \end{array}$$

Suppose that we need to define a relation $num(N, S)$ which holds iff N is the number of cities located in a state S. This can be done with the following rule of ASET-Prolog:

$$num(N, S) \leftarrow |\{X : located\_in(X, S)\}| = N.$$

After grounding, this rule will turn into the rules

$$\begin{array}{l} num(i, tx) \leftarrow |\{X : located\_in(X, tx)\}| = i. \\ num(i, ca) \leftarrow |\{X : located\_in(X, ca)\}| = i. \end{array}$$

where i's are integers from 0 to some maximum integer m. (Notice, that the variable X is bounded and hence it is not replaced by any term.) It is easy to check that the program has exactly one stable model A and that A contains the above facts and the atoms $num(2, tx)$ and $num(1, ca)$.

The next three examples are taken from [66]. They demonstrates the use of rules of the form:

$$\{\overline{x} \ : \ p(\overline{x})\} \subseteq \{\overline{x} \ : \ q(\overline{x}\} \leftarrow \Gamma \tag{14}$$

with s-atoms in the heads. Rules of this form are called *selection* rules and are read as follows: " If Γ holds in a set S of beliefs of an agent then any subset of the set $\{\overline{t} \ : \ q(\overline{t}) \in S\}$ may be the extent of $p(\overline{x})$ in S"[6]. The next example demonstrates the use of selection rules:

Example 5. (Cliques)
Suppose we have a graph defined by the set of facts of the form $node(X)$ and $edge(X, Y)$

$$\begin{array}{l} node(a). \\ node(b). \\ node(c). \\ \\ edge(a, b) \end{array}$$

We would like to define a relation $clique(X)$, i.e. to write a program Π of ASET-Prolog such that for any graph G represented as above, the set of nodes N is a clique of G iff there is a stable model S of $\Pi \cup G$ such that an atom $clique(t) \in S$ iff $t \in N$. Recall that a set of nodes of a graph G is called a clique if *every two nodes from this set are connected by an edge of G.* This can be easily expressed by the following rules:

$$\begin{array}{l} \{X \ : \ clique(X)\} \subseteq \{X \ : \ node(X)\}. \\ \leftarrow clique(X), clique(Y), X \neq Y, not\ edge(X, Y). \end{array}$$

Answer sets of the program consisting of graph G combined with the first rule correspond to arbitrary subsets of nodes of G. Adding the constraint eliminates those which do not form a clique.

The next example demonstrates how selection rules combined with cardinality constraints can allow selection of subsets of given cardinality.

Example 6. (Coloring the graphs)
Suppose we have a graph G defined by the set of facts of the form $node(X)$ and $edge(X, Y)$ as in example (5) together with a set C of colors

$$color(red). \quad color(green). \quad \ldots$$

We would like to *color the graph in a way which guarantees that no two neighboring nodes have the same color.* To this end we introduce a program Π defining a relation $colored(Node, Color)$ such that every coloring will be represented by the atoms of the form $colored(n, c)$ from some stable model of $\Pi \cup G \cup C$. Program

[6] By the extent of $p(\overline{x})$ in S we mean the set of ground terms such that $p(\overline{t}) \in S$.

Π will consist of the following rules:

$$
\begin{aligned}
\{C : colored(X,C)\} \subseteq \{C : color(C)\} &\leftarrow node(X).\\
&\leftarrow |\{C\ :\ colored(X,C)\}| = N,\\
&\ N \neq 1.\\
&\leftarrow colored(X,C),\\
&\ colored(Y,C),\\
&\ edge(X,Y).
\end{aligned}
$$

The first rule allows the selection of arbitrary sets of colors for a given node X. The second limits the selection to one color per node. The third eliminates the selections which color neighbors by the same color. The selections left after this pruning correspond to acceptable colorings.

The next example illustrates the use of s-atoms in the body of rules.

Example 7. (Checking the course prerequisites)
Suppose that we have a record of courses passed by a student, s, given by a collection of atoms

$$passed(s,c1).\quad passed(s,c2).\quad passed(s,c3).$$

and a list of prerequisites for each class

$$prereq(c1,c4).\quad prereq(c2,c4).\quad prereq(c4,c5).$$

Our goal is to express the following rule: *A student S is allowed to take class C if he passed all the prerequisites for C and didn't pass C yet.* This rule can be written as

$$
\begin{aligned}
(a)\quad can\_take(S,C) \leftarrow\ & \{X : prereq(X,C)\} \subseteq \{X : passed(S,X)\},\\
& not\ passed(S,C).
\end{aligned}
$$

It is easy to check that the stable model M of this program, Π, consists of the above facts and an atom $can\_take(s,c4)$. Indeed, after grounding the above rule will turn into rules:

$$
\begin{aligned}
can\_take(s,c_i) \leftarrow\ & \{X : prereq(X,c_i)\} \subseteq \{X : passed(s,X)\},\\
& not\ passed(s,c_i).
\end{aligned}
$$

where $0 \leq i \leq 5$. (We are of course assuming that the variables are properly typed). The s-literals in the bodies of the rules are satisfied for i = 1,2,3, and 4 and are not satisfied for i=5. So $se(\Pi,M)$ consists of the facts and rules

$$can\_take(s,c_i) \leftarrow not\ passed(s,c_i).$$

where i = 1..4. It is easy to check that M is the only stable model of this program.

A careful reader probably noticed that the same example could be formalized in A-Prolog without the use of s-atoms. This can be done by introducing a new predicate symbol $not\_ready(S, C)$ read as "a student S is not yet ready to take a class C" and by replacing rule (a) above by the following two rules:

$$\begin{aligned}(b)\quad not\_ready(S,C) &\leftarrow prereq(X,C)\\ &\quad\; not\; passed(S,X).\end{aligned}$$

$$\begin{aligned}(c)\quad can\_take(S,C) &\leftarrow not\; not\_ready(S,C)\\ &\quad\; not\; passed(S,C).\end{aligned}$$

The following proposition shows that this is not an accident. First we need some notation. Let Π be a logic program over signature σ containing a rule

$$l_0 \leftarrow \Gamma_1, \{X \;:\; p(X)\} \subseteq \{X \;:\; q(X)\}, \Gamma_2 \tag{15}$$

By Π^* we denote the program obtained from Π by replacing rule 15 by the rules

$$d \leftarrow p(X), not\; q(X). \tag{16}$$

$$l_0 \leftarrow \Gamma_1, not\; d, \Gamma_2 \tag{17}$$

where d is formed with a predicate symbol not belonging to σ.

Proposition 4.
1. For any stable model S of Π there is a stable model S^* of Π^* such that $S = S^* \cap lit(\sigma)$.
2. If S^* is a stable model of Π^* then $S = S^* \cap lit(\sigma)$ is a stable model of Π.

This proposition shows that allowing s-atoms in the bodies of rules does not add to the expressive power of A-Prolog. It allows however a more compact representation with fewer predicate symbols. To some extent the above proposition can help to explain why stable models of programs of A-Prolog with s-terms in the bodies of their rules do not have the anti-chain property enjoyed by stable models of "pure" A-Prolog. The following example demonstrates that this is indeed the case.

Example 8. Consider the following program Π

$$p(a).$$

$$q(a) \leftarrow \{X \;:\; p(X)\} \subseteq \{X \;:\; q(X)\}.$$

which has two models, $S_1 = \{p(a)\}$ and $S_2 = \{p(a), q(a)\}$. (Since $S_1 \subset S_2$ the set of models of Π does not form an anti-chain.) It is easy to check however that the models of Π^* are $\{p(a), d\}$ and $\{p(a), q(a)\}$ which, thanks to the presence of a new atom d, do form an anti-chain.

As mentioned before selection rules of ASET-Prolog are closely related to choice rules of [66, 76] which have a form

$$m\{p(\overline{X}) \;:\; q(\overline{X})\}n \leftarrow \Gamma \tag{18}$$

Even though the general semantics of choice rules is somewhat complicated, sometimes such rules can be viewed as a shorthand for several rules of ASET-Prolog. More precisely, let us consider a program Π containing rule (18) and assume that no other rule of Π contains p in the head. Let Π^{++} be a program obtained from Π by replacing rule (18) by rules:

$$\begin{array}{l} \{X : p(X)\} \subseteq \{X : q(X)\} \leftarrow \quad \Gamma \\ \leftarrow n < |\{X : p(X)\}|. \\ \leftarrow |\{X : p(X)\}| < m \end{array}$$

Proposition 5. Let Π and Π^{++} be as above. Then S is a stable model of Π in the sense of [66] iff S is a stable model of ASET-Prolog program Π^{++}.

(Proofs of both propositions will appear in the forthcoming paper on ASET-Prolog.)

6 Reasoning in Dynamic Domains

Let us now consider domains containing agents capable of performing actions and reasoning about their effects. Such domains are often called *dynamic domains* or *dynamic systems.* We will base their description on the formalism of *action languages* [35], which can be thought of as formal models of the part of the natural language that are used for describing the behavior of dynamic domains. A theory in an action language normally consists of an action description and a history description [9], [51]. The former contains the knowledge about effects of actions, the latter consists of observations of an agent. Some discussion of architecture of autonomous agents build on action languages and A-Prolog can be found in [11]

6.1 Specifying Effects of Actions

An *action description language* contains propositions which describe the effects of actions on states of the system modeled by sets of *fluents* – statements whose truth depends on time. Fluent f is true in a state σ iff $f \in \sigma$. Mathematically, an *action description* – a collection of statements in an action description language - defines a *transition system* with nodes corresponding to possible states and arcs labeled by actions from the given domain. An arc (σ_1, a, σ_2) indicates that execution of an action a in state σ_1 may result in the domain moving to the state σ_2. We call an action description *deterministic* if for any state σ_1 and action a there is at most one such successor state σ_2. By a *path* of a transition system T we mean a sequence $\sigma_0, a^1, \sigma_1, \dots, a^n, \sigma_n$ such that for any $1 \leq i < n$, $(\sigma_i, a^{i+1}, \sigma_{i+1})$ is an arc of T; σ_0 and σ_n are called initial and final states of the path respectively. Due to the size of the diagram, the problem of finding its concise specification is not trivial and has been a subject of research for some time. Its solution requires the good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents. An additional level of complexity is added by the need to specify what is not changed by actions. The latter, known as the *frame problem*, is often reduced to the problem of finding a concise and accurate representation of the inertia axiom – a default which says that *things normally stay as they are* [60]. The search for such a representation substantially influenced AI research during the last twenty years. An interesting account of history of this research together with some possible solutions can be found in [74]. In this paper we limit our attention to an action description language $\mathcal{B}$ [35] which signature Σ consist of two disjoint, non-empty sets of symbols: the set $\mathbf{F}$ of fluents and the set $\mathbf{A}$ of *elementary actions.* A set $\{a_1, \ldots, a_n\}$ of elementary actions is called a *compound* action. It is interpreted as a collection of elementary actions performed simultaneously. By actions we mean both elementary and compound actions. By *fluent literals* we mean fluents and their negations. By $\bar{l}$ we denote the fluent literal complementary to l. A set S of fluent literals is called *complete* if, for any $f \in \mathbf{F}$, $f \in S$ or $\neg f \in S$. An action description of $\mathcal{B}(\Sigma)$ is a collection of propositions of the form

1. $causes(a_e, l_0, [l_1, \ldots, l_n])$,
2. $caused(l_0, [l_1, \ldots, l_n])$, and
3. $impossible\_if(a, [l_1, \ldots, l_n])$

where a_e and a are elementary and arbitrary actions respectively and $l_0, \ldots, l_n$ are fluent literals from Σ. The first proposition says that, if the elementary action a_e were to be executed in a situation in which $l_1, \ldots, l_n$ hold, the fluent literal l_0 will be caused to hold in the resulting situation. Such propositions are called *dynamic causal laws.* (The restriction on a_e being elementary is not essential and can be lifted. We require it to simplify the presentation). The second proposition, called a *static causal law*, says that, in an arbitrary situation, the truth of fluent literals, $l_1, \ldots, l_n$ is sufficient to cause the truth of l_0. The last proposition says that action a cannot be performed in any situation in which $l_1, \ldots, l_n$ hold. Notice that here a can be compound, e.g. $impossible\_if(\{a_1, a_2\}, [\ ])$ means that elementary actions a_1 and a_2 cannot be performed concurrently. To define the transition diagram, T, given by an action description $\mathcal{A}$ of $\mathcal{B}$ we use the following terminology and notation. A set S of fluent literals is closed under a set Z of static causal laws if S includes the head, l_0, of every static causal law such that $\{l_1, \ldots, l_n\} \subseteq S$. The set $Cn_Z(S)$ of *consequences* of S under Z is the smallest set of fluent literals that contains S and is closed under Z. $E(a_e, \sigma)$ stands for the set of all fluent literals l_0 for which there is a dynamic causal law $causes(a_e, l_0, [l_1, \ldots, l_n])$ in $\mathcal{A}$ such that $[l_1, \ldots, l_n] \subseteq \sigma$. $E(a, \sigma) = \bigcup_{a_e \in a} E(a_e, \sigma)$. The transition system $T = \langle \mathcal{S}, \mathcal{R} \rangle$ *described* by an action description $\mathcal{A}$ is defined as follows:

1. $\mathcal{S}$ is the collection of all complete and consistent sets of fluent literals of Σ closed under the static laws of $\mathcal{A}$,
2. $\mathcal{R}$ is the set of all triples (σ, a, σ') such that $\mathcal{A}$ does not contain a proposition of the form $impossible\_if(a, [l_1, \ldots, l_n])$ such that $[l_1, \ldots, l_n] \subseteq \sigma$ and

$$\sigma' = Cn_Z(E(a, \sigma) \cup (\sigma \cap \sigma')) \tag{19}$$

where Z is the set of all static causal laws of $\mathcal{A}$. The argument of $Cn(Z)$ in (19) is the union of the set $E(a, \sigma)$ of the "direct effects" of a with the set $\sigma \cap \sigma'$ of facts that are "preserved by inertia". The application of $Cn(Z)$ adds the "indirect effects" to this union.

The above definition is from [57] and is the product of a long investigation of the nature of causality. (See for instance, [49, 81].) The following theorem (a version of the result from [83]) shows the remarkable relationship between causality and beliefs of rational agents as captured by the notion of answer sets of logic programs. First we need some terminology. We start by describing an encoding τ of causal laws of $\mathcal{B}$ into a program of A-Prolog suitable for execution by *smodels*:

1. $\tau(causes(a, l_0, [l_1 \dots l_n]))$ is the collection of atoms $d\_law(d)$, $head(d, l_0)$, $action(d, a)$ and $prec(d, i, l_i)$ for $1 \leq i \leq n$, and $prec(d, m+1, nil)$ (where d is a new term used to name the corresponding law.)
2. $\tau(caused(l_0, [l_1 \dots l_n]))$ is the collection of atoms $s\_law(d)$, $head(d, l_0)$, $prec(d, i, l_i)$ for $1 \leq i \leq n$, and $prec(d, m+1, nil)$.
3. $\tau(impossible\_if([a_1, \dots, a_k], [l_1 \dots l_n]))$ is a constraint

$$\leftarrow h(l_1, T), \dots, h(l_n, T), occurs(a_1, T), \dots, occurs(a_k, T).$$

Here T ranges over integers, $occurs(a, t)$ says that action a occurred at moment t, and $h(l, t)$ means that fluent literal l holds at t. Finally, for any action description $\mathcal{A}$

$$\tau(\mathcal{A}) = \{\tau(law) : law \in \mathcal{A}\} \tag{20}$$

$$\phi(\mathcal{A}) = \tau(\mathcal{A}) \cup \Pi(1) \tag{21}$$

where $\Pi(1)$ is an instance of the following program

$$\Pi(N) \begin{cases} 1.\ h(L, T') & \leftarrow d\_law(D), \\ & \quad head(D, L), \\ & \quad action(D, A), \\ & \quad occurs(A, T), \\ & \quad prec\_h(D, T). \\ 2.\ h(L, T) & \leftarrow s\_law(D), \\ & \quad head(D, L), \\ & \quad prec\_h(D, T). \\ 3.\ all\_h(D, K, T) & \leftarrow prec(D, K.nil). \\ 4.\ all\_h(D, K, T) & \leftarrow prec(D, K, P), \\ & \quad h(P, T), \\ & \quad all\_h(D, K' \\ 5.\ prec\_h(D, T) & \leftarrow all\_h(D, 1, T). \\ 6.\ h(L, T') & \leftarrow h(L, T), \\ & \quad not\ h(\overline{L}, T'). \end{cases}$$

Here D, A, L are variables for the names of laws, actions, and fluent literals respectively, T, T' are consecutive time points from interval $[0, N]$ and K, K' stand for consecutive integers. The first two rules describe the meaning of dynamic and static causal laws, rules (3), (4), (5) define what it means for preconditions of law D to succeed, and rule (6) represents the inertia axiom from [60].

Theorem 4. For any action a and any state σ, a state σ' is a successor state of a on σ iff there is an answer set S of

$$\phi(\mathcal{A}) \cup \{h(l,0) \ : \ l \in \sigma\} \cup \{occurs\_at(a_i,0) \ : \ a_i \in a\}$$

such that, $\sigma' = \{l \ : \ h(l,1) \in S\}$.

The theorem establishes a close relationship between the notion of causality and the notion of rational beliefs of an agent. The systematic study of the relationship between entailment in action theories and in A-Prolog started in [32], where the authors formulated the problem and obtained some preliminary results. For more advanced result see [83].

6.2 Planning in A-Prolog

Now we will show how the above theory can be applied to classical planning problems of AI [80], [24], [53],[58]. Let us consider for instance the *blocks world* problem which can be found in most introductory AI textbooks:

The domain consists of a set of cubic blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can only pick up one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks.

To build a formal representation of the domain let us introduce names $b_1, \dots, b_n$ for blocks and t for the table. We use a fluent $on(B, L)$ to indicate that block B is on location L and an action $move(B, L)$ which moves block B to a position L. The corresponding types will be described as follows:

$$\begin{array}{rl}
block(b_1). \quad \dots & block(b_n). \\
loc(t). & \\
loc(L) \leftarrow & block(L). \\
fluent(on(B,L)) \leftarrow & block(B), \\
 & loc(L). \\
act(move(B,L)) \leftarrow & block(B), \\
 & loc(L).
\end{array}$$

The executability conditions for action *move* are defined by the rules:

$$\begin{array}{ll} impossible\_if(move(B,L),[on(A,B)]) & \leftarrow block(B), \\ & \quad block(A), \\ & \quad loc(L). \\ impossible\_if(move(B_1,B_2),[on(A,B_2)]) & \leftarrow block(B_1), \\ & \quad block(B_2), \\ & \quad block(A). \\ impossible\_if(move(B,B),[\,]). & \end{array}$$

The action's direct effect is represented by a dynamic causal law,

$$\begin{array}{ll} causes(move(B,L),on(B,L),[\,]) & \leftarrow block(B), \\ & \quad loc(L). \end{array}$$

The static causal law,

$$\begin{array}{ll} caused(\neg on(B,L_2),[on(B,L_1)]) & \leftarrow block(B), \\ & \quad loc(L_1), \\ & \quad loc(L_2). \\ & \quad L_1 \neq L_2 \end{array}$$

guarantees the uniqueness of a block's location. It is easy to see that the resulting program has a unique answer set, S. Atoms from S, formed by predicates *impossible_if*, *causes*, and *caused*, form an action description, $\mathcal{A}_b$, of action language $\mathcal{B}$ which defines the transition diagram, T, of the blocks wold domain.

For simplicity we restrict ourselves to a planning problem of the following type:

Given an initial node, σ_0, of the diagram, a non-negative integer n, and a collection Σ_f of goal nodes find a path of length less than or equal to n from σ_0 to one of the elements of Σ_f. The path determines the agent's *plan* – a sequence of actions it needs to perform in order to achieve its goal. We will refer to such plans as solutions of the planning problem.

We assume that an initial state, σ_0, is defined by a collection I of formulas of the form $h(l,0)$, and that the goal is given by a collection, G, of statements $g(l_0),\dots,g(l_k)$ which specifies what fluent literals must be true in a goal state. For instance, I may be

$h(on(a,t),0)$. $h(on(b,a),0)$. $h(on(c,t),0)$.

and the goal may be

$g(on(b,t))$. $g(on(a,b))$.

A planning problem defined in this way will be denoted by $(\mathcal{A},I,G,n)$.

The actual planning is done with the help of a program we call a *planning module.* The simplest planning module, PM_0, consists of the *goal constraints* and the *possible plans generator*. Constraints may be defined as follows:

$$\begin{array}{ll} fails(T) & \leftarrow 0 \leq T \leq n, \\ & \quad g(F), \\ & \quad not\ h(F,T) \end{array}$$

$$\begin{array}{ll} succeeds(T) & \leftarrow 0 \leq T \leq n \\ & \quad not\ fails(T). \\ succeeds & \leftarrow succeeds(T). \end{array}$$

$\leftarrow not\ succeeds.$

$fails(T)$ holds if at least one of the fluent literals from the goal does not hold at time T; *succeeds* holds when all the goals are satisfied at some moment of time $0 \leq T \leq n$. The last constraint requires the goal to be achievable in at most n steps.

The generator of the planning module may consists of the rules

$0\{occurs(A,T) : act(A)\}1 \leftarrow 0 \leq T < n.$

$act\_occur(T) \leftarrow occurs(A,T).$

$$\begin{array}{l} \leftarrow succeeds(T), \\ \quad 0 \leq T_1 < T \\ \quad not\ act\_occurs(T). \\ \leftarrow succeeds(T), \\ \quad T < T_1 \leq n \\ \quad act\_occurs(T). \end{array}$$

We use the choice rule of [66] to guarantee that every answer set of a program containing the planning module will contain at most one occurrence of the statement $occurs(a,t)$ for every moment $0 \leq t < n$. The two constraints guarantee that at least one action occurs at any moment of time before the goal is achieved and that no actions occur afterwards. (As mentioned in section 5, the choice rule above can be viewed as a shorthand for a selection rule of ASET-Prolog or for a collection of rules of A-Prolog.) We will also need the rules

$$\begin{array}{ll} h(F,0) & \leftarrow not\ h(\neg F,0). \\ h(\neg F,0) & \leftarrow not\ h(F,0). \end{array}$$

sometimes called the *awareness axioms*, which say that for every fluent f, either f or $\neg f$ should be included in the beliefs of a reasoning agent. (If the agent's information about the initial situation is complete this axiom can be omitted).

Let $P = (\mathcal{A}, I, G, n)$ be a planning problem and consider

$$Plan(P) = \tau(\mathcal{A}) \cup I \cup G \cup PM_0 \tag{22}$$

Using theorem 4 it is not difficult to show that

Proposition 6. A sequence $a_1, \ldots, a_k$ $(0 \leq k \leq n)$ is a solution of a planning problem P with a deterministic action description iff there is an answer set S of $Plan(P)$ such that

1. for any $0 < i \leq k$, $occurs(a_i, i-1) \in S$;
2. S contains no other atoms formed by $occurs$.

The proposition reduces the process of finding solutions of planning problems to that of finding answer sets of programs of A-Prolog. To apply it to our blocks world we need to show that the corresponding action description, $\mathcal{A}_b$, is deterministic. This immediately follows from the following proposition:

Proposition 7. If every static causal law of an action description $\mathcal{A}$ of $\mathcal{B}$ has at most one precondition then $\mathcal{A}$ is deterministic.

A-Prolog's inference engines like *smodels*, *dlv*, and *ccalc* are sufficiently powerful to make this method work for comparatively large applications. For instance in [7] the authors used this method for the development of a decision support system to be used by the flight controllers of the space shuttle. One of the system's goals was to find the emergency plans for performing various shuttle maneuvers in the presence of multiple failures of the system's equipment. Efficiency wise, performance of the planner was more than satisfactory (in most cases the plans were found in a matter of seconds.) This is especially encouraging since performance of A-Prolog satisfiability solvers is improving at a very high rate.

The system is implemented on top of *smodels*. It includes a knowledge base containing information about the relevant parts of the shuttle and its maneuvers, and of the actions available to the controllers. The effects of actions are given in an action description language $\mathcal{B}$. The corresponding action description contains a large number of static causal laws. It is interesting to notice that these laws, which are not available in more traditional planning languages like [71], played a very important role in the system design. We are not sure that a concise, elaboration tolerant, and clear description of the effects of controller's actions could be achieved without their use.

The system's planning module is based on the same generate and test idea as PM_0 but contains a number of constraints prohibiting certain combinations of actions. Constraints of this sort substantially improve the quality of plans as well as the efficiency of the planner [37].

To illustrate the idea let us show how this type of heuristic information can be used in blocks world planning. The rules, R, below express "do not destroy a good tower" heuristic suggested in [6]. It ensures that the moves of blocks which satisfy the planner's goal are immediately cut from its search space.

$$\begin{array}{ll} h(ok(t),T) & \leftarrow 0 \leq T \leq n. \\ h(ok(B_1),T) & \leftarrow 0 \leq T \leq n, \\ & \quad g(on(B_1,B_2)), \\ & \quad h(on(B_1,B_2),T), \\ & \quad h(ok(B_2),T). \end{array}$$

$$\begin{array}{l} \leftarrow occurs(move(B,L),T), \\ \quad h(ok(B),T). \end{array}$$

The first two rules define a fluent $ok(B)$ which holds at moment T if all the blocks of the tower with the top B have positions specified by the planner's goal. The last rule prohibits movements of the ok blocks. The planning module PM_1, consisting of PM_0 combined with the R rules above, returns better plans than PM_0 and is substantially more efficient. In a sense R-rules can be viewed as a declarative specification of control information limiting the search space of the planner. (It is interesting to note that in [6] a similar effect was achieved by expanding the original action description language with a variant of temporal logic. The use of A-Prolog makes this unnecessary).

7 Acknowledgments

There are many people who directly or indirectly contributed to this publication by creating its subject matter and influencing its author, and I would like to thank all of them. Special thanks however are due to Bob Kowalski. If it were not for his paper [42] I (almost accidently) read in the early eighties I probably would not get interested in logic programming and would have never had a chance to become familiar with this wonderful field of research. For that, and for the opportunity to learn from Bob's work during the last twenty years, I am very grateful.

References

1. J. Alferes and L. Pereira. *Reasoning With Logic Programming.* Springer Verlag, 1996.
2. K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA., 1988.
3. K. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3,4):335–365, 1991.
4. K. Apt and A. Pellegrini. On the occur-check free logic programs. *ACM Transaction on Programming Languages and Systems*, 16(3):687–726, 1994.
5. C. Aravindan, J. Dix, and I. Niemela. Dislop: A research project on disjunctive logic programming, *AI communications*, 10 (3/4):151-165.
6. F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. *Annals of Mathematics and Artificial Intelligence*, 22:1-2, 5-27.
7. M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R. Watson An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science - Proceedings of Practical Aspects of Declarative Languages'01*, (2001), 1990:169–183
8. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.
9. C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming*, 31(1-3):201–243, May 1997.
10. C. Baral, M. Gelfond, and O. Kosheleva. Expanding queries to incomplete databases by interpolating general logic programs. *Journal of Logic Programming*, vol. 35, pp 195-230, 1998.

11. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J Minker, editor, *Logic Based AI.* pp. 257–279, Kluwer, 2000.
12. A. Bondarenko, P.M. Dung, R. Kowalski, F. Toni, An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence* 93(1-2) pages 63-101, 1997.
13. M. Cadoli, T. Eiter, and G. Gottlob. Default logic as a query language. IEEE Transactions on Knowledge and Data Engineering, 9(3), pages 448–463, 1997.
14. W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.
15. P. Cholewinski. Stratified Default Logic. *In Computer Science Logic*, Springer LNCS 933, pages 456–470, 1995.
16. P. Cholewinski. Reasoning with Stratified Default Theories. *In Proc. of 3rd Int'l Conf. on Logic Programming and Nonmonotonic Reasoning*, pages 273–286, 1995.
17. P. Cholewinski, W. Marek, and M. Truszczyński. Default Reasoning System DeReS. In *Int'l Conf. on Principles of Knowledge Representation and Reasoning*, 518-528. Morgan Kauffman, 1996.
18. A. Colmerauer, H. Kanoui, R. Pasero, and P. Russel. Un systeme de communication homme-machine en francais. Technical report, Groupe de Intelligence Artificielle Universitae de Aix-Marseille, 1973.
19. T. Eiter and G. Gottlob. Complexity aspects of various semantics for disjunctive databases. In Proc. of PODS-93, pages 158-167, 1993.
20. T. Eiter, N. Leone, C. Mateis., G. Pfeifer and F. Scarcello. A deductive system for nonmonotonic reasoning, Procs of the LPNMR'97, 363–373, 1997
21. T. Eiter, W. Faber, N. Leone. Declarative problem solving in DLV. In J Minker, editor, *Logic Based AI*, 79–103 Kluwer, 2000.
22. K. Clark. Negation as failure. In Herve Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.
23. J. Delgrande and T. Schaub. Compiling reasoning with and about preferences into default logic. In *Proc. of IJCAI 97*, 168–174, 1997.
24. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proc. of the 4th European Conference on Planning, ECP'97*, 1348:169–181, 1997
25. E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. *Proc. of CL-2000*, 822-836, 2000.
26. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.
27. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
28. M. Gelfond and A. Gabaldon. Building a knowledge base: an example. *Annals of mathematics and artificial Intelligence*, 25:165–199.
29. M. Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.
30. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.
31. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387, 1991.
32. M. Gelfond and V. Lifschitz. Representing Actions and Change by Logic Programs. *Journal of Logic Programming*, 17:301–323.

33. M. Gelfond, V Lifschitz, H. Przymusinska, and M. Truszczynski. Disjunctive defaults. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second Int'l Conf.*, pages 230–237, 1991.
34. M. Gelfond, H. Przymusinska. On Consistency and Completeness of Autoepistemic Theories, Fundamenta Informaticae, vol. 16, Num. 1, pp. 59-92, 1992.
35. M. Gelfond and V. Lifschitz. Action Languages. *Electronic Transactions on Artificial Intelligence*, Vol. 2, 193-210, 1998 http://www.ep.liu.se/ej/etai/1998/007
36. M. Gelfond and T. Son. Reasoning with prioritized defaults. In J. Dix, L. M. Pereira, T. Przymusinski, editors, *Lecture Notes in Artificial Intelligence*, 1471, pp 164-224, 1998.
37. Y. Huang, H. Kautz and B. Selman. Control Knowledge in Planning: Benefits and Tradeoffs. *16th National Conference of Artificial Intelligence (AAAI'99)*, 511–517.
38. K. Inoue and C. Sakama. Negation as Failure in the Head. *Journal of Logic Programming*, 35(1):39-78, 1998.
39. C. Sakama and K. Inoue Prioritized Logic Programming and its Application to Commonsense Reasoning, *Artificial Intelligence* 123(1-2):185-222, Elsevier, 2000.
40. A. C. Kakas, R. Kowalski, F. Toni, The Role of Abduction in Logic Programming, *Handbook of Logic in Artificial Intelligence and Logic Programming 5*, pages 235-324, D.M. Gabbay, C.J. Hogger and J.A. Robinson eds., Oxford University Press (1998).
41. M. Kaminski. A note on the stable model semantics of logic programs. *Artificial Intelligence*, 96(2):467–479, 1997.
42. R. Kowalski. Predicate logic as a programming language. *Information Processing 74*, pages 569–574, 1974.
43. R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.
44. C. Koch and N. Leone Stable model checking made easy. In proc. of IJCAI'99, 1999.
45. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):289–308, 1987.
46. K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231–245, 1989.
47. V. Lifschitz. Restricted Monotonicity. *In proc. of AAA-93*, pages 432–437, 1993
48. V. Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.
49. V. Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.
50. V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.
51. V. Lifschitz, Two components of an action language. *Annals of Math and AI*, 21(2-4):305–320, 1997.
52. V. Lifschitz and L. Tang and H. Turner, Nested expressions in logic programs. *Annals of Math and AI*, Vol. 25, pages 369-389, 1999
53. V. Lifschitz. Action languages, Answer Sets, and Planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, 357–353, Spring-Verlag, 1999.
54. W. Marek and V.S. Subrahmanian. The relationship between logic program semantics and non–monotonic reasoning. In G. Levi and M. Martelli, editors, *Proc. of the Sixth Int'l Conf. on Logic Programming*, pages 600–617, 1989.
55. W. Marek, and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38, pages 588–619, 1991.

56. W. Marek, and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375–398, Spring-Verlag. 1999.
57. N. McCain and H. Turner. Causal theories of action and change. In *Proc. of AAAI*, pages 460–465, 1997.
58. N. McCain and H. Turner. Satisfiability planning with causal theories. In *Proc. of KR*, pages 212–223, 1998.
59. J. McCarthy. Programs with common sense. In *Proc. of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office.
60. J. McCarthy and P. 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.
61. J. Minker Overview of disjunctive logic programming. *Annals of mathematics and artificial Intelligence*, 12:1–24, 1994.
62. R. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1):75–94, 1985.
63. D. Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14:16–26, 1949.
64. I. Niemela. Logic Programming with stable model semantics as a constraint programming paradigm. In proceedings of the workshop on computational aspects of nonmonotonic reasoning, pp 72–79, Trento, Italy, 1998.
65. I. Niemela and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. 4th international conference on Logic programming and non-monotonic reasoning*, pages 420–429, 1997.
66. I. Niemela and P. Simons. Extending the Smodels system with cardinality and weight constraints. In J Minker, editor, *Logic Based AI*, pp. 491–522, Kluwer, 2000.
67. D. Pearce and G. Wagner. Reasoning with negative information 1 – strong negation in logic programming. Technical report, Gruppe fur Logic, Wissentheorie and Information, Freie Universitat Berlin, 1989.
68. D. Pearce. From here to there: Stable negation in logic programming. In D. Gabbay and H. Wansing, editors. *What is negation?*, Kluwer, 1999.
69. T. Przymusinski. Perfect model semantics. *In Proc. of Fifth Int'l Conf. and Symp.*, pages 1081–1096, 1988.
70. T. Przymusinski. Stable semantics for disjunctive programs. *New generation computing*, 9(3,4):401–425, 1991.
71. E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. *In Proc. of KR89*, pages 324–332, 1987.
72. R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.
73. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.
74. M. Shanahan. *Solving the frame problem: A mathematical investigation of the commonsense law of inertia.* MIT press, 1997.
75. J. Schlipf. The expressive powers of the logic programming semantics. *Journal of the Computer Systems and Science*, 51, pages 64–86, 1995.
76. Simons, P. Extending the stable model semantics with more expressive rules. In *5th International Conference, LPNMR'99*, 305–316.

77. T. Soininen and I. Niemela. Developing a declarative rule language for applications in program configuration. *In practical aspects of declarative languages*, LNCS 1551, pages 305-319, 1999.
78. D. Seipel and H. Thone. DisLog - A system for reasoning in disjunctive deductive databases. *In proc. of DAISD'94.*
79. K. Stroetman. A Completeness Result for SLDNF-Resolution. *Journal of Logic Programming*, 15:337–355, 1993.
80. V. Subrahmanian and C. Zaniolo. Relating stable models and AI planning domains. In L. Sterling, editor, *Proc. ICLP-95*, pages 233–247. MIT Press, 1995.
81. M. Thielscher. Ramification and causality. *Artificial Intelligence*, 89(1-2):317–364, 1997.
82. H. Turner. Signed logic programs. In *Proc. of the 1994 International Symposium on Logic Programming*, pages 61–75, 1994.
83. H. Turner. Representing actions in logic programs and default theories. *Journal of Logic Programming*, 31(1-3):245–298, May 1997.
84. H. Turner. Splitting a Default Theory, *In Proc. of AAAI-96*, pages 645–651, 1996.
85. M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM.*, 23(4):733–742, 1976.
86. A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.
87. G. Wagner. Logic programming with strong negation and inexact predicates. *Journal of Logic and Computation*, 1(6):835–861, 1991.