

- introduction
- A-Prolog
- problem solving modules in A-Prolog
- action languages
- answer set solvers

Action Languages: introduction

- normally, a problem solving task involves finding solutions to problems. A **declarative way** of doing this is to enumerate the possible answers specifying its properties, and then verifying that these answers are in fact solutions to the problem.
- the difference between an **intelligent agents** and the others is the possibility of generating plans beyond just following a given set of instructions

Algorithm *vs* Especification

- **classical logic** was first used in this sense, but it is not possible to have a good formalization of the frame axioms.
- **A-Prolog** is a nonmonotonic formalism with computationally tractable algorithms, in which negation-as-failure is interpreted under the semantics of stable models

- there is also the possibility in A-Prolog of representing another type of negation: **strong negation**
- Comparison Prolog vs. A-Prolog:
 - Prolog interpreters **include several non declarative operators** (like *cut*); on the other hand answer set semantics is completely declarative
 - the **order** of the literals in the body of a rule, and the order of the rules in a program are important in Prolog's evaluation
 - Prolog's evaluation is done **from the goals towards the facts**, this order is not part of the semantics of A-Prolog
 - there are other problems like **floundering** and **occurs check** that affect Prolog execution, which are not present in A-Prolog
 - in spite of all these facts, Prolog **is still a valid alternative** in producing sound answers under the answer set semantics

A-Prolog

- the **alphabet** of a program in A-Prolog consists of: variables, constants, functions symbols, predicate symbols, the connectives (\neg , *or*, \leftarrow , *not*, $\&$), punctuation symbols and \perp
- the connectives are **not the classical logic connectives** $\neg, \wedge, \vee, \Rightarrow$
- the **signature** is the part of the alphabet that varies from program to program: constant, function symbols and predicate symbols. If it is not specified, we understand that it is formed by all symbols appearing in the program
- terms, atoms, literals, ground terms, ground atoms, ground literals are defined as in classical logic
- a **naf literal** is an atom, or an atom preceded by *not*. An **extended literal** is a literal, or a literal preceded by *not*.

- a **rule** is formula of the form

$$L_0 \text{ or } \dots \text{ or } L_k \leftarrow L_{k+1} \ \& \ \dots \ \& \ L_m \ \& \ \text{not } L_{m+1} \ \& \ \dots \ \& \ \text{not } L_n$$

where L_i are literal, and L_0 can be \perp if $k = 0$.

The left hand size of \leftarrow is called **head**, and the right hand size is called **body**.

- if $k = 0$ and $L_0 = \perp$ then the rule is called **constraint**, and it is noted

$$\leftarrow L_{k+1} \ \& \ \dots \ \& \ L_m \ \& \ \text{not } L_{m+1} \ \& \ \dots \ \& \ \text{not } L_n$$

- in general, the symbol $\&$ is replaced by “,”.
- a **program in A-Prolog*** consists in a finite set of rules. The program will be associated with a language formed by the implicit signature.

- several subsets of A-Prolog* will be defined:
 - A-Prolog: rules where all L_i are atoms and $k = 0$.
 - A-Prolog^{-not}: rules where all L_i are atoms, $k = 0$ and $m = n$.
 - A-Prolog[¬]: rules where $k = 0$.
 - A-Prolog^{or}: rules where all L_i are atoms.
 - A-Prolog[⊥], A-Prolog^{-not,⊥}, A-Prolog^{¬,⊥}, A-Prolog^{or,⊥} if we allow constraint in each of the previous classes.
 - A-Datalog: subset of A-Prolog where no function symbol is allowed in the signature.

- the **semantics** for A-Prolog programs is based on the stable models for normal logic programs
- a **partial Herbrand interpretation** for a program is a set of ground literals which is consistent, or is the whole set of literals in the language
- a partial Herbrand interpretation S is a **model** of a ground rule in A-Prolog ^{\neg, \perp}

$$L_0 \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

iff the following hold

1. if $L_0 \neq \perp$, then $\{L_1, \dots, L_m\} \subseteq S$ and $\{L_{m+1}, \dots, L_n\} \subseteq \bar{S}$ imply that $L_0 \in S$.
 2. if $L_0 = \perp$, then $\{L_1, \dots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \dots, L_n\} \not\subseteq \bar{S}$.
- an **answer set** for an A-Prolog ^{$\neg, -not, \perp$} program Π is a partial Herbrand interpretation that is a **minimal model** for all the rules in the program
 - it can be shown that any ground A-Prolog ^{$\neg, -not, \perp$} program Π has a unique answer set. It will be noted $\mathcal{M}^{\neg, -not, \perp}(\Pi)$.

- let Π be a ground program in $\text{A-Prolog}^{\neg, \perp}$, and S a partial Herbrand interpretation. The **reduct** Π^S is the program obtained from Π applying
 1. eliminating all rules with *not* L in the body, for some $L \in S$
 2. eliminating all naf literals from the body of the remaining rules.
- an **answer set** for a ground program Π in $\text{A-Prolog}^{\neg, \perp}$ is a partial Herbrand interpretation S such that

$$S = \mathcal{M}^{\neg, -not, \perp}(\Pi^S)$$

- a partial Herbrand interpretation S is a **model** of a ground program Π in $\text{A-Prolog}^{\neg, or, \perp}$ if for every rule in Π of the form

$$L_0 \text{ or } \dots \text{ or } L_k \leftarrow L_{k+1}, \dots, L_m, \text{ not } L_{m+1}, \dots, \text{ not } L_n$$

the following hold

1. if $L_0 = \perp$ and $k = 0$, then $\{L_{k+1}, \dots, L_m\} \not\subseteq S$ or $\{L_{m+1}, \dots, L_n\} \not\subseteq \bar{S}$.
 2. in other case, $\{L_{k+1}, \dots, L_m\} \subseteq S$ and $\{L_{m+1}, \dots, L_n\} \subseteq \bar{S}$ imply that $\{L_0, \dots, L_k\} \cap S \neq \emptyset$.
- an **answer set** of a ground program Π in $\text{A-Prolog}^{\neg, or, -not, \perp}$ is a partial Herbrand interpretation which is minimal between the models of Π
 - it can be show that answer sets for ground programs in $\text{A-Prolog}^{\neg, or, -not, \perp}$ are unique, and will be noted $\mathcal{M}^{\neg, or, -not, \perp}(\Pi)$.

- let Π be a program in A-Prolog^{*}, and S a partial Herbrand interpretation. The **reduct** Π^S is the program obtained from Π applying the following transformations
 1. eliminating the rules with *not* L in the body, for some $L \in S$.
 2. eliminating every other naf literals from the bodies of the remaining rules.
- an **answer set** for a ground program Π in A-Prolog^{*} is a partial Herbrand interpretation such S such that

$$S = \mathcal{M}^{\neg, -not, \perp}(\Pi^S)$$

- the semantics for **non-ground programs** is defined as the semantics for the ground program obtained by instantiating each variable in the program with all possible ground terms in the language.
- a **query** to a program in de A-Prolog is defined from ground literals combined with classical connectives \vee \wedge .

Problem solving modules in A-Prolog

1. integrity constraints
2. finite enumeration
3. at least one enumeration
4. exactly one enumeration
5. propositional satisfiability

Problem solving modules in A-Prolog: 1.Integrity Constraints

- we would like to represent integrity constraints in A-Prolog subsetw without \perp
- it is possible to represent an A-Prolog $^{\perp}$ constraint of the form

$$\leftarrow p_1, \dots, p_n, \textit{not } q_1, \dots, \textit{not } q_m$$

in A-Prolog $^{-\perp}$ with the followin formula

$$\textit{inconsistent} \leftarrow \textit{not inconsistent}, p_1, \dots, p_n, \textit{not } q_1, \dots, \textit{not } q_m$$

where *inconsistent* is a new atom.

- example: :

$$a \leftarrow \text{not } b$$
$$b \leftarrow \text{not } a$$
$$\leftarrow a$$

has the same answer set as

$$a \leftarrow \text{not } b$$
$$b \leftarrow \text{not } a$$
$$p \leftarrow \text{not } p, a$$

Problem solving modules in A-Prolog: 2. Finite Enumeration

- between propositions p_1, \dots, p_n , meaning that either the propositions or their negation must belong to every answer set
- the A-Prolog ^{\neg, or} program containing the following pairs of rules

$$p_i \leftarrow \text{not } \text{no_}p_i$$
$$\text{no_}p_i \leftarrow \text{not } p_i$$

for all $1 \leq i \leq n$, being $\text{no_}p_i$ new propositions, satisfies the requirements

- this can be also represented in A-Prolog ^{\neg, or} with

$$p_i \text{ or } \neg p_i \leftarrow$$

Problem solving modules in A-Prolog: 3.At Least One Enumeration

- between instances of `possible(X)`, meaning that at least one instant of this predicate must be true in all answer sets
- the following module in A-Prolog

```
selected(X) ← possible(X), not notSelected(X)
noSelected(X) ← possible(X), not selected(X)
some ← selected(X)
inconsistent ← not inconsistent, not some
```

satisfies this requirement.

- example: the program

```
possible(a) ←  
possible(b) ←  
selected(X) ← possible(X), not notSelected(X)  
noSelected(X) ← possible(X), not selected(X)  
some ← selected(X)  
inconsistent ← not inconsistent, not some
```

has the following answer sets:

```
{possible(a), possible(b), selected(a), noSelected(b), some}  
{possible(a), possible(b), selected(b), noSelected(a), some}  
{possible(a), possible(b), selected(a), selected(b), some}
```

Problem solving modules in A-Prolog: 4. Exactly One Enumeration

- between instances of `possible(X)`, meaning that one, and only one instance of this predicate is true in all answer sets
- the following module in A-Prolog[⌈]

```
¬selected(X) ← selected(Y), X ≠ Y
selected(X) ← possible(X), not ¬selected(X)
```

satisfies the requirement

- example: the program

```
possible(a) ←  
possible(b) ←  
¬selected(X) ← selected(Y), X ≠ Y  
selected(X) ← possible(X), not ¬selected(X)
```

has the following answer sets:

```
{possible(a), possible(b), selected(a), ¬selected(b)}  
{possible(a), possible(b), selected(b), ¬selected(a)}
```

Problem solving modules in A-Prolog: 5.Propositional Satisfiability

- let $S = \{C_i\}$ be a set of propositional clauses, we would like to have an A-Prolog program such that determines if S is satisfiable
- consider the program Π_S in A-Prolog such that
 1. for each proposition p appearing in S we add new predicate symbols p, no_p in the language, and the finite enumeration rules for atom p
 2. and for each C in S we add a new predicate symbol c in the language and the following rules
 - if there is a positive literal $p \in C$

$$c \leftarrow p$$

- if there is a negative literal $\neg p \in C$

$$c \leftarrow no_p$$

3. also, we need the following constraint for each clause C in S

$$\leftarrow \text{not } c$$

then answer sets of Π_S have a **one-to-one correspondence** with models of S

Action Languages

- example: let $S = (p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee \neg p_3)$ then Π_S is the program

```
p1 ← not no_p1   no_p1 ← not p1
p2 ← not no_p2   no_p2 ← not p2
p3 ← not no_p3   no_p3 ← not p3
c1 ← p1          c1 ← no_p2
c1 ← p3
c2 ← no_p1       c2 ← no_p3
← not c1
← not c2
```

Action Languages: action language \mathcal{A}

- with the objective of formalizing reasoning about actino effects, we will introduce a **simple, high level language** with an intuitive semantics.
- then we will translate this language into A-Prolog, in order to find solutions to the specified problems
- **language \mathcal{A}** was propose by Gelfond & Lifschitz in 1992, and was extended in several ways

Sub-languages of \mathcal{A} $\left\{ \begin{array}{l} \text{domain description language} \\ \text{observations language} \\ \text{queries} \end{array} \right.$

- the **alphabet** of \mathcal{A} consists of two non-empty sets of symbols \mathbf{F} and \mathbf{A} , called **fluent symbols** and **action symbols**
- a **fluent literal** is f or $\neg f$, where $f \in \mathbf{F}$.
- a **state** σ is a set of fluents, where f is true in σ if $f \in \sigma$, and $\neg f$ is true in σ if $f \notin \sigma$.
- example: blocks world

$$\mathbf{F} = \{\text{ontable}(X), \text{on}(X, Y), \text{clear}(X), \text{holding}(X), \text{handempty}\} \forall \text{ block } X, Y$$
$$\mathbf{A} = \{\text{pickup}(X), \text{putdown}(X), \text{stack}(X, Y), \text{unstack}(X, Y)\} \forall \text{ block } X, Y$$
$$\sigma = \{\text{ontable}(a), \text{on}(a, b), \text{clear}(a), \text{handempty}\}$$

- the **semantics** of \mathcal{A} is defined in terms of situations
- a **situation** is represented as a list of actions, where the initial situation is the empty list
- $[a_n, \dots, a_2, a_1]$ is the situation resulting from the application of a_1 to the initial situation, and then applying a_2 , and so on up to applying action a_n
- different situations may correspond to the same state

- example:

situations

[]

[stack(a, b), unstack(a, b)]

[stack(a, b), unstack(a, b), stack(a, b), unstack(a, b)]

correspond to

$\sigma = \{\text{ontable}(a), \text{on}(a, b), \text{clear}(a), \text{handempty}\}$

if σ is the initial state

- the domain description language in \mathcal{A} is formed by a set D of effect declarations

a causes f if $p_1, \dots, p_n, \neg q_1, \dots, \neg q_m$

where **a** is an action, **f** is a fluent literal and p_i, q_j are fluents

- the intuitive semantics is that if $p_1, \dots, p_n, \neg q_1, \dots, \neg q_m$ hold in a state corresponding to situation s , then **f** will hold in situation $[a|s]$ resulting from applying **a** to s
- Special cases:
 - if $n = m = 0$ then

a causes f

- $\{a \text{ causes } f_1, \dots, a \text{ causes } f_n\}$ can be represented by

a causes f_1, \dots, f_n

- in case **a**, **f**, p_i, q_i have variables, the formula is considered as a schema

- example:

`pickup(X)causes¬ontable(X), ¬handempty, holding(X), ¬clear(X)`

is equivalent to

`pickup(a)causes¬ontable(a), ¬handempty, holding(a), ¬clear(a)`

`pickup(b)causes¬ontable(b), ¬handempty, holding(b), ¬clear(b)`

- the role of effect propositions is to define a [transition function](#) Φ between states and action to states.

- for all action a , fluent g and state σ
 - if a **causes** g **if** $p_i, \neg q_j$ belongs to the description and $\{p_i, \neg q_j\}$ hold in σ , then $g \in \Phi(a, \sigma)$
 - if a **causes** $\neg g$ **if** $p_i, \neg q_j$ belong to the description and $\{p_i, \neg q_j\}$ hold in σ , then $g \notin \Phi(a, \sigma)$
 - if a **causes** $\neg g$ **if** $p_i, \neg q_j$ do not belong to the description, then $g \in \Phi(a, \sigma)$ iff $g \in \sigma$.
- for each domain predicate description in D , exists at most one transition function σ . If Φ exists, then D is said to be **consistent**

- the set of **observations** O is a set by propositions of the form

f after a_1, \dots, a_n

where **f** is a fluent literal, and a_i are actions

- intuitively, the meaning of these propositions is that if a_1, \dots, a_n are executed from the initial situation then **f** holds in the state $[a_n, \dots, a_1]$

- **special cases:**

– if $m = 0$ then

initially f

– **{initially f₁, ..., initially f_n}** is represented with como

initially f₁, ..., f_n

- given a consistent domain description D , a set of observations O is used to determine the **states obtained from the initial situation**, denoted with σ_0 .
 D determines a unique transition function, but O can determine **several initial states**.
- given D and O , the pair (Φ_D, σ_0) is called a **model** of D, O . D, O is **consistent** if it has at least one model, and **complete** if it has a unique model

- example: *Yale shooting problem*
with fluens **alive**, **loaded** and actions **load**, **shoot**.
 D is represented with

load causes loaded

shoot causes \neg alive if loaded

If $O = \{\mathbf{initially\ alive}\}$ then we have two initial states

$$\sigma_0 = \{\mathbf{alive}\}$$

$$\sigma'_0 = \{\mathbf{alive, loaded}\}$$

then D, O is consistent but not complete

If $O' = \{\mathbf{initially\ alive, \neg loaded}\}$ then D, O' is consistent and complete.

- **queries** are proposition of the form

f after a_1, \dots, a_n

where **f** is a fluent literal, and a_i are actions

- we say that a domain description D in presence of observations O **denote a query** Q if for all initial states σ_0 corresponding to D, O the fluent literal **f** holds in the state $[a_m, \dots, a_1]\sigma_0$. It is noted $D \models_O Q$.

- example: let D be

load **causes** loaded

shoot **causes** \neg alive **if** loaded

- if $O = \{\text{initially alive}\}$ then query

$$Q = \{\neg\text{alive after shoot}\}$$

is such that $D \not\models_0 Q$.

- if $O' = \{\text{initially alive, loaded}\}$ then $D \models_0 Q$.

- on the other hand if

$$Q' = \{\neg\text{alive after load, shoot}\}$$

then $D \models_0 Q'$.

- in language \mathcal{A} we can express several kinds of reasoning about action

Reasoning about Actions {
Temporal Projection
Reasoning about the initial situation
Observation assimilation
Planning

- in temporal projection observations are only of the form **initially** f , and we are only interested in drawing conclusion about hypothetical future
- there are two cases, depending whether the initial state is unique.
A set of observations O is complete if for every fluent f , either **initially** f or **initially** $\neg f$ belong to O , but not both.

- in reasoning about the initial situation observations are arbitrary, but queries are only of the form **initially** f (backward reasoning)
- example let D be the *Yale shooting problem* description and

$$O_1 = \{\mathbf{initially\ alive}; \neg\mathbf{alive\ after\ shoot}\}$$

Then there is only one σ_0 , and we can say that $D \models_{O_1} \mathbf{initially\ loaded}$

- in *observation assimilation* we generalize both previous cases. Observations and queries are arbitrary.
- example: let D be the description of the *Yale Shooting problem* and

$$O_2 = \{\mathbf{initially\ alive, loaded\ after\ shoot}\}$$

Also there is only one σ_0 for D, O_2 , and we can see that $D \models_{O_2} \neg\mathbf{alive\ after\ shoot}$

- **Planning**: we have a domain description D , a set of observations about the initial situation O , and a set of fluent literals G called **goal**.

We require to find a sequence of actions a_1, \dots, a_n such that for each $g \in G$ the following holds

$$D \models_O g \text{ after } a_1, \dots, a_n$$

- the sequence a_1, \dots, a_n is called a **plan**.
- example: let D be the description of the *Yale Shooting problema* and

$$O = \{\text{initially alive}\}$$

$$O' = \{\text{initially alive; initially loaded}\}$$

Then for $G = \{\neg \text{alive}\}$ we have that **shoot** is a plan for D, O', G but not for D, O, G .
On the other hand, **load, shoot** is a plan for D, O, G .

Translation to A-Prolog*

- Translation of temporal projection with a complete initial state
 1. translate domain description
 2. translate observations
 3. include inertia rule

- given D, O we create the program $\Pi_1(D, O)$ with the following rules
 - for each effect proposition in D of the form

a causes f if $p_1, \dots, p_n, \neg q_1, \dots, q_m$

where f is a fluent we have in the subprogram $\Pi_1^{ef}(D)$ a rule of the form

$$\text{holds}(f, \text{res}(a, S)) \leftarrow \text{holds}(p_1, S), \dots, \text{holds}(p_n, S), \\ \text{not holds}(q_1, S), \dots, \text{not holds}(q_m, S)$$

and if $f = \neg g$ is a negative fluent literal we include the rule

$$\text{ab}(g, a, S) \leftarrow \text{holds}(p_1, S), \dots, \text{holds}(p_n, S), \\ \text{not holds}(q_1, S), \dots, \text{not holds}(q_m, S)$$

- for each proposition in O of the form

initially f

where f is a fluent we include in the subprogram $\Pi_1^{obs}(O)$ the rule

$$\text{holds}(f, s_0) \leftarrow$$

and if f is a negative fluent literal we **do not add anything** to the program

- in order to include the inertial rules we add the subprogram Π_1^{in} with the following rules

$$\text{holds}(F, \text{res}(A, S)) \leftarrow \text{holds}(F, S), \text{not ab}(F, A, S)$$

- example: let D be the description of the *Yale Shooting problem* and

$$O_3 = \{\text{initially alive; initially } \neg\text{loaded}\}$$

Then Π_1 equivalent to D, O_3 is the program

```
holds(loaded, res(load, S)) ←  
  ab(alive, shoot, S) ← holds(loaded, S)  
  holds(alive, s0) ←  
  holds(F, res(A, S)) ← holds(F, S), not ab(F, A, S)
```

- let $D, =$ be a domain description and observations from which we want to do **temporal projection with complete initial state**
- then the translation into A-Prolog results in a program Π_1 with the following properties:
 - Π_1 has only one answer set M
 - let (σ_0, Φ) be the only model of D, O then

$$f \in \Phi(a_n, \dots, \Phi(a_1, \sigma_0) \dots) \text{ iff } \text{holds}(f, [a_n, \dots, a_1]) \in M$$

- we can obtain a more elegant translation, with symmetric definition, using A-Prolog⁻.
- the resulting program $\Pi_2(D, O)$ has the same properties as in the previous case
- in similar way we can obtain translation for temporal projections with incomplete initial states, reasoning about the initial situation, and observations assimilation

- the previous translation uses a representation scheme based on [Situation Calculus](#)
- this representation is appropriate for [verifying correctness](#) in simple plans, but not for generating the plans
- in order to do planning, we have two possibilities:
 - rely on interpreters that are able to do [answer extraction](#), instantiating free variables in queries
 - using the [generate and test](#) strategy where possible plans are systematically generated and tested in order to see if they satisfy the given properties
- an alternative representation is using lineal and discrete time, reasoning about the value of fluents in each time point. This is the representation used in [event calculus](#)

- in this case the process of planning can be done **enumerating** in each answer set the different action sequence, and **rejecting by means of constraints** those that do not satisfy the requirements
- this method is called **Answer Set Planning**, and is similar to SATPLAN
- we will do this translation for temporal projection, and then use the result in a translation for planning

- let D, O be a domain description and a set of observations, with an complete initial state description Then the program $\Pi_{2.ev}$ is formed by the following subprograms:
 - **translation of effect propositions** the subprogram $\Pi_{2.ev}^{ef}(D)$ contains for each proposition of the form

a causes f if $p_1, \dots, p_n, \neg q_1, \dots, \neg q_m$

where f is a fluent, the rules:

$$\text{holds}(f, T + 1) \leftarrow \text{occurs}(a, T), \text{holds}(p_1, T), \dots, \text{holds}(p_n, T), \\ \text{notHolds}(q_1, T), \dots, \text{notHolds}(q_m, T)$$
$$\text{ab}(f, a, T) \leftarrow \text{occurs}(a, T), \text{holds}(p_1, T), \dots, \text{holds}(p_n, T), \\ \text{notHolds}(q_1, T), \dots, \text{notHolds}(q_m, T)$$

and if $f = \neg g$ is a negative fluent literal then

$$\text{notHolds}(g, T + 1) \leftarrow \text{occurs}(a, T), \text{holds}(p_1, T), \dots, \text{holds}(p_n, T), \\ \text{notHolds}(q_1, T), \dots, \text{notHolds}(q_m, T)$$

$$\text{ab}(g, a, T) \leftarrow \text{occurs}(a, T), \text{holds}(p_1, T), \dots, \text{holds}(p_n, T), \\ \text{notHolds}(q_1, T), \dots, \text{notHolds}(q_m, T)$$

- **translation of observation** the subprogram $\Pi_{2.ev}^{obs}(O)$ contains for each proposition of the form

initially f

where f is a fluent the rule

$$\text{holds}(f, 1) \leftarrow$$

and if $f = \neg g$ is a negative fluent literal then the rule

$$\text{notHolds}(g, 1) \leftarrow$$

- **inertia rules** the subprogram $\Pi_{2.ev}^{in}$ consists of the rules

$$\text{holds}(F, T + 1) \leftarrow \text{occurs}(A, T), \text{holds}(F, T), \text{not ab}(F, A, T)$$
$$\text{notHolds}(F, T + 1) \leftarrow \text{occurs}(A, T), \text{notHolds}(F, T), \text{not ab}(F, A, T)$$

- the program $\Pi_{2.ev}$ has the following **soundness and completeness** property for consistent domain description D , O and fluents f :
 - $D \models_O f$ **after** a_1, \dots, a_n iff $\pi_{2.ev} \cup \{\text{ocurrs}(a_i, i)\}_{1 \leq i \leq n} \models \text{holds}(f, n + 1)$
 - $D \models_O \neg f$ **after** a_1, \dots, a_n iff $\pi_{2.ev} \cup \{\text{ocurrs}(a_i, i)\}_{1 \leq i \leq n} \models \text{notHolds}(f, n + 1)$

- the program $\Pi_{2.ev}(D, O)$ can be easily extended to do planning, by means of enumerating the sequence of actions, and eliminating the incompatible answer sets
- the program $\Pi_{2.pl}(D, O)$ which finds a plan of length l is the program resulting from the union of $\Pi_{2.ev}(D, O)$ and
 - **choice rules** the subprogram $\Pi_{2.pl}^{choice}$ stating that at each time point only one action is executed

$$\text{notOccurs}(A, T) \leftarrow \text{occurs}(B, T), A \neq B$$
$$\text{occurs}(A, T) \leftarrow T \leq l, \text{not notOccurs}(A, T)$$

- **goal** the subprogram $\Pi_{2.pl}^{goal}$ which constraints the final state of the process

$$\leftarrow \text{not holds}(h, l + 1)$$

- it is possible to show an **exact correspondence** between answer sets of $\Pi_{2.pl}(D, O)$ and plans of length l in D, O
- the main problem of this representation is that **it is necessary to know exactly the plan length l before finding it.**
- a possible solution is to establish an **upper bound** on l , while adding a **do-nothing** action

Answer Set Solvers

- it is possible to ask queries about the previous programs using standard Prolog interpreters
- this method is **sound but not complete** with respect to the answer set semantics
- but in order to find answer set it is better to use specific **answer set solver**
- we will analyse **SMODELS** and **DLV**
- SMODELS home page is in <http://www.tcs.hut.fi/Software/smodels/>
- SMODELS is oriented to compute answer sets from **programs in A-Prolog[⊥] and A-Prolog^{⊥,⊃}**

- it consists in **two modules**: `lparse` y `smodels`
- module `lparse` takes as input the program, and checks whether it is **finite and without cycles**, and then **instantiates** it generating an intermediate representation
- module `smodels` takes as input this intermediate representation of the program, and generates a description of **all its answer sets**
- the input language includes several kinds of **integrity constraints restrictions**, **optimization sentences**, and a **computation sentence**

$$\text{Integrity Constraints} \begin{cases} \textit{cardinality} : N\{A_i, \textit{not} B_j\}M \\ \textit{weight} : N[A_i = W_i, \textit{not} B_j = W_j]M \end{cases}$$

- example: the program

$$\begin{aligned} &1\{a, b, c, \}2 \leftarrow p \\ & \quad p \leftarrow \end{aligned}$$

has the following answer sets: $\{a, p\}$ and $\{a, b, p\}$.

- the **computation sentence** is of the form $N\{A_i, \text{not } B_j\}$, and it works as a filter for answer sets.
- **optimization sentences** have one of the forms

$$\begin{aligned} &\text{maximize}\{A_i, \text{not } B_j\} \\ &\text{minimize}\{A_i, \text{not } B_j\} \\ &\text{maximize}[A_i = W_i, \text{not } B_j = W_j] \\ &\text{minimize}[A_i = W_i, \text{not } B_j = W_j] \end{aligned}$$

- if the system finds one or more of these optimization sentences, then the results will include only those answer sets that are **optimal** in these criteria

- **DLV** can be found at <http://www.dbai.tuwien.ac.at/proj/dlv/>
- DLV is oriented towards computing answer sets of programs in **A-Prolog***
- it has several post-processors for knowledge representation, diagnosis, planning, and other applications
- the system works executing the following steps
 1. converts the input program in an internal representation
 2. eliminates variables instantiating the program
 3. generates answer sets, and verifies them
 4. process these answer sets following the instructions of the post-processors
- the main difference with **SMODELS** is that DLV handles **disjunctive programs**

- also, DLV only admits input programs with the **range restrictive property** (each variable in the head has to appear in a positive literal in the body)
- input programs allow **weak constraints** in the form

$$:\sim p_1, \dots, p_n, \textit{not } q_1, \dots, \textit{not } q_m \text{ [weight : level]}$$

where p_i, q_j are literals, and *weight*, *level* are integers

- given a program with weak restrictions, DLV returns the **preferred answer sets** assigning an order first by level, and then by weight in the set of constraints that they do not satisfy

Action Languages

	Smodels	DLV
programs	A-Prolog	A-Prolog ^{\neg, or}
queries	indirect by constraints	direct
modes	–	credulous\skeptical
extension	constraints, optimization	weak constraints, post-processors
arithmetic in rules	$p(T+1):-p(T)$	not admissible in the head
function symbols	limited support	not allowed
strong negation	allowed	allowed
anonymous variables	not allowed	allowed
rules	strongly range restricted	range restricted
API	C/C++ user-defined functions	–