IM1202-232411M
NOTES MODEL-BASED AI

Open Universiteit

Author
Jan Baljan
14-11-2023

This document contains (lecture) notes that were made during the course **Model-based Artificial Intelligence** at the Open Universiteit, the Netherlands. The notes are written based on the books on Answer Set Programming by Lifschitz [1] and Knowledge Representation, and the Design of Intelligent Agents by Gelfond and Khal [2]. Furthermore, the course workbook was used along with several papers from the literature.

Contents

1 Knowledge representation and models in AI

1.1 Knowledge representation

Davis et al. [3] argue that the idea of *knowledge representation* can best be understood in terms of five distinct roles that it plays. These roles are the following:

1. It is a fundamental **surrogate**, that is, a substitution for the thing itself, used to enable an entity to determine *consequences* by thinking rather than acting. In other words, by reasoning about the world rather than taking action in it.

2. It is a set of **ontological commitments**, that is, the objects that are implicitly assumed to exist and provide meaning to the formal language or system.

3. It is a **fragmentary theory of intelligent reasoning** expressed in terms of three components:

   (a) The representation's fundamental conception of intelligent reasoning.

   (b) The set of *inferences* that the representation permits.

   (c) The set of *inferences* that the representation recommends.

4. It is a medium for **pragmatically efficient computation**, that is, the *computational environment* in which thinking is accomplished. A representation provides guidance for organising information, so that the recommended inferences can be facilitated.

5. It is a medium for **human expression**, that is, a *language* in which we say things about the world.

At the heart of knowledge representation lies the idea of **reasoning**. In particular, intelligent reasoning, that can be defined by looking at the work from fields other than AI. Five fields have provided inspiration for the notion of intelligent reasoning, i.e., mathematical logic, psychology, biology, statistics, and economics.

- **Mathematical logic**: Historically, mathematical logic makes the assumption that intelligent reasoning is some variety of formal calculation, typically *deduction*. The modern view in AI is embodied by the logicists.

- **Psychology**: Rooted in psychology, this view sees reasoning as a *characteristic human behaviour* that has given rise to both extensive work on human problem solving and the large collection of knowledge-based systems.

- **Biology**: Loosely rooted in biology, this view considers reasoning to be the *architecture of the machinery* that accomplishes it. Thus, reasoning is a *characteristic stimulus-response behaviour* that emerges from the parallel interconnection of a large collection of very simple processors. Researchers working on *connectionism* are the current descendants of this line of work.

- **Statistics**: Derived from probability theory, this view adds to logic the notion of *uncertainty*, yielding a view in which reasoning intelligently means obeying the *axioms of probability theory*.

- **Economics**: This view adds to logic and uncertainty the further ingredient of *values* and *preferences*, leading to a view of intelligent reasoning that is defined by adherence to *utility theory.*

It is important to acknowledge the legitimacy of a variety of approaches to specifying sanctioned inferences through one or more of the above views. Traditional logic sanctions only inferences which are sound, i.e., encompassed by logical entailment. Probabilistic reasoning systems sanction inferences specified by probability theory, while rational agents rely on concepts from economic rationality. The frames theory, from the social sciences, is both informal and empirical, as an unavoidable consequence of its conception of intelligence.

Nevertheless, all of these forms of reasoning are valid and serve to remind us that a knowledge representation is a theory of intelligent reasoning. Consider the frames theory, which states that one who encounters a new situation, selects from memory a structure called a *frame.* A frame is a remembered framework to be adapted to fit reality by changing details as necessary. For instance, being in a certain kind of living room or going to a child's birthday party. This is similar to the notion of a predicate, in which its parameter values are not concrete. Thus, the frames theory can be rephrased and represented as the logical entailment:

$$F_1(\vec{X_1}), \ldots, F_n(\vec{X_m}) \models G_F$$

where F_i is a frame with a vector of variable arguments $\vec{X_j}$, which entails an adapted and filled in framework G_F. In other words, the formalised and abstract knowledge is taken from the knowledge base, combined, adapted, and filled in with concrete values that serve the situation of the formalised framework, and then presented as the conclusion of this reasoning (in the form of G_F). Filling in the variables for values is denoted as the process of *grounding.*

Based on the requirements of the EU about ethical guidelines for trustworthy AI [4], we can relate the above mentioned roles of knowledge representation and argue the benefit of using model-based artificial intelligence:

- **Reliability and reproducibility**: Knowledge representation is a surrogate of reality, and as such, is a formalised model of the real world. Although total reliability is not possible in any AI system, a system using knowledge representation is more robust in terms of handling various kinds of inputs. The reason being that the model is not only designed for a determined application context with a limited input space and dataset, but it encompasses a model of actual knowledge.

- **Traceability**: In knowledge representation, ontological commitments are explicit, which means that decisions about the design of the system that are normally implicit (black-box) are now easily traceable.

- **Accessibility and universal design**: Due to the declarative nature of modelling knowledge about the world, it is often easy to understand the resulting formalism without much technical background, i.e., it is a natural form of human expression.

1.2 Model-based and model-free

It is important to understand the key differences between **model-based solvers** and **model-free learners**, as these are the systems used in current AI developments. Solvers require a specification of the problem encoded in a certain representation, whereas learners require a set of problem instances for which they can learn the underlying patterns. Solvers are thus general, as they can easily deal with new problem instances, provided a suitable representation of the problem. Learners need experience on the related problems. Furthermore, learners are often very efficient, but require that the input and output are of a bounded size, they need relatively large datasets, and the output is heuristic and thus less precise. Solvers, on the other hand, are systematic and general, precise, and require no training data, but they are less efficient and harder to run in parallel.

Current accounts in psychology describe the dichotomy between model-based solvers and model-free learners. The human mind is thought to consist of two interacting systems: a **System 1** associated with the *intuitive mind*, and a **System 2** associated with the *analytical mind*. Common characteristics of these systems are:

System 1	**System 2**
fast	slow
associative	deliberative
unconscious	conscious
effortless	effortful
parallel	serial
automatic	controlled
heuristic	systematic
specialised	general

The parallel between System 1 and System 2 on one hand, and learners and solvers on the other hand, is evident. These processes in the human mind, however, are not independent. System 2 is assumed to be evolved more recently than System 1. For intelligent reasoning, there cannot be an analytical mind (System 2) without an intuitive mind (System 1). Furthermore, our System 1 is susceptible to influences by powerful markets and a weak political system, which leads to misalignment between the AI systems and human values. Instead, flexible and transparent AI systems need a System 2 that cannot be provided by learners alone. Thus, the necessity exists for integration between model-based and model-free approaches.

1.3 Introduction to ASP

Answer Set Programming (ASP) is a logic-based approach to artificial intelligence. It is a declarative programming language, which contrasts the usual imperative languages. The main distinction between these two sub-categories of programming languages lies in the approach of expressing problems.

The word *imperative* implies that we make imperative statements in the language, such as; “Take a walk outside.” or “Read the memory address of 0x42.”. The programmer is tasked with writing down the exact instructions that the computer needs to perform.

In contrast, the word *declarative* implies that we make declarative statements about the world. These statements describe the problem and its constraints, rather than the exact steps to solve the problem.

In summary, the programmer describes **how** to solve a problem in imperative languages, whereas the programmer describes **what** the problem is and **what** the underlying constraints are in declarative languages.

The mechanisms of ASP allow the programmer to describe the problem domain in a declarative way, i.e., expressing the facts about the world and the rules that govern the world. The rules are of the form $head \leftarrow body$, which can be read as the *head* is true if the *body* is true. Using the facts and the rules, new facts are obtained and used in rules. The recursive nature of this approach establishes new facts until no more facts can be established. The resulting answer set solution (or stable model) is the collection of derived facts. For example, rules can specify problems that aim at finding plans or finding explanations.

2 Syntax and semantics of ASP

2.1 Formal language and jargon

Answer Set Programming (ASP) like any other language can be defined by its syntax and semantics. To define the syntax of ASP, we can use the theory of formal languages and define its *alphabet*. In logic, this alphabet is usually called a **signature**.

2.1.1 Signature

Definition 2.1. Signature.
Formally, the signature of ASP is a four tuple $\sum = \langle \mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V} \rangle$ of disjoint sets. These sets contain the **objects**, **functions**, **predicates**, and **variables**, respectively.

Functions and predicates have an associated **arity** - a non-negative integer indicating the number of parameters.

Often, elements of $\mathcal{O}, \mathcal{F}$, and $\mathcal{P}$ are referred to as object, function, and predicate **constants** if they do not contain any variables.

An example signature can be obtained by defining some relations and filling in the sets for the associated relations. Consider the relationship of *parent* (*mother* and *father*) and *child*, and the relationship of one's own *gender*. Knowledge about these relationships can be derived from a small example family of three persons consisting of, *John* who identifies as *male* and is the father of *Sam*, *Alice* who identifies as *female* and is the mother of *Sam*, and *Sam* who identifies as *male* and is their child.

We can use a rule to derive the parent relation with the following logic.

$$parent(X,Y) \leftarrow father(X,Y).$$
$$parent(X,Y) \leftarrow mother(X,Y).$$

Intuitively, we say that X is a parent of Y if X is the father of Y or X is the mother of Y. Likewise for the child relation:

$$child(Y,X) \leftarrow parent(X,Y).$$

In other words, we say that Y is a child of X if X is a parent of Y.

As a result of applying these two rules with the prior knowledge about the family, the following statements can be derived.

$father(john, sam).$	$parent(john, sam).$	$child(sam, john).$
$mother(alice, sam).$	$parent(alice, sam).$	$child(sam, alice).$
$gender(alice, female).$	$gender(sam, male).$	$gender(john, male).$

The associated signature $\sum_f$ is thus:

$$\begin{aligned}
\mathcal{O} &= \{john, alice, sam, male, female\}\\
\mathcal{F} &= \varnothing\\
\mathcal{P} &= \{father, mother, gender, parent, gender\}\\
\mathcal{V} &= \{X, Y\}
\end{aligned}$$

2.1.2 Sorts

Sometimes it is convenient to expand the notion of signature by including it in another collection of symbols called **sorts**. Sorts are used to restrict the parameters of predicates, as well as the parameters and values of functions. Sorts act as **types** in procedural languages, in that they specify the contract of a relationship. In the previous example, we know that *John*, *Alice*, and *Sam* are individual persons. Thus, we can assign them to the sort (or type) *person*.

$$person = \{john, alice, sam\}.$$

Now that we have a sort for this subset of objects, we can specify the relationship of father more concretely, i.e., $father(person, person)$, which means that the parameters of the predicate $father/2$ is restricted to be of the sort $person$.

2.1.3 Terms

Definition 2.2. Terms.
Terms over signature $\sum$ are defined as follows:

1. Variables and object constants are terms.
2. If $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is also a term.

Object and function constants are used to create **terms**. Terms that have no variables (uppercase symbols) are often used to name objects in the (problem) domain. For instance, the object constant *sam* is a name of a person.

We can sub-divide terms into arithmetic terms, such as $2 + 3$ or $a \times b$, and **ground terms**. Ground terms contain no symbols for arithmetic functions $(+, -, *$, etc.) and no variables. Using the family logic, here are some examples:

- *john*, *alice*, and *sam* are ground terms.
- X and Y are terms (not ground terms).
- $father(X, Y)$ is not a term

The reason why $father(X, Y)$ is not a term is because $father/2$ is a predicate. Predicates are by definition not terms, whereas functions are used to construct terms. The distinction between predicates and functions lies in that functions can be nested, e.g., $f(f(\ldots f(X)\ldots))$, whereas predicates cannot be nested. Predicates can be used to construct atoms (which will be covered later). Functions with variables, on the other hand, form an expression that can be evaluated by filling in the variables with terms. The recursive nature of functions leads to the derivation of ground terms.

Therefore, as an example, if $2 + 3$ is present in a program, then its value 5 and $2 + 3$ are both terms; 5 is a ground term, whereas $2 + 3$ is not. If $f(X_1, X_2)$ and $g(Y)$ are functions present in a program, then $f(2, f(g(3), 8))$ is a ground term.

2.1.4 Example: Sorted signature

Consider the family signature again. Now, we add a new function symbol called $car/1$, which indicates a unary relation between the person and ownership of their car. Using this function, we can define *ground terms*, such as $car(john), car(alice), car(sam)$, and *non-ground terms*, such as $car(X)$ or $car(Y)$. The definition of car complicates things, as according to our definition, $car(car(sam))$ is also a ground term. This is not reasonable in the context of the domain. Therefore, to avoid this difficulty, we can consider the sorted signature $\sum_s$ defined by the following.

- Object constants are divided into sorts:

$$\begin{aligned} gender &= \{male, female\} \\ person &= \{john, alice, sam\} \\ thing &= \{car(X) : person(X)\} \end{aligned}$$

  Note that the set $thing$ uses set-building notation, meaning that $person(X)$ is a condition, if true, then $car(X)$ will be produced, i.e., $\forall x\ person(x)$ include $car(x)$.

- $\mathcal{F} = \{car\}$ where car is a function symbol that maps elements of sort $person$ to that of $thing$, i.e., car maps $john$ to $car(john)$.

- Predicate symbols now contain sorted parameters, such as $father(person, person)$ and $mother(person, person)$, which act as contracts/restrictions.

The definition of a term now requires that $f(t_1, \ldots, t_n)$ is a term if the sorts of the parameters $t_1, \ldots, t_n$ are compatible with the required parameter sorts of f.

2.1.5 Atoms and Literals

Definition 2.3. Atomic statement.
An atomic statement or simply an **atom** is an expression of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms. An atom is called **ground** if every term $t_1, \ldots, t_n$ is ground.

Given a sorted signature, these terms should correspond to the sorts assigned to the parameters of p.

Given $\sum_s$, then $father(john, sam)$ and $mother(alice, X)$ are atoms of the sorted signature $\sum_s$. Zero-arity predicates have their parentheses omitted. For example, the predicate is_above_18 has an arity of 0 and is an atom.

Definition 2.4. Literals.
A literal is an atom $p(t_1, \ldots, t_n)$ or its negation $\neg p(t_1, \ldots, t_n)$. The latter is often read as $p(t_1, \ldots, t_n)$ is *false* and is referred to as a **negative literal**. An atom or its negation are called **complementary** - a complementary literal l is denoted by $\bar{l}$. Ground atoms and their negations are referred to as **ground literals**.

As an example, let $p(a)$ be an atom, then $\neg p(a)$ is the complement of $p(a)$, i.e, $\overline{p(a)}$. Furthermore, $\neg\neg p(a)$ is the complement of $\overline{p(a)}$, i.e., $\overline{\overline{p(a)}} = p(a)$.

2.1.6 Rules

With the aforementioned definitions, we can describe the syntax of an ASP program. Note that the term "program" and "knowledge base" are used synonymously. A program Π consists of a signature $\sum$ and a collection of **rules** of the general form:

$$l_0 \textit{ or } \ldots \textit{ or } l_i \leftarrow l_{i+1}, \ldots, l_m, \textit{ not } l_{m+1}, \ldots, \textit{ not } l_n.$$

In `clingo`, we replace $\neg$ with `-`, $\leftarrow$ with `:-`, and *or* with $|$. The symbol *not* is a distinct *logical connective* called **default negation** (or **negation as failure**). *not l* is often read as *"it is not believed that l is true"*. This, however, does not imply that l is false. A rational reasoner could conclude that there is insufficient evidence, and thus belief neither statement p nor its negation $\neg p$. Therefore, default negation *not* is different from the classical logical negation $\neg$, in that $\neg p$ states that p is *false*, whereas *not p* is a statement about belief.

The disjunction *or*, referred to as **epistemic disjunction**, is also different from its logical counterpart. The propositional statement $p \vee \neg p$ is a tautology (always true), whereas p *or* $\neg p$ implies that p is believed to be true or believed to be false. Since a rational reasoner can remain uncertain, it is certainly not a tautology.

Definition 2.5. ASP Rule.
The left hand side of a rule is called the **head** and the right hand side the **body**. The body can be viewed as a set of **extended literals** (or *e-literals* or *premises*) - literals possibly preceded by *not*.

A rule with an empty head is referred to as a **constraint** and written as

$$\leftarrow l_{i+1}, \ldots, l_m, \textit{ not } l_{m+1}, \ldots, \textit{ not } l_n.$$

A rule with an empty body is referred to as a **fact** and written as

$$l_0 \textit{ or } \ldots \textit{ or } l_i.$$

A rule r with variables is viewed as the set of **ground instantiations**. These are rules obtained from r by replacing its variables by ground terms of $\sum$ and evaluating arithmetic terms, e.g., replacing $2 + 3$ with 5. This process is called the **grounding** of Π.

2.1.7 Satisfiability of Syntactic Constructs

Definition 2.6. Satisfiability.
A set S of ground literals satisfies

1. literal l **if** $l \in S$;
2. Default negation of a literal $\textit{not } l$ **if** $l \notin S$;
3. Epistemic disjunction $l_1 \textit{ or } \dots \textit{ or } l_n$ **if** for some $1 \leq i \leq n, l_i \in S$;
4. A set of ground extended literals **if** S satisfies every element of this set;
5. rule r **if** whenever S satisfies the body, it satisfies the head.

As an example consider the set S

$$S = \{\neg p(a), q(b), \neg t(c)\}$$

and the rule r

$$p(a) \textit{ or } p(b) \leftarrow q(b), \neg t(c), \textit{ not } t(b).$$

To check whether the set S satisfies r, we need to first check whether the body of r satisfies S. For the first two extended literals in the body, we see that they are indeed in S (using clause 1). The last literal $\textit{not } t(b)$ is also satisfied as $t(b) \notin S$ (using clause 2). Thus, by clause 4, we have that the body satisfies S, which means that the head **must** satisfy S as well (using clause 5). However, neither $p(a)$ nor $p(b)$ are in S (using clause 3). Therefore, S does not satisfy r. This result comes directly from the truth table of the implication, as denoted in table 1.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Table 1: Truth table of $p \rightarrow q$.

The following enumeration shows several examples of varying sets S:

- $S = \varnothing$: does satisfy r, because both the body and the head of r are not satisfied by $\varnothing$, which makes the implication true ($F \leftarrow F \implies T$).
- $S = \{p(a)\}$: does satisfy r, because the body of r is not satisfied, but the head is satisfied (using clause 3), which makes the implication true ($T \leftarrow F \implies T$).
- $S = \{q(b), t(c)\}$: does satisfy r, because both the body and the head of r are not satisfied by S, which makes the implication true ($F \leftarrow F \implies T$).
- $S = \{p(a), q(b), \neg t(c)\}$: does satisfy r, because both the body and the head of r are satisfied by S, which makes the implication true ($T \leftarrow T \implies T$).

Later, we will see that the minimal set S is indeed the answer set solution to the program. In this case $S = \varnothing$.

2.2 Informal Answer Set Semantics

Informally, an ASP program is a specification for answer sets - sets of beliefs held by a rational agent. Answer sets are also known as **Stable Models**. The agent must be guided to produce these answer sets by way of the following *informal principles*:

1. Satisfy the rules of Π, i.e., believe in the head of a rule if you believe in its body.
2. Do not believe in contradictions.
3. Rationality principle: believe nothing you are not forced to believe.

The following illustrate several constructions of programs with their associated semantic meaning.

2.2.1 Modus Ponens

Consider the classic modus ponens example:

$p(b) \leftarrow q(a).$	"Believe $p(b)$ if you believe $q(a)$."
$q(a).$	"Believe $q(a)$."

The resulting answer set is $\{p(b), q(a)\}$, which satisfies the rules of the program. As an intuitive example, we can say that $q(a)$ represents the belief that is is raining and $p(b)$ represents the belief that the streets are wet. Now, since we believe the fact $q(a)$ that it is raining, due to the body of the rule being satisfied, we must also believe in $p(b)$ that the streets are therefore wet.

2.2.2 Epistemic Disjunction

Consider the example of using the epistemic disjunction connective:

$p(a)\ \mathit{or}\ p(b).$	"Believe $p(a)$ or believe $p(b)$"

The three sets that satisfy this body-less rule are $\{p(a)\}$, $\{p(b)\}$, and $\{p(a), p(b)\}$. However, according to the rationality principle, only the first two are answer sets. It would be irrational to believe the third set as it would mean that we belief more than is necessary.

It is important to distinguish epistemic disjunction from the exclusive or (XOR), even though they are very similar. Consider the following program

$$
\begin{aligned}
&p(a)\ \mathit{or}\ p(b).\\
&p(a).\\
&p(b).
\end{aligned}
$$

The answer set is $\{p(a), p(b)\}$, due to the second and third rule. Therefore, if *or* were exclusive, then the program would be contradictory.

To express XOR, we can use the following rules:

$$p(a) \textit{ or } p(b).$$
$$\neg p(a) \textit{ or } \neg p(b).$$

This program would produce the answer sets: $\{p(a), \neg p(b)\}$ and $\{\neg p(a), p(b)\}$. The reason is because if you choose $p(a)$ in the first rule, then to avoid a contradiction, we cannot choose $\neg p(a)$ in the second rule. Otherwise, the answer set would be $\{p(a), \neg p(a)\}$, which is a contradiction. Therefore, the only remaining option is to choose $\neg p(b)$. Likewise, when choosing $p(b)$ in the first rule and choosing $\neg p(a)$ in the second rule.

2.2.3 Constraints

$p(a) \textit{ or } p(b).$	"Believe $p(a)$ or believe $p(b)$."
$\leftarrow p(a).$	"It is impossible to believe $p(a)$."

The second rule is a constraint, which forces us to not believe $p(a)$. This leads to the answer set $\{p(b)\}$. A constraint does not introduce new information, but rather limits the possible beliefs that a rational reasoner might have. Constraints can be viewed as filters.

2.2.4 Default negation

An agent can make conclusions based on the absence of information. For example, an agent might enjoy its free time after class if there is lack of belief that the agent has detention. Such reasoning is captured by default negation. Consider the following example:

$p(a) \leftarrow \textit{ not } q(a).$	"If $q(a)$ does not belong to your set of beliefs, then $p(a)$ must belong to your set of beliefs."
$p(b) \leftarrow \textit{ not } q(b).$	"If $q(b)$ does not belong to your set of beliefs, then $p(b)$ must belong to your set of beliefs."
$q(a).$	"Believe $q(a)$."

Starting from the body-less rules (facts), we see that $q(a)$ is always in the answer set. This means that the body of the first rule is never satisfied. Since there is no rule in the program with the head $q(b)$, we cannot be forced to belief $q(b)$. Thus, the body of the second rule is satisfied, and $p(b)$ must be believed. In conclusion, the resulting answer set is $\{q(a), p(b)\}$.

2.2.5 Closed World Assumption (CWA)

The Closed World Assumption implies that a statement that is true is also known to be true. Therefore, conversely, what is not know to be true, i.e., unknown, is false. In other words, CWA implies that a lack of knowledge is automatically considered false. To achieve this assumption in ASP, the following program suffices.

$\neg q(X) \leftarrow \textit{ not } q(X).$	"If $q(X)$ is not believed to be true, believe that it is false."

It guarantees that the answer sets of a program are complete with respect to the given predicate. In this case, every answer set produced by the program must contain either $q(t)$ or $\neg q(t)$ for every ground term t.

For example, in a database, if we are not able to find some evidence e or derive e, then we assume that e is false.

2.3 ASP Entailment

2.3.1 Monotonicity

Definition 2.7. Entailment.
A program Π **entails** a literal l if l belongs to all answer sets of Π. For entailment, we write

$$\Pi \models l.$$

Π entails a set of literals if it entails every literal in this set. Often instead of saying Π entails l, we say that l is a **consequence** of Π.

Entailment in ASP differs from entailment in classical logic. In classical logic, entailment forms the basis for mathematical reasoning and has the important property of **monotonicity**. When the addition of new axioms to a theory T cannot decrease the set of consequences of T, we say that the reasoning is monotonic. Formally, the entailment relation $\models$ is called monotonic if for every A, B, and C, and entailments $A \models B$ and $A, C \models B$, it holds that if $A \models B$, then $A, C \models B$. This property guantees that a mathematical theorem, once proven, stays proven. That is, further evidence cannot lead to retractions of the consequences. This is how classical logic operates.

However, this is **not** the case for ASP entailment. The addition of new information to program Π may invalidate a previous conclusion. In other words, for a **non-monotonic** entailent $\models$ relation, $A \models B$ does not guarantee that $A, C \models B$. For instance, program $\Pi_1 = \{p(a) \leftarrow \textit{ not } q(a).\}$ entails $p(a)$, i.e.,

$$\Pi_1 \models p(a).$$

However, the addition of a new fact $q(a)$ would retract this conclusion

$$\begin{aligned}\Pi_2 &= \Pi_1 \cup \{q(a)\} \\ \Pi_2 &\not\models p(a)\end{aligned}$$

Non-monotonic logic is related to commonsense reasoning, as conclusions are tentative, and able to be retracted based on further evidence.

2.3.2 Query an ASP program

Given a program Π, we can use entailment to find the truth values of literals. This can be done via a *query*.

Definition 2.8. Answer to a Query.

- The answer to a **ground conjunctive query** of the form $l_1 \wedge \cdots \wedge l_n$, where $n \geq 1$, is
  - *Yes* if $\Pi \models \{l_1, \ldots, l_n\}$,
  - *No* if there is i such that $\Pi \models \overline{l_i}$,
  - *Unknown* otherwise
- The answer to a **ground disjunctive query** of the form l_1 *or* $\ldots$ *or* l_n, where $n \geq 1$, is
  - *Yes* if there is i such that $\Pi \models l_i$,
  - *No* if $\Pi \models \{\overline{l_1}, \ldots, \overline{l_n}\}$
  - *Unknown* otherwise
- The answer to a query $q(X_1, \ldots, X_n)$ where $X_1, \ldots, X_n$ is a list of variables occurring in q is a **sequence** of ground terms $t_1, \ldots, t_n$ such that $\Pi \models q(t_1, \ldots, t_n)$

Consider the following program

$$p(a) \leftarrow \ not\ p(b).$$
$$p(c).$$

which has the answer set $S = \{p(a), p(c)\}$. The answer to the query $?p(a)$ is *Yes*, because $\Pi \models p(a)$ in all answer sets. The answer to the query $?p(b)$ is Unknown, because Π entails neither $p(b)$ nor $\neg p(b)$.

The answer to the query $?(p(b) \vee p(c))$ is *Yes*, because there is an element in the disjunctive query that is the consequence of Π, i.e., $\Pi \models p(c)$. The answer to the query $?\neg p(c)$ is *No*, because $\Pi \models p(c)$, which is the complement of $\neg p(c)$.

2.4 Formal Answer Set Semantics

The formal definition of an answer set can be defined in two parts. First, we define an answer set for programs **without** *default negation* - programs with *not* l that read as "it is not believed that l is true". The second part of the definition explains how to remove *default negation* so that the first part can be applied.

Definition 2.9. Consistency.
Pairs of literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called **contrary**, i.e., one is the complement of the other and vice versa. A set S of ground literals is called **consistent** if it contains **no** contrary literals.

Definition 2.10. Answer Sets, part I.
Let Π be a program that does not contain *default negation*, i.e., consisting of the form:

$$l_0 \ or \ \dots \ or \ l_i \leftarrow l_{i+1}, \dots, l_m.$$

An **answer set** of Π is a *consistent* set S of ground literals such that

- S satisfies the rules of Π (using the definition of satisfiability); and
- S is minimal - there is no proper subset of S that satisfies the rules of Π.

The three informal principles are now formalised and their relation is the following:

- Satisfy the rules of Π, which is the same as the formal definition.
- Do not believe in contradictions. Program Π needs to be consistent.
- Rationality principle: believe nothing you are not forced to believe. The answer set S must be minimal.

Now with the above formal semantics, we can dissect several examples. Starting with

$$p(a) \ or \ p(b).$$

The sets $\{p(a)\}, \{p(b)\}$ and $\{p(a), p(b)\}$ all satisfy the program. However, due to the minimality constraint, $\{p(a), p(b)\}$ is not an answer set. Instead, the remaining sets are consistent and minimal answer sets. Another interesting aspect displayed by this program is that it **entails** neither $p(a)$ nor $p(b)$, because neither occur in **all** answer sets. Therefore, we cannot query the program to find the truth value of either ground literal.

Consider the following program consisting of a constraint.

$$p(a) \ or \ p(b).$$
$$\leftarrow p(a).$$

There are two minimal sets that satisfy the first rule, i.e., $S_1 = \{p(a)\}$ and $S_2 = \{p(b)\}$. We can see that S_1 satisfies the body of the second rule. However, it is not possible to satisfy an empty head if the body is satisfied. Therefore, the whole implication cannot be satisfied, which means that S_1 cannot be an answer set. S_2 is thus the only answer set.

Definition 2.11. Answer Sets, part II.
Let Π be an arbitrary program and S be a set of ground literals. By Π^S we denote the program obtained from Π by:

1. Removing all rules containing *not* l such that $l \in S$.
2. Removing all other premises containing *not* .

We refer to Π^S as the **reduct** of Π with respect to S. Thus, S is an answer set to Π if S is an answer set to Π^S.

2.4.1 Example: default negation reduct

As an example, consider the following program Π that consists of *default negations*.

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

To assert that $S = \{q(a), p(b)\}$ is the only consistent, minimal, and satisfiable answer set, we can apply part II of the answer set definition.

Using clause 1, we can remove the entire first rule, because the body contains $not\ q(a)$ and $q(a) \in S$. Then, using clause 2, we can remove the premise in the body of the second rule. This leaves an empty body in its place. The reduct of Π is thus the following program.

$$
\begin{aligned}
&p(b).\\
&q(a).
\end{aligned}
$$

It is now trivial to see that S is indeed the correct answer set, because S is the only answer set of Π^S.

2.4.2 Example: Inconsistent program

As another example, consider the program Π.

$$p(a) \leftarrow \; not\ p(a).$$

Intuitively, we have two candidate answer sets. Either the set $S_1 = \{p(a)\}$ or $S_2 = \varnothing$.

The reduct of Π with respect to S_1 is $\Pi^{S_1} = \varnothing$, because we can remove the only rule from Π. However, the only answer set of Π^{S_1} is $\varnothing$, which means that S_1 is not minimal. Thus, S_1 is not an answer set to the original program Π.

Next, the reduct of Π with respect to S_2 is $\Pi^{S_2} = \{p(a) \leftarrow .\}$. The only answer set to this reduct is $\{p(a)\}$. Therefore, S_2 is not an answer set.

Interestingly, we see that we cannot deduce an answer set solution to the program. This is not surprising, as the original program Π denotes that we must believe $p(a)$ if we do not believe $p(a)$. An ASP program without answer sets is referred to as **inconsistent**.

2.5 Representing Defaults

A **default** is a statement of natural language that indicates typical or common behavior in the absence of complete information. For instance, words like "normally", "typically", or "as a rule" are used in default statements. It is common for humans to reason by stating defaults, their exceptions, and to learn the skill of reasoning with them. This concept ties back to the concept of **non-monotonic** reasoning (or commonsense reasoning), where further evidence about phenomena can retract our former belief in a phenomenon.

Definition 2.12. Default.
In ASP, a default d, which is stated as "Normally, elements of class C have property P" is often represented with the rule:

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X) \end{aligned}$$

2.5.1 Example: Morality of Burglar

Consider the following example about the morality of burglars:

1. Nami is a burglar.
2. *Normally*, burglars are bad people.
3. Therefore, Nami is a bad person.

The second statement is an example of a default. To model this as a program, we can ignore the peculiarity of the default for now, which results in the following program Π_1.

$burglar(nami).$	"Believe that Nami is a burglar."
$bad(X) \leftarrow burglar(X).$	"Believe that X is a bad person if X is a burglar."

As expected, the answer set to Π_1 is $\{bad(nami), burglar(nami)\}$.

Assume now that in addition to the default (statement 2), we learn that Nami is an exception. She steals from other bad people, and gives to those in need. We can express this sentiment with an additional predicate $stole(X, Y)$, which indicates that we believe X stole from Y. Furthermore, we can adjust the rule ($bad(X) \leftarrow burglar(X).$) by adding two additional literals.

$bad(X) \leftarrow burglar(X),$	
$not\ ab(default\_bad(X)),$	"There is no evidence that X is abnormal."
$not\ \neg bad(X)$	"There is no evidence that X is not bad.".

Here, the literal $ab(default\_bad(X))$ has the meaning that we believe X is abnormal with respect to the default rule about burglars being bad people. Therefore, if we are able to deduce that X is an abnormality, then it is not possible to deduce the truthfulness of the fact that X is a bad person. This type of exception is called a **weak exception**. Note that this is not the same as concluding $\neg bad(X)$, which is a strong statement that we make about X, that is, it is not true that X is a bad person. Rather, we are unable to claim neither $bad(X)$ nor $\neg bad(X)$.

The other additional literal $\neg bad(X)$ tells us that X is not a bad person, and $not\ \neg bad(X)$ tells us that it is not possible to derive that X is a bad person. This type of exception is called a **strong exception**.

Definition 2.13. Exception axioms.
Weak exceptions make the default inapplicable, i.e., make the agent unable to use the default to come to a hasty conclusion. **Strong exceptions** refute the default's conclusion, i.e., allow the agent to derive the opposite of the default.

A weak exception $e(X)$ to a default d can be encoded with the **cancellation axiom**

$$ab(d(X)) \leftarrow \; not \, \neg e(X).$$

If e is a strong exception, then we need one more rule:

$$\neg p(X) \leftarrow e(X).$$

The following listing illustrates the above rule and additionally the logic for weak and strong exceptions. The code is written in `Clingo` syntax.

```
1  burglar(nami).
2  burglar(john).
3
4  bad(arlong).
5
6  stole(nami, arlong).
7
8  % strong exception
9  -bad(nami) :- bad(Y),
10               stole(nami, Y),
11               nami != Y.
12
13 % weak exception
14 ab(default_bad(X)) :- burglar(X), bad(Y),
15                       not -stole(X, Y), X != Y.
16
17 bad(X) :- burglar(X),
18           not ab(default_bad(X)),
19           not -bad(X).
20
21 #show -bad/1.
22 #show bad/1.
```

Due to the `#show` directive, the program's answer set is limited to the truthfulness of `bad/1`. Therefore, the program yields $\{bad(arlong), \neg bad(nami)\}$. The *strong exception* is applied on line 9 to make sure that Nami, specifically, is an exception if she stole from another bad person. Using a variable instead of Nami would have generalised the exception to the set of all burglars.

The weak exception, applied on line 14, ensures that we cannot make hasty claims about the morality of burglars, because we do not yet have enough information. That is why neither $bad(john)$ nor $\neg bad(john)$ is included in the answer set. However, if we observe more data about John's burglaries, then this outcome might change.

3 Basic Modeling

3.1 Choice rules

Up until now, every logic program had a single stable model solution. However, a problem may have possibly many solutions or no solution at all. This is analogous to finding the roots of an equation, in which there can be one, none, or more than one solutions.

In `Clingo`, we can express the idea of finding multiple stable models with **choice rules**. Choice rules indicate alternative ways to form a stable model. They give us the ability to include some form of non-determinism to our logic programs, because there is not one answer set anymore. The head of a choice rule includes an expression in braces.

```
{p(a); q(b)}.
```

This rule describes all possible ways to choose from the atoms `p(a)` and `q(b)`. There are four combinations, and thus four stable models.

```
{}
{p(a)}
{q(b)}
{p(a), q(b)}
```

For large amounts of atoms inside the braces, the possible number of combinations can be vast, i.e., the powerset of the choice construct. Therefore, to express lower and upper bounds on the size of the produced subsets, we can put integers before and after the braces indicating lower and upper bounds, respectively. For example, `1 {p(1..3)} 2` expresses the subsets of the set $\{p(1), p(2), p(3)\}$ that consist of only 1 or 2 elements. Thus, this program yields the stable models:

```
{p(1)}                {p(2)}                {p(3)}
{p(1), p(2)}          {p(2), p(3)}          {p(1), p(3)}
```

If the lower and upper bound are equal to each other, e.g., `3 {p(1..3)} 3`, then a different format can be used with an equal sign.

```
{p(1..3)} = 3.
```

Programs with multiple choice rules have more possibilities for stable models. For instance,

```
1 {p(1..3)} 2
{p(4)}
```

has 6 stable models due to the first choice rule, and for each of these models, there are two choices, either include `p(4)` or not. Therefore, there are 12 stable models.

3.2 Variables in Choice rules

Using variables in choice rules allows for the classical set building notation in `Clingo`. Suppose we are interested in the set of integers from 1 to N where $N = 3$ and each integer is either positive or negative. We could represent this as follows.

```
{num(X); num(-X)} = 1 :- X = 1..3.
```

This results in 8 stable models, because for each number $n \in [1 \dots 3]$, either positive or negative n can be included in an answer set.

```
{num(1), num(2), num(3)}       {num(-1), num(2), num(3)}
{num(1), num(2), num(-3)}      {num(-1), num(2), num(-3)}
{num(1), num(-2), num(3)}      {num(-1), num(-2), num(3)}
{num(1), num(-2), num(-3)}     {num(-1), num(-2), num(-3)}
```

In general, the program has 2^N stable models, because for `X` in the range $1 \dots N$, we have a choice between either `num(X)` or `num(-X)`.

Variables can also be used inside a choice rule **locally**, which means that the expression in between braces is expressed in terms of predicates defined earlier. For instance, suppose we construct a list of facts `n(1..10)`, which indicates the set of numbers from 1 to 10. Using local variables and set building notation, we can derive the squares of the numbers.

```
{square(X*X) : n(X)} = 1.
```

Local variables are syntactically distinguished by the fact that all occurrences of the variables are between the braces.

Variables that are not between the braces, i.e., are not local, are said to be **global**. The difference is that substituting new values for a global variable produces new instances of the rule, whereas substituting values for a local variable does not produce new instances of the rule.

For instance, consider the program:

```
p(1; 2; 3).
n(1; 2).
{q(X, Y) : n(Y)} = 1 :- p(X).
```

To ground this program, we start with grounding the *global variables*, which results in the program:

```
p(1; 2; 3).
n(1; 2).
{q(1, Y) : n(Y)} = 1 :- p(1).
{q(2, Y) : n(Y)} = 1 :- p(2).
{q(3, Y) : n(Y)} = 1 :- p(3).
```

Then, for each of the instantiated choice rules, we can ground the *local variables*, which results in the program:

```
p(1; 2; 3).
n(1; 2).
{q(1, 1); q(1, 2)} = 1 :- p(1).
{q(2, 1); q(2, 2)} = 1 :- p(2).
{q(3, 1); q(3, 2)} = 1 :- p(3).
```

The program is now **ground**, and also *simplified*, so that we can reason and compute the stable model solutions. The first choice rule consists of a choice between two atoms, the second consists of a choice between two *different* atoms, and likewise for the third choice rule. Thus, giving $2 \times 2 \times 2 = 2^3 = 8$ stable model solutions.

3.3 Anonymous variables

Sometimes variables need to be used in rules that are irrelevant to the expected outcome. These variables can be made anonymous by using an underscore. For example, consider a database of students and their department using the predicate $studies(X, Y)$. We wish to extract the list of students.

```
studies(dave, english; mary, cs; bob, cs; pat, math).
student(X) :- studies(X, _).
```

Here, the second argument to $studies/2$ is ignored. In reality, `Clingo` *projects out* anonymous variables by using **auxiliary predicates**. The above program then becomes:

```
studies(dave, english; mary, cs; bob, cs; pat, math).
aux(X) :- studies(X, Var).
student(X) :- aux(X).
```

3.4 Modeling Knowledge base

To reason about the world, an agent must have information about it. We decide exactly what kind of agent we are building and educate the agent appropriately. The collection of statements about the world that we choose to give to the agent is called the **knowledge base** (KB).

3.4.1 Example: Family Relationships

Previously, we introduced basic family relationships, such as the membership atoms $person(john)$, $person(sam)$, and $person(alice)$, and their binary relationship $father(john, sam)$ and $mother(alice, sam)$.

In addition, $gender(male)$ and $gender(female)$ were introduced along with some basic rules.

$$gender\_of(john, male).$$
$$gender\_of(sam, male).$$
$$gender\_of(alice, female).$$

$$parent(X, Y) \leftarrow father(X, Y).$$
$$parent(X, Y) \leftarrow mother(X, Y).$$
$$child(X, Y) \leftarrow parent(Y, X).$$

To expand on this KB example about family relationships, we can introduce a new person to the family, Bill. Bill is the younger brother of Sam. Intuitively, the following rule can be applied to derive $brother(X, Y)$ where X is the brother of Y.

$$\begin{aligned} brother(X, Y) \leftarrow\ & gender\_of(X, male). \\ & father(F, X). \\ & father(F, Y). \\ & mother(M, X). \\ & mother(M, Y). \end{aligned}$$

This rule says that X is a brother of Y if X is male and both X and Y have the same parents. Unfortunately, this rule does not hold up, as the answer to the query $?brother(sam, X)$ gives the surprising answer, $brother(sam, sam)$. To fix the above rule, we need to add an additional term, $X \neq Y$ to explicitly state that X must be distinct to Y.

The reason why this error occurred is because we as humans take it for granted that X and Y must be distinct, but the reasoner does not unless we tell it to. A very large part of our knowledge is so deeply ingrained in us that we do not normally think about it. To bring this knowledge out, we need a well-developed skill of introspection. This type of reasoning is called **commonsense reasoning**.

Even though the reasoning agent can answer many questions about the family relationships, there are still questions yielding interesting answers.

$$?father(alice, bill)$$
$$?father(bill, sam)$$
$$?father(john, bob)$$

The first query answers *unknown*, even though it is a definitive *no*. A female cannot be the father of a person. To establish this, we can incorporate the following information.

$$\neg father(X, Y) \leftarrow gender\_of(X, female).$$

The second query also answers *unknown*, but we know that Bill and Sam are brothers.

Thus, this query should also answer with *no*. This is expressed with the following rule.

$$\neg father(X, Y) \leftarrow father(Z, Y), X \neq Z.$$

However, the above rule will produce an error about unsafe variable X.

Definition 3.1. Unsafe variables.
A rule is **unsafe** if it contains an unsafe variable - one that does not occur in a literal in the body that is neither built-in nor preceded by default negation.

In other words, a safe rule is one where every variable occurs in at least one literal in the body that is neither built-in nor preceded by default negation.

In the previous rule, the variable X is unsafe, because it only occurs in the built-in predicate $X \neq Z$. We can make the rule safe by noting that X is a parameter to $father$ and hence is of sort $person$. Adding this information eliminates the unsafe error, since now every variable is represented in the body by literals that are neither built-in nor default negation operators.

$$\neg father(X, Y) \leftarrow father(Z, Y), X \neq Z, person(X).$$

As for the third query, which reads "is John the father of Bob", there are two assumptions we can make. First, if we assume that we are given complete information about the family relationships, then of course, the answer is *no*. This is the **Closed World Assumption** (CWA), where the *unknown* is treated as *false*. This assumption states that since there is no mention of John being the father of Bob, it is safe to assume that John is not the father of Bob.

The second assumption views this query with incomplete information, where it is not certain that we have been given the complete information about John's family. Hence, the cautious answer *unknown* will be yielded. This is the **Open World Assumption** (OWA), where the *unknown* is still treated as *unknown* unless there is evidence to make a claim about its truthfulness.

ASP reasoning uses the second assumption, OWA, to solve the problem. Nevertheless, to use the Closed World Assumption, we need to explicitly state that default negation, i.e., not being able to say anything about the truthfulness, should entail logical negation.

$$\begin{aligned}\neg father(X, Y) \leftarrow\ & person(X), person(Y),\\ & not\ father(X, Y).\end{aligned}$$

This rule says that if there is no reason to believe that X is the father of Y, then X is not the father of Y. Note that the addition of $person(X)$ and $person(Y)$ ensure that X and Y are safe variables.

Adding to our existing knowledge base, we can address the notion of ancestors, i.e., X is an ancestor of Y, given the information of a family tree. Consider figure 3.1 for an example tree.

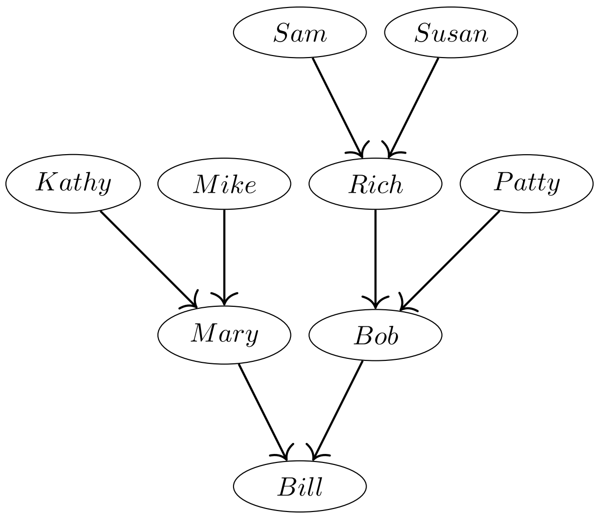


Figure 3.1: Example directed tree of a family.

For example, Mike is an ancestor of Bill, but so is Susan. Since this tree can be arbitrarily deep and we do not know how many parents of persons we need to consider, it is useful to express this notion with a **recursive definition**. The predicate *ancestor* can be recursively defined as follows.

$$\begin{aligned} ancestor(X,Y) &\leftarrow parent(X,Y). && \textit{Base case} \\ ancestor(X,Y) &\leftarrow parent(Z,Y), ancestor(X,Z). && \textit{Inductive step} \\ \neg ancestor(X,Y) &\leftarrow person(X), person(Y), && \textit{Closed world assumption} \\ & \qquad not\ ancestor(X,Y). \end{aligned}$$

3.4.2 Example: Hierarchical representation

Consider the following statements:

A lion is a mammal.

Mammals and reptiles are animals.

A snake is a reptile.

Animals are organisms.

Mammals do not lay eggs.

Since there is a hierarchy of **classes** and **sub-classes**, we can represent this knowledge in a tree-structure. From the statements, we can ascertain several classes of objects, such as mammals, reptiles, animals, organisms, and egg-layers. The lion and snake are not classes, but rather **concrete objects**. Consider the tree-structure of figure 3.2.

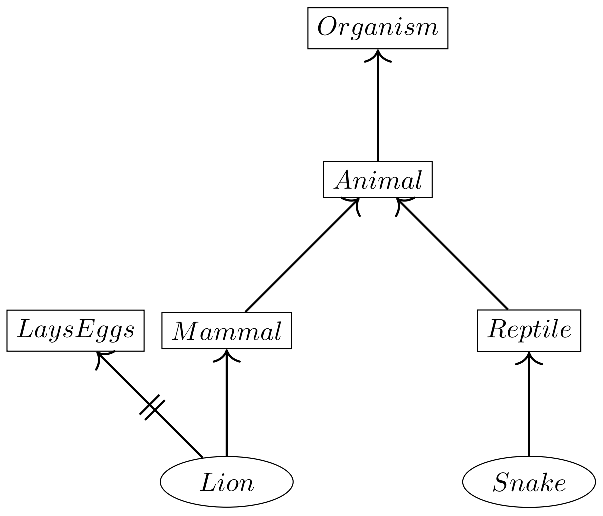


Figure 3.2: Animal hierarchical structure.

We can first define the membership of this knowledge base.

```
class(organism).                    class(laysEggs).
class(mammal).                      class(reptile).
class(animal).
is_subclass(mammal, animal).
is_subclass(reptile, animal).
is_subclass(animal, organism).
```

Then, we define the recursive predicate to infer whether X is a subclass of Y.

```
subclass(X, Y) :- is_subclass(X, Y).
subclass(X, Y) :- is_subclass(X, Z), subclass(Z, Y).
-subclass(X, Y) :- class(X), class(Y), not subclass(X, Y).
```

Now we can infer, for example, that a mammal is an organism. To also include the concrete objects, i.e., lion and snake, in this representation, we need to represent them as *objects* and also assign them to the corresponding class membership.

```
object(lion).              object(snake).
is_a(lion, mammal).        is_a(snake, reptile).
```

Again, using a recursive rule, we can define the `member/2` predicate.

```
member(X, Y) :- is_a(X, Y).
member(X, Y) :- is_a(X, Z), subclass(Z, Y).
-member(X, Y) :- object(X), class(Y), not member(X, Y).
```

4 Combinatorial Modeling

In a combinatorial search problem, the goal is to find a solution among a finite number of candidates. The ASP approach is to encode such a problem as a logic program whose stable models correspond to solutions, and then use an answer set solver to find the stable models.

4.1 Search space

Often, we want to start with the full set of possibilities, and then limit our search space by using constraints. To address the full search space, we can use the semantics of choice rules. Recall that a choice rule without any bounds specifies the subsets of some set building expression. For example,

```
n(1..3).
{square(X*X) : n(X)}.
```

Creates the squares of the subsets of $\{1 \ldots 3\}$, i.e., creates the powerset. The amount of stable models is thus 2^3, because for each squared number, it can either be included in the subset or not. The program outputs the following stable models:

```
{}
{square(4)}
{square(9)}
{square(1)}
{square(4), square(9)}
{square(1), square(9)}
{square(1), square(4)}
{square(1), square(4), square(9)}
```

Setting the upper and lower bound to 1 would yield 3 stable models, each containing a single distinct squared number. This is useful, because often we want to build a set using not only local variables, but also global variables. To understand this notion, consider filling a grid of cells with r rows and c columns.

```
rows(1..4).
cols(1..4).
{cell(X, Y) : rows(X)} = 1 :- cols(Y).
```

This logic program has 256 stable models. For each of the 4 column, there are 4 possible choices of rows, i.e., 4^4. Intuitively, we can understand this by looking at the global variable Y, which causes the rule to be duplicated and substituted for all possible Y during the grounding phase. This results in:

```
rows(1..4).
cols(1..4).
{cell(X, 1) :  rows(X)} = 1 :- cols(1).
{cell(X, 2) :  rows(X)} = 1 :- cols(2).
{cell(X, 3) :  rows(X)} = 1 :- cols(3).
{cell(X, 4) :  rows(X)} = 1 :- cols(4).
```

Notice now that this approach guarantees that there are exactly 4 different cells in each of the 256 answer sets. Another way to understand the resulting stable models is that this program yields all possible 4 cell combinations on a grid of 4×4. To get the entire grid filled with cells, we need to replace the upper/lower bound by the amount of rows, i.e., `{cell(X, Y) : rows(X) } = 4 :- cols(Y).`, which yields a single stable model.

4.1.1 Example: Partitioning sum-free numbers

Consider the problem of partitioning a set $X = \{1 \dots n\}$ into k subsets that are *sum-free*. A sum-free set is denoted by the property that the sum of every two elements of X is not an element of X. For example, for $n = 4$ and $k = 2$, we can make the partitioned subsets $\{1, 4\}$ and $\{2, 3\}$, because $1 + 4 \notin \{1, 4\}$ and $2 + 3 \notin \{2, 3\}$.

To encode this problem in ASP, we can first consider modeling the search space, i.e., all possible partitions of n numbers.

```
num(1..n).
part(1..k).
{set(X, K) : part(K)} = 1 :- num(X).
```

Here, `set(X, K)` means that number X is in partition K. It is important to denote `num(X)` as the global variable (outside the braces), and `part(K)` as the local variable (inside the braces). We want the choice rule to reflect that for each of the defined numbers, it should partition it in k possible ways. the choice rule bound guarantees that we get a unique partition for each number.

For example, let $n = 3$ and $k = 2$, then for each number $x \in \{1, 2, 3\}$, we either derive $set(x, 1)$ or $set(x, 2)$. Thus the total search space becomes k^n. Then, we model the constraint that excludes all answer sets that are **not** sum-free. In other words, any two elements of a partition summed together must not be an element of the partition.

```
:- set(X, K), set(Y, K), set(Z, K), Z = X + Y.
```

4.2 Graphs

A graph G consists of two sets, (V, E), where V is the set of vertices (nodes), and E is the set of edges denoted as a subset of the Cartesian product $V \times V$. Graph structures are often expressed as combinatorial problems. For example, path planning problems where constraints are evident.

4.2.1 Example: Strongly connected components

A set of vertices V is a strongly connected component if there is a path between every two vertices in V. For example, consider the directed graph in figure 4.1.

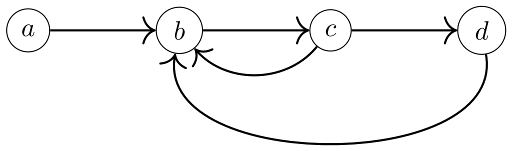


Figure 4.1: Example directed graph.

Here, we see that $\{b, c, d\}$ is a strongly connected component, because there is a directed path between every two elements of this set. To encode this problem in ASP, we need to first define the vertices and edges.

```
node(a; b; c; d).
edge(a, b; b, c; c, b; c, d; d, b).
```

Then, we need to define the search space using a choice rule.

```
1 { in(X) : node(X) }.
```

The lower bound is set to 1, so that the empty set is excluded. this rule effectively generates the powerset minus the empty set of V. The next step is to define the notion of a "path". That is, a directed path between X and Y is a finite sequence of edges that joins a sequence of vertices. Therefore, a recursive definition can be made with the base case stating that an edge between X and Y is itself a path between X and Y. The recursive case states that there is a path between X and Y if there is a path between X and Z and there is a path between Z and Y.

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), path(Z, Y).
```

The last step is to constrain the search space by limiting the possible stable model solutions so that they adhere to the definition of strongly connected component.

```
:- in(X), in(Y), not path(X, Y), X != Y.
```

Using the above graph example, this logic program yields the stable model solutions:

```
{in(a)}                      {in(b), in(c)}
{in(b)}                      {in(c), in(d)}
{in(c)}                      {in(b), in(d)}
{in(d)}                      {in(b), in(c), in(d)}
```

Usually, we want to find the largest strongly connected component, because it encompasses information about reachability for *all* of its vertices.

5 Aggregates and Optimisation

Aggregates significantly extend the expressive possibilities of the ASP language. They enable us to specify certain properties of sets of atoms. The aggregate `#count`$\{X : p(X)\} > 1$ describes the count of terms X for which there is a $p(X)$, and tests whether this count is larger than 1. For example, let the set of atoms be $\{p(a), p(b), p(c)\}$, then the aggregate `#count`$\{X : p(X)\} = 3$. Thus, the inequality holds.

Definition 5.1. Aggregates
An aggregate has the general form

$$AGG\{X_1, \ldots, X_n \ : \ A_1, \ldots, A_m\} \odot w_0$$

where:

- AGG is an aggregate function, e.g., `#count`.
- $X_1, \ldots, X_n$ are variables, e.g., variable X in `#count`$\{X : p(X)\} > 1$.
- $A_1, \ldots, A_m$ are atoms, e.g., atom $p(X)$ in `#count`$\{X : p(X)\} > 1$.
- $\odot$ is an mathematical operator, $\odot \in \{=, \neq, <, \leq, \geq, >\}$, e.g., $>$ in `#count`$\{X : p(X)\} > 1$.
- w_0 is a term, e.g., term 1 in `#count`$\{X : p(X)\} > 1$

5.1 Count aggregate in Clingo

The aggregate `#count` calculates the number of elements in a set. It can be used to denote **cardinality boundaries** in logic programs. For example, the expression

```
#count{X, Y : edge(X, Y)}
```

represents the number of elements of the set `edge/2`. The part of an aggregate expression to the right of the colon may be a list of several atoms, negated atoms, and/or comparisons. For example, the expression

```
#count{X, Y, Z : edge(X, Y), edge(Y, Z)}
```

denotes the number of paths of length 2.

The part of an aggregate on the left of the colon may include complex terms. For example,

```
#count{X*Y : X = 1..n, Y = 1..n}
```

denotes the number of unique products for all combinations between 1 and n.

As with choice constructs, we can have both local and global variables. Global variables in an aggregate constrain the aggregate to only count for the specific instances of the global variables. For example,

```
outdegree(X, N) :- vertex(X), N = #count{Y : edge(X, Y)}.
```

calculates, for each vertex X, its out-degrees. That is, the number of directed edges going out from X.

5.2 Sum aggregate in Clingo

The aggregate `#sum` calculates the sum of a set of integers. For example, the rule

```
q(N) :- N = #sum{X*X : p(X)}.
```

denotes the predicate `q(N)` where N is the sum of the squares derived from `p/1`. Combined with the facts `p(a; 1; 2)`, the stable model solution is `{p(a), p(1), p(2), q(5)}`. Interestingly, `p(a)` was ignored by the summation aggregate because it does not contain an integer.

If the sum aggregate is applied to an expression containing several terms to the left of the colon, then the value of the summation can be described in terms of “weights”. The *weight* of a tuple (left of the colon) is its first member. This has an important implication, because the `#sum` aggregate **first** accumulates the set, and only then sums up each weight of the set. Therefore, in the program

```
age(alice, 12; bob, 12; john, 10; sam, 9).
total_age(T) :- T = #sum{ A : age(N, A) }.
#show total_age/1.
```

the stable model is `{total_age(31)}`, which is derived from the sum $12 + 10 + 9$. Due to the property of sets having no duplicates, we can see that 12 is only counted once. To solve this, we need to derive a set of unique tuples, instead of singletons.

```
total_age(T) :- T = #sum{ A, N : age(N, A) }.
```

Here, the stable model is `{total_age(43)}`, because for each tuple, `(A, N)`, in the derived set, its first member A is summed up.

5.3 Minimum and Maximum aggregates

The aggregates `#max` and `#min` represent the largest and the smallest elements of a set. For example, given two sets `p/1` and `q/1`, each containing integer values for their first arguments, we can compute the smallest distance between the values of the two sets.

```
#min{ |X-Y| :  p(X), q(X) }
```

Within `clingo`, the total order for evaluating comparisons is defined on numbers, symbolic constants, and two additional ground terms. These ground terms are `#inf` and `#sup`. The former is the least possible element in the total order, whereas the latter is the greatest possible element in the total order.

```
#inf   ...    -7   -6   -5   ...    5   6   7   ...    abc   ...    #sup
```

In application to the empty set, the minimum aggregate returns `#sup`, whereas the maximum aggregate returns `#inf`.

5.4 Optimisation aggregates

When there are several stable model solutions, we may be interested in finding the stable model that is good, or even the best possible model, given some measure of quality. The measure of quality can be the `#sum` aggregate that measures the quality of a model by summing the integers in a set of atoms. To find the model for which this sum is the largest or smallest is an optimisation problem.

The directives `#maximize` and `#minimize` instruct `clingo` to improve first stable model that is generated using the `#sum` aggregate as the measure of quality. `clingo` then continues looking for better and better models until the best is found.

These directives allow us to solve **combinatorial optimisation problems**, where the goal is to find the best among several alternatives. When using one of the optimisation directives, `clingo` also yields an optimisation value. This value is negative when using the `#maximize` directive, where a larger absolute value is preferred. On the other hand, this value is positive when using the `#minimize` directive, where a smaller positive value is preferred.

As an example, consider the travelling salesman problem, where we are given a set of cities and their distances to each other, and are tasked to find a path through all cities with the least amount of distance.

```
% define timesteps 1..n where n is the number of cities
n(1..4).
city(a; b; c; d).

% define kilometres between each city
edge(a, c, 20; b, c, 15; d, c, 20).
edge(b, a, 10; d, a, 25; b, d, 10).

% distance between cities is symmetric
edge(X, Y, C) :- edge(Y, X, C).

% for each timestep, generate all possible cities
{ path(C, N) : city(C) } = 1 :- n(N).

% reject answer sets where the same city occurs in different timesteps
:- path(C, N1), path(C, N2), N1 != N2.

% define cost as the sum of the distances
cost(X) :- X = #sum{D, C1, C2 : edge(C1, C2, D), path(C1, N), path(C2, N+1)}.

% find model with minimal cost
#minimize { X : cost(X) }.

#show path/2.
```

5.5 Semantics of aggregates

Based on the semantics defined by Faber et al.[5], we can formulate the semantics that will help us compute answer sets of programs that contain aggregates. Recall that grounding of aggregates is similar to the grounding of choice constructs, because local and global variables are used in the same way in aggregates, albeit with a different semantics. However, the computation of the reduct with respect to some interpretation is indeed different.

Definition 5.2. Reduct of aggregates
Given a ground program Π that consists of aggregates and an interpretation I, then Π^I is the **reduct** of Π with respect to I, where, in addition to the previous definition of reduct, the following axiom must be satisfied:

> Remove from Π the rule r if there is a literal $b \in body(r)$ that is *aggregate* and evaluates to *false* with respect to I.
>
> $$\Pi^I = \{r \mid r \in \Pi, \forall b \in body(r) \implies I \models b\}$$
>
> where $I \models b$ is *true* if the aggregate b can be satisfied by the literals in I.

As an example, consider the program Π:

$$p \leftarrow \#sum\{1 : p; -1 : q\} \geq 0.$$
$$p \leftarrow \#sum\{1 : q\} > 0.$$
$$q \leftarrow \#sum\{1 : p\} > 0.$$

To compute all answer sets of Π, we can systematically consider all possible answer sets and compute the reduct with respect to these interpretations, i.e., $\varnothing$, $\{p\}$, $\{q\}$, and $\{p, q\}$.

- $I = \varnothing$ gives the entailment of body literal with respect to I:
  - $I \models \#sum\{1 : p; -1 : q\} \geq 0$, because $\#sum(\varnothing) = 0$ and $0 \geq 0$.
  - $I \not\models \#sum\{1 : q\} > 0$, because $\#sum(\varnothing) = 0$ and $0 > 0$ is *false*.
  - $I \not\models \#sum\{1 : p\} > 0$, because $\#sum(\varnothing) = 0$ and $0 > 0$ is *false*.

  Thus, the reduct is $\Pi^I = \{p \leftarrow \#sum\{1 : p; -1 : q\} \geq 0\}$ and its minimal model is $\{p\}$, which means $I = \varnothing$ is not an answer set.

- $I = \{p\}$ gives the entailment of body literal with respect to I:
  - $I \models \#sum\{1 : p; -1 : q\} \geq 0$, because $\#sum(\{1\}) = 1$ and $1 \geq 0$.
  - $I \not\models \#sum\{1 : q\} > 0$, because $\#sum(\varnothing) = 0$ and $0 > 0$ is *false*.
  - $I \models \#sum\{1 : p\} > 0$, because $\#sum(\{1\}) = 1$ and $1 > 0$.

  Thus, the reduct is $\Pi^I = \{p \leftarrow \#sum\{1 : p; -1 : q\} \geq 0; q \leftarrow \#sum\{1 : p\} > 0\}$ and its minimal model is $\{p, q\}$, which means $I = \{p\}$ is not an answer set.

- $I = \{q\}$ gives the entailment of body literal with respect to I:

  - $I \not\models \#\mathit{sum}\{1 : p; -1 : q\} \geq 0$, because $\#\mathit{sum}(\{-1\}) = -1$ and $-1 \geq 0$ is *false*.
  - $I \models \#\mathit{sum}\{1 : q\} > 0$, because $\#\mathit{sum}(\{1\}) = 1$ and $1 > 0$.
  - $I \not\models \#\mathit{sum}\{1 : p\} > 0$, because $\#\mathit{sum}(\varnothing) = 0$ and $0 > 0$ is *false*.

  Thus, the reduct is $\Pi^I = \{p \leftarrow \#\mathit{sum}\{1 : q\} > 0\}$ and its minimal model is $\{q\}$, which means $I = \{p\}$ is not an answer set.

- $I = \{p, q\}$ gives the entailment of body literal with respect to I:
  - $I \models \#\mathit{sum}\{1 : p; -1 : q\} \geq 0$, because $\#\mathit{sum}(\{1, -1\}) = 0$ and $0 \geq 0$.
  - $I \models \#\mathit{sum}\{1 : q\} > 0$, because $\#\mathit{sum}(\{1\}) = 1$ and $1 > 0$.
  - $I \models \#\mathit{sum}\{1 : p\} > 0$, because $\#\mathit{sum}(\{1\}) = 1$ and $1 > 0$.

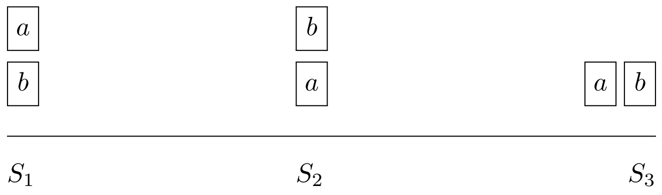
  Thus, the reduct is $\Pi^I = \Pi$ and its minimal model is $\{p, q\}$, which means $I = \{p, q\}$ is the only answer set.

6 Dynamic Domains

Answer Set Programming has important applications to the study of **dynamic systems**. These systems contain *states* that can be changed by performing *actions*. It can be used to predict or to plan. In a **prediction** problem, the task is to determine how the current state of a dynamic system will change after executing a given sequence of steps. In a **planning** problem, the task is to find the sequence of steps from a given initial state to a goal state.

6.1 Motivating example: Blocks world

The blocks world is a dynamic system of several blocks stacked on top of each other, forming towers. Each tower is based on a table. For example, given 2 blocks, a and b, then there are 3 possible configurations of states.



To represent the configuration of blocks, we can use terms of the form $on(b, l)$, where b is a block and l is a location. In other words, the term states that block b is on location l. The table itself is represented with the constant t. Thus the 3 possible configurations can be described with the following sets:

$$S_1 = \{on(a, b), on(b, t)\}$$
$$S_2 = \{on(b, a), on(a, t)\}$$
$$S_3 = \{on(a, t), on(b, t)\}$$

The action of a hypothetical robot arm moving block B to location L will be denoted by terms of the form $put(B, L)$. For example, given we are in state S_1, then action $put(a, t)$ changes the current state to S_3. The execution of a sequence of actions determines the system's trajectory.

Furthermore, to describe changes in the system, we use integers from 0 to some finite n to denote **steps** of the trajectory. We can also distinguish between **fluents**, which are properties that can be changed by actions, and **statics**, which cannot be changed. Therefore, the objects in the domain are blocks, locations, configurations, steps, actions, and fluents.

Two new predicates, $holds(fluent, step)$ and $occurs(action, step)$, can be used to describe what fluents are true and what actions occurred at any given step. Based on the above formulation of the blocks world, we can encode the relationship $holds(on(B, L), I)$, which says that block B is on location L at time step I. To denote that block B was put on location L at step I, we simply say $occurs(put(B, L), I)$.

Given a set of blocks, we can encode all possible locations (including the table location). From which, the derivation of the fluents and actions is also possible.

```
block(a; b).
```

```
location(X) :- block(X).
location(t).

#const n = 2.
step(0..n).

% "block B is on location L" is a property that changes with time
fluent(on(B, L)) :- block(B), location(L), B != L.

% "put block B on location L" is a possible action,
% provided we do not put a block onto itself.
action(put(B, L)) :- block(B), location(L), B != L.
```

To further describe the initial configuration of, say S_1, we can expand the program with the following *holds* relationship and derive the negation of *holds* by using the closed world assumption.

```
holds(on(a, b), 0).
holds(on(b, t), 0).

-holds(on(B, L), 0) :- block(B), location(L),
                       not holds(on(B, L), 0).
```

Now that the complete initial configuration is specified, we can define the blocks world theory, i.e., the effect of actions. Since each action takes one step, the following rule describes the effect of action $put(B, L)$.

```
holds(on(B, L), I+1) :- occurs(put(B, L), I), I < n.
```

This rule can be viewed as a special case of a **causal law**, which is a statement of the form "a **causes** f **if** $p_0, \ldots, p_m$", that says that action a executed in a state of the domain satisfying conditions $p_0, \ldots, p_m$ causes fluent f to become true in the resulting state.

The action $put(B, L)$ has a direct effect on the state of the blocks world, but also an **indirect** effect, as block B no longer is on top of its previous location. Furthermore, the fact that a block B is on L means that no other block B' can be on L at time step I. We can encode both rules with the following.

```
% No block occupies more than one location
-holds(on(B, L2), I) :- holds(on(B, L1), I),
                        location(L2),
                        L1 != L2.

% No block can support more than one block directly on top
-holds(on(B2, B), I) :- block(B),
                        holds(on(B1, B), I),
                        block(B2),
                        B1 != B2.
```

If we expand the program with the fact $occurs(put(a, t), 0)$, then the query $holds(on(a, t), 1)$ yields "yes". However, the query $holds(on(b, t), 1)$ yields "maybe". This is because it is not explicitly derived that the predicate holds in both time step 0 and 1. Therefore, the reasoning agent must be taught that if there is lack of evidence to the contrary, then assume that *normally things stay as they are*. This is a form of **default** statement, which is also known as the **Inertia Axiom**. This axiom can be expressed by two rules:

```
holds(F, I+1) :- holds(F, I),
                  not -holds(F, I+1),
                  I < n.

-holds(F, I+1) :- -holds(F, I),
                  not holds(F, I+1),
                  I < n.
```

The rules state that without explicit evidence to the contrary, the value of fluent F remains constant at step $I+1$.

Actions which should not be allowed still result in consistent answer sets. To remedy this, we need to place restrictions on the executability of actions. This way, the restrictions may lead to contradictions, and thus no solution should be found.

```
% cannot put block B on L if another block B1 is already on B.
-occurs(put(B, L), I) :- location(L),
                         holds(on(B1, B), I).

% cannot put a block on a block that is already occupied.
-occurs(put(B1, B), I) :- block(B1),
                          block(B),
                          holds(on(B2, B), I).
```

6.2 Transition diagram

To generalise the notion of a dynamic system, we can model the system as a **transition diagram**. A transition diagram is a directed graph whose nodes correspond to physically possible states of the domain and whose arcs are labeled by actions. Such models are called *Markovian*.

A transition $\langle \sigma_0, \{a_0, \dots, a_k\}, \sigma_1 \rangle$ denotes that if the set of actions $\{a_0, \dots, a_k\}$ were executed simultaneously while in state σ_0, then the result may be the transition to state σ_1.

A path $\langle \sigma_0, a_0, \sigma_1, \dots, a_{n-1}, \sigma_n \rangle$ of the diagram represents a possible trajectory of the system with initial state σ_0 and final state σ_n. The transition diagram of a dynamic system contains all possible trajectories of that system.

Recall the causal effect of actions that can be described in the form:

$$a \textbf{ causes } f \textbf{ if } p_0, \dots, p_m.$$

The law says that *action a, executed in a state satisfying conditions $p_0, \dots, p_m$, causes fluent f to become true in the resulting state.*

Consider a simple example of a dynamic system, in which the states are described by two Boolean inertial fluents, f and g, and their arcs are labeled by one action a whose effect is described by a single causal law, a **causes** $\neg f$ **if** f. We denote this description of our system as $\mathcal{D}_0$. Common sense tells us that if a is executed in state $\sigma_0 = \{f, g\}$, the new successor state will contain $\neg f$ implied by the causal law. Note that g will remain true due to the *inertia axiom*. Figure 6.1 shows the transition diagram for $\mathcal{D}_0$.

Causal and other relations between fluents can be described by **state constraints**, i.e., statements of the form

$$f \textbf{ if } p_0, \dots, p_m.$$

These state constraints say that *every satisfying condition $p_0, \dots, p_m$ must also satisfy f.*

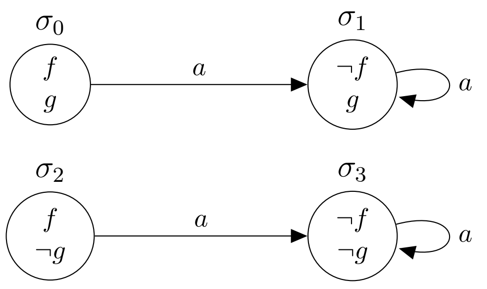


Figure 6.1: Transition diagram of system $\mathcal{D}_0$, where $\mathcal{F} = \{f, g\}$ and $\mathcal{A} = \{a\}$. The only causal law in this system is a **causes** $\neg f$ **if** f.

They are used to define **indirect effects** of actions. Finding concise ways of defining these effects is called the **Ramification Problem**.

In the blocks world example, the state constraints are denoted in lines 21-29. Note that state constraints are not dependent on actions.

To illustrate the idea of **state constraints**, let us expand $\mathcal{D}_0$ by adding an inertial fluent h and a state constraint $\neg h$ **if** $\neg f$. Figure 6.2 contains the transition diagram of $\mathcal{D}_1$. As a result, the state σ_1 contains the direct effect $\neg f$ due to a derived by the causal law, g derived by inertia, and indirect effect $\neg h$ due to a derived by the new state constraint.

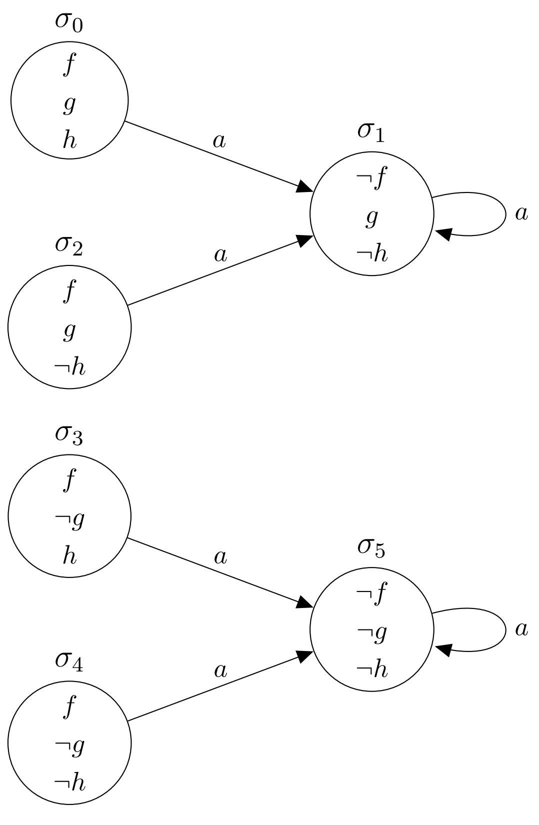


Figure 6.2: Transition diagram of system $\mathcal{D}_1$, where $\mathcal{F} = \{f, g, h\}$ and $\mathcal{A} = \{a\}$. The only causal law in this system is a **causes** $\neg f$ **if** f. The only state constraint in this system is $\neg h$ **if** $\neg f$.

Executability conditions are represented by laws of the form

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p_0, \ldots, p_m$$

that say that *it is impossible to execute actions* $a_0, \ldots, a_k$ *simultaneously in a state satisfying conditions* $p_0, \ldots, p_m$.

To illustrate this in the running example, let $\mathcal{D}_2$ denote the expansion of $\mathcal{D}_1$ with executability condition **impossible** a **if** $\neg f$, which says that it is impossible to perform action a in any state that contains fluent $\neg f$. The transition diagram of $\mathcal{D}_2$ has the exact same form as $\mathcal{D}_1$, except that the cycles in σ_0 and σ_5 have been eliminated.

Descriptions $\mathcal{D}_0$, $\mathcal{D}_1$, and $\mathcal{D}_2$ can be viewed as theories in action language $\mathcal{AL}$. **Action languages** are formal models of parts of natural language used for describing the behaviour of dynamic systems. Another way to view them is as tools for describing transition diagrams.

6.3 Action Language Syntax

An Action language $\mathcal{AL}$ is parameterised by a *sorted signature* containing three special sorts: *statics*, *fluents*, and *actions*. The fluents are further partitioned into two sorts: *inertial* and *defined*. We refer to both statics and fluents as **domain properties**. A **domain literal** is a domain property p or its negation $\neg p$. If domain literal l is formed by a fluent, we refer to it as a **fluent literal**; otherwise it is a **static literal**.

A set S of domain literals is called **complete** if for any domain property p either p or $\neg p$ is in S. Furthermore, S is called **consistent** if there is no p such that $p \in S$ and $\neg p \in S$.

Definition 6.1. Statements of $\mathcal{AL}$.
Language $\mathcal{AL}$ allows the following types of statements:

1. *Casual Laws*:

$$a \textbf{ causes } l_{in} \textbf{ if } p_0, \ldots, p_m.$$

2. *State Constraints*:

$$l \textbf{ if } p_0, \ldots, p_m.$$

3. *Executability Conditions*:

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p_0, \ldots, p_m$$

where a is an action, l is an arbitrary domain literal, l_{in} is a literal formed by an inertial fluent, $p_0, \ldots, p_m$ are domain literals, $k \geq 0$, and $m \geq -1$ (where keyword **if** is omitted if $m = -1$).

Definition 6.2. System Description.
A **System Description** of $\mathcal{AL}$ is a collection of statements of $\mathcal{AL}$.

6.4 Action Language Semantics

A system description $\mathcal{SD}$ serves as a *specification* of the transition diagram $\mathcal{T}(\mathcal{SD})$, which defines all possible trajectories of the corresponding dynamic system. Therefore, to define the semantics of $\mathcal{AL}$, we have to precisely define the states and legal transitions of this diagram.

6.4.1 States

If $\mathcal{SD}$ does not contain **defined fluents** in its sorted signature, then the definition of a state is simple. A state is a *complete* and *consistent* set of domain literals satisfying state constraints of $\mathcal{SD}$. Recall the examples of descriptions $\mathcal{D}_0$, $\mathcal{D}_1$, and $\mathcal{D}_2$, which used this definition of a state.

For system descriptions *with* defined fluents, the situation is more complex. To see why, consider the following example. Let $\mathcal{D}_3$ be a system description with two inertial fluents, f and g, and a single defined fluent h, which is defined by the following rules:

$$h \textbf{ if } f$$
$$h \textbf{ if } \neg g$$

Clearly, $\sigma_0 = \{f, g, h\}$ is a state of $\mathcal{D}_3$, because the body of the first rule is satisfied by $f \in \sigma_0$ and the head is satisfied by $h \in \sigma_0$, and the body of the second rule is not satisfied, $\neg g \notin \sigma_0$, which means the rule **is** satisfied regardless. Following the same logic of satisfiability, $\sigma_f = \{f, g, \neg h\}$ is **not** a state of $\mathcal{D}_3$, because even though the body of the first rule is satisfied, $f \in \sigma_f$, the head is not satisfied, $h \notin \sigma_f$, making the whole rule unsatisfiable with the given state set.

So far so good. The definition holds up. However, what about $\{\neg f, g, h\}$? Based on the definition: *"A state is a complete and consistent set of domain literals satisfying state constraints of* $\mathcal{SD}$*"*, the proposed state $\{\neg f, g, h\}$ is indeed a state of $\mathcal{D}_3$. However, we see that the truth value of h cannot be derived by the rules of the state constraints. Therefore, the suggested definition does not work, and we need a better definition of **state**.

Definition 6.3. Translation to ASP: state constraints and defined fluents. By $\Pi_c(\mathcal{SD})$ (where c stands for constraints), we denote the logic program defined as follows:

1. For every **state constraint**
$$l \textbf{ if } p$$
$\Pi_c(\mathcal{SD})$ contains
$$l \leftarrow p.$$

2. For every **defined fluent** f, $\Pi_c(\mathcal{SD})$ contains the CWA:
$$\neg f \leftarrow \textit{ not } f.$$

Let σ_{nd} denote the collection of all domain literals formed by **inertial fluents** and **statics**. The $_{nd}$ stands for *nondefined*. With this, we can properly define what a **state** of a transition diagram is.

Definition 6.4. Formal definition of State.
A *complete* and *consistent* set σ of domain literals is a **state** of the transition diagram defined by a system description $\mathcal{SD}$ if σ is a unique answer set of program

$$\Pi_c(\mathcal{SD}) \cup \sigma_{nd}.$$

In other words, a **state** is a *complete* and *consistent* set of literals σ that is the **unique answer set** of the program that consists of:

- the **nondefined literals** from σ (inertial fluents and statics);
- the **encoding** of the **state constraints** (in ASP); and
- the **CWA** for each **defined fluent** (in ASP).

If we consider the above example $\mathcal{D}_3$ again, we can transform the defined fluent and state constraints into the program $\Pi_c(\mathcal{SD})$:

$$h \leftarrow f.$$
$$h \leftarrow \neg g.$$
$$\neg h \leftarrow \ not\ h.$$

It is now easy to check that the transition diagram defined by $\mathcal{D}_3$ has states: $\{f, g, h\}$, $\{f, \neg g, h\}$, $\{\neg f, \neg g, h\}$, $\{\neg f, g, \neg h\}$.

To check whether $\sigma_0 = \{f, \neg g, h\}$ is a state of $\mathcal{D}_3$, it is sufficient to check that σ_0 is the only answer set of $\Pi_c(\mathcal{SD}) \cup \{f, \neg g\}$. Conversely, to see that $\sigma_f = \{\neg f, g, h\}$ is not a state of $\mathcal{D}_3$, it is sufficient to see that σ_f is not the unique answer set of $\Pi_c(\mathcal{SD}) \cup \{\neg f, g\}$.

6.4.2 Transitions

Like the formulation of states in ASP, we can define the transition relation of $\mathcal{T}(\mathcal{SD})$ in terms of answer sets of a logic program. To describe a transition $\langle \sigma_0, a, \sigma_1 \rangle$, we construct a program $\Pi(\mathcal{SD}, \sigma_0, a)$ consisting of logic programming encodings of the system description $\mathcal{SD}$, initial state σ_0, and set of actions a, such that answer sets of this program determine the states that the system can move into after execution of a in state σ_0.

Definition 6.5. Encoding of System Description.
The encoding $\Pi(\mathcal{SD})$ of system description $\mathcal{SD}$ consists of the encoding of the signature of $\mathcal{SD}$ and rules obtained from statements of $\mathcal{SD}$.

Encoding of the Signature We start with the encoding of the signature $sig(\mathcal{SD})$.

- For each **constant symbol** c of some sort $sort\_name$ other than sorts $fluent$, $static$, or $action$, $sig(\mathcal{SD})$ contains:
$$sort\_name(c).$$

- For every **static** g of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains:

$$static(g).$$

- For every **inertial fluent** f of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains:

$$fluent(inertial, f).$$

- For every **defined fluent** f of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains:

$$fluent(defined, f).$$

- For every **action** a of $\mathcal{SD}$, $sig(\mathcal{SD})$ contains:

$$action(a).$$

Encoding of Statements of $\mathcal{SD}$ For this encoding, we only need two (time) steps, 0 and 1, which stand for the beginning and the end of a transition. Using a constant n for the maximum number of steps, we define:

$$\begin{aligned}&\#const\ n = 1.\\&step(0..n).\end{aligned}$$

We also introduce a relation $holds(f, i)$ that says that fluent f is true at step i. Likewise, we also need relation $occurs(a, i)$ that says that action a occurred at step i.

We use this notation to encode statements of $\mathcal{SD}$ as follows:

- For every **causal law**

  $$a \textbf{ causes } l \textbf{ if } p_0, \ldots, p_m.$$

  $\Pi(\mathcal{SD})$ contains:

  $$\begin{aligned}holds(l, I+1) \leftarrow & holds(p_0, I), \ldots, holds(p_m, I),\\ & occurs(a, I),\\ & I < n.\end{aligned}$$

- For every **state constraint**:

  $$l \textbf{ if } p_0, \ldots, p_m.$$

  $\Pi(\mathcal{SD})$ contains:

  $$holds(l, I) \leftarrow holds(p_0, I), \ldots, holds(p_m, I).$$

- For every **defined fluent**, $\Pi(\mathcal{SD})$ contains the CWA:

  $$\begin{aligned}\neg holds(F, I) \leftarrow & fluent(defined, F),\\ & not\ holds(F, I).\end{aligned}$$

- For every **executability condition**:

$$\textbf{impossible } a_0, \dots, a_k \textbf{ if } p_0, \dots, p_m$$

$\Pi(\mathcal{SD})$ contains:

$$\neg occurs(a_0, I) \textit{ or } \dots \textit{ or } \neg occurs(a_k, I) \leftarrow holds(p_0, I), \dots, holds(p_m, I).$$

- Furthermore, $\Pi(\mathcal{SD})$ contains the **inertia axiom**:

$$\begin{aligned} holds(F, I+1) \leftarrow & fluent(inertial, F),\\ & holds(F, I),\\ & \textit{not } \neg holds(F, I+1),\\ & I < n. \end{aligned}$$

$$\begin{aligned} \neg holds(F, I+1) \leftarrow & fluent(inertial, F),\\ & \neg holds(F, I),\\ & \textit{not } holds(F, I+1),\\ & I < n. \end{aligned}$$

- For every **action**, $\Pi(\mathcal{SD})$ contains the CWA:

$$\neg occurs(A, I) \leftarrow \textit{ not } occurs(A, I).$$

This completes the construction of encoding $\Pi(\mathcal{SD})$ of system description $\mathcal{SD}$.

The two remaining parts of $\Pi(\mathcal{SD}, \sigma_0, a)$ are the encoding of the initial state σ_0 and the encoding of action a.

$$holds(\sigma_0, 0) \overset{\text{def}}{=} \{holds(l, 0) : l \in \sigma_0\}$$

and

$$occurs(a, 0) \overset{\text{def}}{=} \{occurs(a_i, 0) : a_i \in a\}.$$

Definition 6.6. System encoding
To complete program $\Pi(\mathcal{SD}, \sigma_0, a)$, we gather the description of the system's laws, the initial state, and the actions that occur in it:

$$\Pi(\mathcal{SD}, \sigma_0, a) \overset{\text{def}}{=} \Pi(\mathcal{SD}) \cup holds(\sigma_0, 0) \cup occurs(a, 0).$$

Definition 6.7. Formal definition of Transition
Let a be a non-empty collection of actions and σ_0 and σ_1 be states of the transition diagram $\mathcal{T}(\mathcal{SD})$ defined by a system desciption $\mathcal{SD}$. A state-action-state triplet $\langle \sigma_0, a, \sigma_1 \rangle$ is a **transition** of $\mathcal{T}(\mathcal{SD})$ **iff** $\Pi(\mathcal{SD}, \sigma_0, a)$ has an answer set A such that $\sigma_1 = \{l : holds(l, 1) \in A\}$.

In other words, given the encoding of the system description $\Pi(\mathcal{SD})$, the initial state σ_0, and the actions a, as a logic program, we say that it is a **transition** if the program yields an answer set from which the state σ_1 can be derived.

6.4.3 Example: The Briefcase Domain

As an example, consider a briefcase which has 2 clasps. We denote an action, *toggle*, which moves a given clasp into the up position if it is down, and in the down position if it is up. If both clasps are in the up position, then the briefcase is *open*; otherwise it is *closed*.

The signature of the briefcase domain consists of a sort $clasp = \{1, 2\}$, inertial fluent $up(C)$ that holds **iff** clasp C is up, defined fluent *open* that holds **iff** both clasps are up, and the action $toggle(C)$ that toggles clasp C. The system description for this domain $\mathcal{D}_{bc}$ consists of the causal laws and state constraint:

$$\begin{aligned}
&toggle(C) \textbf{ causes } up(C) \textbf{ if } \neg up(C)\\
&toggle(C) \textbf{ causes } \neg up(C) \textbf{ if } up(C)\\
&open \textbf{ if } up(1), up(2)
\end{aligned}$$

Here, the variable C ranges over the sort *clasp*. Strictly speaking, since these laws contain variables, they are not proper statements of our action language. Such laws are often referred to as **schemas**. However, simply grounding the laws will yield a proper representation of the briefcase domain.

To figure out what the states of our domain are, we need a program $\Pi(\mathcal{D}_{bc})$ that consists of the state constraints and the CWA of the defined fluents.

$$\begin{aligned}
&open \leftarrow up(1), up(2).\\
&\neg open \leftarrow \; not\ open.
\end{aligned}$$

Now we can check whether arbitrary sets of domain literals are considered states of $\mathcal{D}_{bc}$. For example, consider

$$\sigma = \{\neg up(1), up(2), \neg open\}.$$

First, we need to check whether it is complete and consistent, which by definition it is. Next, we need to consider the nondefined inertial fluents and statics $\sigma_{nd} = \{\neg up(1), up(2)\}$, and check if σ is the only answer set of the program $\Pi(\mathcal{D}_{bc}) \cup \sigma_{nd}$. Clearly, it is, because we cannot derive *open*, and due to the CWA, we indeed can derive $\neg open$.

Now, let $\sigma = \{\neg up(1), up(2), open\}$. According to the description of the briefcase, this should not be a possible state, as the briefcase can only be open if *both* clasps are up. Although the set is complete and consistent, we see that it is not the answer set to the program consisting of $\Pi(\mathcal{D}_{bc})$ and facts $\neg up(1)$ and $up(2)$. Therefore, σ is not a state.

To define the transitions of the briefcase system, we can use the definitions of translating the action language into a logic program. From there, it is trivial to reason about the transitions of the system. First, we encode the signature of our system with its respective sorts, fluents, and actions.

```
% sort "clasp" over {1, 2}
```

```
clasp(1; 2).

% inertial fluent over sort clasp and defined fluent open
fluent(inertial, up(C)) :- clasp(C).
fluent(defined, open).

% toggle action over sort clasp
action(toggle(C)) :- clasp(C).

% time step
#const n = 1.
step(0..n).
```

Next, we translate our axioms. The first two axioms describe causal laws in our system. Encoding these yields the following.

```
% causal law: toggle(C) causes up(C) if -up(C)
holds(up(C), I+1) :- occurs(toggle(C), I),
                     -holds(up(C), I),
                     I < n.

% causal law: toggle(C) causes -up(C) if up(C)
-holds(up(C), I+1) :- occurs(toggle(C), I),
                      holds(up(C), I),
                      I < n.
```

The last law of $\mathcal{D}_{bc}$ is a state constraint, which is encoded in the following.

```
% state constraint: open if up(1), up(2)
holds(open, I) :- holds(up(1), I), holds(up(2), I).
```

The last three constructs are not dependent on the axioms from $\mathcal{D}_{bc}$. These are the CWA for defined fluents, the inertia axiom, and the CWA for actions. Accordingly, they are translated as follows.

```
% CWA for defined fluents
-holds(F, I) :- fluent(defined, F),
                not holds(F, I),
                step(I).

% inertia axiom
holds(F, I+1) :- fluent(inertial, F),
                 holds(F, I),
                 not -holds(F, I+1),
                 I < n.

-holds(F, I+1) :- fluent(inertial, F),
                  -holds(F, I),
                  not holds(F, I+1),
                  I < n.

% CWA for actions
-occurs(A, I) :- action(A), step(I),
                 not occurs(A, I).
```

This concludes the encoding of the briefcase domain. Now, we can use program $\Pi(\mathcal{D}_{bc})$ together with an initial state and a set of actions, to derive the resulting state. For example, let $\Pi(\sigma, a)$ be the program with initial state $\sigma = \{\neg up(1), up(2), \neg open\}$ and $a = \{toggle(1)\}$, which can be encoded as follows:

```
% initial state
-holds(up(1), 0).
holds(up(2), 0).
-holds(open, 0).

% action
occurs(toggle(1), 0).

% display
#show holds/2.
#show -holds/2.
```

The answer set of $\Pi(\mathcal{D}_{bc}) \cup \Pi(\sigma, a)$, denoted by A is:

```
A = {
    -holds(up(1),0), holds(up(2),0), -holds(open,0),
    holds(up(1),1), holds(up(2),1), holds(open,1),
}
```

Using the definition of transition, the resulting state σ_1 is obtained by examining the literals in A at step 1, i.e., $\sigma_1 = \{l : holds(l, 1) \in A\}$. This results in $\sigma_1 = \{up(1), up(2), open\}$.

6.5 Planning

In **classical planning**, a typical problem is defined as follows:

- A **goal** is a set of fluent literals that the agent wants to become true.
- A **plan** for achieving a goal is a sequence of agent actions that takes the system from the current state to one that satisfies this goal.
- **Problem**: Given a description of a deterministic dynamic system, its current state, and a goal, find a plan to achieve this goal.

A sequence α of actions is called a **solution** to a classical planning problem if the problem's goal becomes true at the end of the execution of α. When the agent has a limit on the length of the allowed plans, this limit is often referred to as the **horizon** of the planning problem.

To solve a classical planning problem $\mathcal{P}$ with horizon n, we construct a program $plan(\mathcal{P}, n)$ such that solutions of $\mathcal{P}$ whose length do not exceed n correspond to answer sets of $plan(\mathcal{P}, n)$. The program consists of 4 parts:

- ASP encodings of the system description
- The current state
- The goal state
- Domain-independent ASP program called the **simple planning module**.

The encoding of the system description and the current state are identical to the encodings described in the previous section. The encoding for the goal state can be defined by a new relation $goal(I)$ that holds if and only if all fluent literals from the problem's goal G are satisfied at step I of the system's trajectory. This relation can be defined by the rule

```
goal(I) :- holds(f_1, I), ..., holds(f_m, I),
               -holds(g_1, I), ..., -holds(g_n, I)
```

where $G = \{f_1, \ldots, f_m\} \cup \{g_1, \ldots, g_n\}$.

We can now define the domain-independent **simple planning module** of the program. The first two rules define the success of the search for a plan:

```
% if goal was reached, derive success
success :- goal(I),
           I <= n.

% only keep successful answer sets solutions
:- not success.
```

The first rule defines **success** as the existence of a step in the system's trajectory that satisfies the **goal**. The second is a constraint that states that failure to succeed is unacceptable.

The second part of the planning module is responsible for generating sequences of actions of appropriate length that may be possible desired plans. If the consequence of these sequences is *success*, then a plan is found.

```
% for each step, choose an action such that the goal is not (yet) reached
{occurs(A, I) : action(A)} = 1 :- step(I), not goal(I), I < n.
```

This choice construct guarantees that every sequence of actions contains exactly one occurrence of an action at each step prior to the goal being achieved, and **no** occurrences of actions after the goal has been achieved.

To find one of the shortest plans, we can optionally add a `#minimize` directive that minimises the amount of actions in the answer sets.

```
% minimize amount of actions to find shortest sequence
#minimize{1, I : occurs(A, I)}.
```

7 Diagnostic Agents

In dynamic domains, certain actions can have unexpected outcomes. For instance, the light switch was toggled but the light bulb is not on, or a car that does not start even though the key is inserted and properly rotated. In such cases, it is important to diagnose the behaviour. The light bulb could be broken or the car battery could be empty. Using dynamic models as logic programs, we can automatically generate diagnostics of unexpected behaviour within the dynamic domain.

To build agents capable of finding explanations of unexpected observations, we can divide actions of our domain into two *disjoint* classes: **agent actions** and **exogenous actions**. The former are performed by the agent associated with the domain, whereas the latter are performed by nature or by other agents (within a multi-agent system). The two main assumptions here are:

1. The agent is capable of making *correct observations*, *performing actions*, and *recording these observations and actions.*

2. *Normally*, the agent is capable of observing all relevant exogenous actions occuring in its environment.

Note that the second assumption is *defeasible*, i.e., some exogenous actions can remain **unobserved**. A typical **diagnostic problem** is informally specified as follows:

- A **symptom** consists of a recorded history of the system such that its last collection of observations is *unexpected*, i.e., it contradicts the agent's expectations.

- An **explanation** of a symptom is a collection of unobserved past occurrences of exogenous actions that may account for the unexpected observations. This idea of explanation is closely tied to the second assumption. Although the agent's ability to perform, observe, and consider causal laws, is **not** defeasible, the **completeness** of its observations of exogenous actions is defeasible. Accordingly, the agent realises that it may miss some of the exogenous actions. A collection of missing occurrences of exogenous actions that may account for the discrepancy is therefore viewed as an explanation.

- **Diagnostic problem**: Given a description of a dynamic system and a symptom, find a possible explanation of the symptom.

7.1 Recording the History of a Domain

In addition to a description of the transition diagram that represents possible trajectories of the system, we assume that the knowledge base of a **diagnostic agent** contains the system's **recorded history**. This history consists of **observations made by the agent** together with a record of its **own actions**.

The recorded history defines a collection of paths in the transition diagram that can be interpreted as the system's possible pasts with respect to the agent perceiving the past. If the agent's knowledge is **complete**, i.e., contains complete information about the initial state and the occurrences of actions, and the system's actions are **deterministic**, then there is only one such path.

Definition 7.1. Syntax of recorded history.
The recorded history Γ_{n-1} of a system up to a current step n is a collection of **observations** that come in one of the following forms:

1. $obs(f, true, i)$, fluent f was observed to be true at step i; or

2. $obs(f, false, i)$, fluent f was observed to be false at step i; or

3. $hpd(a, i)$, action a was performed by the agent or observed to happen at step i

where i is an integer from the interval $[0, n)$.

Definition 7.2. Semantics of recorded history.
A path $\langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ in the transition diagram $\mathcal{T}(\mathcal{SD})$ is a **model of a recorded history** Γ_{n-1} of dynamic system $\mathcal{SD}$ if for any $0 \leq i \leq n$:

1. $a_i = \{a : hpd(a, i) \in \Gamma_{n-1}\}$;

2. **if** $obs(f, true, i) \in \Gamma_{n-1}$ **then** $f \in \sigma_i$;

3. **if** $obs(f, false, i) \in \Gamma_{n-1}$ **then** $\neg f \in \sigma_i$.

We say that Γ_{n-1} is **consistent** if it has a model.

As an example, let us revisit the brief case domain. The system description, $\mathcal{D}_{bc}$, contains the agent's knowledge about the domain.

$$\begin{aligned}
&toggle(C) \textbf{ causes } up(C) \textbf{ if } \neg up(C)\\
&toggle(C) \textbf{ causes } \neg up(C) \textbf{ if } up(C)\\
&open \textbf{ if } up(1), up(2)
\end{aligned}$$

Now suppose that initially clasp 1 was fastened, i.e., $\neg up(1)$, and the agent unfastened it. The corresponding history is

$$\Gamma_0 \begin{cases} obs(up(1), false, 0). \\ hpd(toggle(1), 0). \end{cases}$$

There are two models of Γ_0 that satisfy this history. In both models, the action a_0 is $toggle(1)$, but the states σ_0 and σ_1 are different. Therefore, path $\langle \sigma_0, toggle(1), \sigma_1 \rangle$ is a model of Γ_0 if:

$$M_1 \begin{cases} \sigma_0 = \{\neg up(1), \neg up(2), \neg open\} \\ \sigma_1 = \{up(1), \neg up(2), \neg open\} \end{cases} \qquad M_2 \begin{cases} \sigma_0 = \{\neg up(1), up(2), \neg open\} \\ \sigma_1 = \{up(1), up(2), open\} \end{cases}$$

Since Γ_0 has models, this means that it is **consistent** with respect to the transition diagram for the briefcase domain. Although we have a **consistent history**, our knowledge is **not complete**. There are two possible trajectories consistent with the agent's recorded history, i.e., M_1 and M_2. At the end of one trajectory the briefcase is open (M_2), and at the end of the other trajectory the briefcase is closed (M_1). An intelligent agent would need more information to come up with a conclusion about the state of the briefcase.

Definition 7.3. Entailment.

- A fluent literal l **holds** in a model M of Γ_{n-1} at step $i \leq n$, which is denoted by $M \models h(l,i)$, **if** $l \in \sigma_i$;
- Γ_{n_1} **entails** $h(l,i)$, denoted by $\Gamma_{n-1} \models h(l,i)$, **if** for every model M of Γ_{n-1}, the consequence $M \models h(l,i)$ **holds**.

Using the above definition of entailment, we see that $\Gamma_0 \models \mathit{holds}(\mathit{up}(1), 1)$, because in every model of Γ_0, its consequence is the literal $\mathit{holds}(\mathit{up}(1), 1)$.

Consider another example where

$$\Gamma_0 \begin{cases} \mathit{obs}(\mathit{up}(1), \mathit{true}, 0) \\ \mathit{obs}(\mathit{up}(2), \mathit{true}, 0) \\ \mathit{hpd}(\mathit{toggle}(1), 0) \\ \mathit{obs}(\mathit{open}, \mathit{true}, 1) \end{cases}$$

This history is **not consistent** with the transition diagram, because it has no model. There is no path in the transition diagram that we can follow in this situation. It makes sense that the briefcase can only be *open* if both clasps are up, but the first clasp must be down at step 1 due to the *toggle* action.

7.2 Defining Explanations

Consider an agent currently at step n. The agent observes the values of a collection of fluents at step n. Let us denote these observations by O^n. The pair $\mathcal{C} = \langle \Gamma_{n-1}, O^n \rangle$ is often referred to as the **system's configuration**. If new observations are **consistent** with the agent's view of the world, i.e., if $\mathcal{C}$ is consistent, then O^n simply becomes part of the recorded history. Otherwise, the agent needs to start seeking the **explanation of the inconsistency**.

The only possible explanation for unexpected observations would be that some **exogenous action** occurred that the agent did not observe.

Definition 7.4. Possible Explanation.

- A configuration $\mathcal{C} = \langle \Gamma_{n-1}, O^n \rangle$ is called a **symptom** if it is **inconsistent**, i.e., it has no model.
- A **possible explanation** of a symptom $\mathcal{C}$ is a set $\mathcal{E}$ of statements $\mathit{occurs}(a,i)$ where a is an exogenous action, $0 \leq i \leq n$, and $\mathcal{C} \cup \mathcal{E}$ is **consistent**.

As an example, consider the following simple story: *Mixing baking soda with lemon juice produces foam, unless the baking soda is stale. Joanna mixed baking soda with lemon juice, but there was no foam as a result.* In this example, Joanna acts as the agent and we can distinguish between an agent action ("mixing baking soda and lemon juice") and an exogenous action ("baking soda went stale"). Note that the exogenous action is a natural phenomenon and out of the agent's control. This system, denoted by $\mathcal{D}_{mix}$, is described as follows using $\mathcal{AL}$:

Ingredients baking soda (bs) and lemon juice (lj):

$$\begin{aligned}&ingredient(bs)\\&ingredient(lj)\end{aligned}$$

Fluents:

$$\begin{aligned}&fluent(inertial, mixed(bs, lj))\\&fluent(inertial, stale(bs))\\&fluent(defined, foam(bs, lj))\end{aligned}$$

Actions:

$$\begin{aligned}&action(agent, mix)\\&action(exogenous, wentStale).\end{aligned}$$

The **causal laws** and **state constraints** describing the **normal behaviour** of the system:

$$\begin{aligned}&mix \textbf{ causes } mixed(bs, lj)\\&foam(bs, lj) \textbf{ if } mixed(bs, lj), \neg stale(bs)\end{aligned}$$

The **causal law** for the **exogenous action** is the following:

$$wentStale \textbf{ causes } stale(bs)$$

Next consider a possible history of the system:

$$\Gamma_0 \begin{cases} hpd(mix, 0)\\ obs(stale(bs), false, 0)\\ obs(mixed(bs, lj), false, 0)\\ obs(foam(bs, lj), false, 0)\end{cases}$$

It is trivial to see that the path $\langle \sigma_0, mix, \sigma_1 \rangle$ where

$$\begin{aligned}\sigma_0 &= \{\neg stale(bs), \neg mixed(bs, lj), \neg foam(bs, lj)\}\\ \sigma_1 &= \{\neg stale(bs), mixed(bs, lj), foam(bs, lj)\}\end{aligned}$$

is the only model of Γ_0, and therefore, $\Gamma_0 \models h(foam(bs, lj), 1)$. In other words, the agent (Joanna) expects there to be foam.

Now, to illustrate the notion of possible explanations, assume that the agent observes that there is no foam after the *mix* action. In diagnostic jargon, it means that the configuration

$$\mathcal{C} = \langle \Gamma_0, obs(foam(bs, lj), false, 1)\rangle$$

is a **symptom**. The possible explanation for the symptom is

$$\mathcal{E} = \{occurs(wentStale, 0)\}$$

In other words, there was an **unobserved exogenous action** *wentStale* that occurred at step 0, that can explain why there is no foam after the *mix* action.

It is possible, and in fact probable, that the system is much more complex. Therefore, there may be many, possibly irrelevant, exogenous actions in our system. If we were to expand the baking system with another exogenous action *makeCoffee*, there would be two explanations:

$$\mathcal{E}_1 = \{occurs(wentStale, 0)\}$$
$$\mathcal{E}_2 = \{occurs(wentStale, 0), occurs(makeCoffee, 0)\}$$

Notice that we are more interested in the explanation that includes *wentStale* rather than the one that includes *makeCoffee*. However, it may be difficult to dismiss the *makeCoffee* action, as it could be relevant in the grand(er) scheme of things. Due to the *butterfly effect*, the act of making coffee might causally lead to the baking soda being stale.

No rational agent is interested in finding all possible explanations of a symptom. Rather, we want the best possible explanation based on some quality criteria. The criteria could be an ordering relation between explanations. For instance,

- $\mathcal{E}_1$ can be viewed to be **better** than $\mathcal{E}_2$ if $\mathcal{E}_1 \subset \mathcal{E}_2$; or
- $\mathcal{E}_1$ can be viewed to be **better** than $\mathcal{E}_2$ if $|\mathcal{E}_1| < |\mathcal{E}_2|$; or
- $\mathcal{E}_1$ can be viewed to be **better** than $\mathcal{E}_2$ if $\mathcal{E}_1$ contains actions that are more likely to occur or more relevant to the symptom than those in $\mathcal{E}_2$.

Definition 7.5. Best Explanation.
Let $\mathcal{C}$ be a symptom and $<$ be a **partial linear order** defined on possible explanations of $\mathcal{C}$, where $\mathcal{E}_1 < \mathcal{E}_2$ is read as $\mathcal{E}_1$ is **better** than $\mathcal{E}_2$. Then, possible explanation $\mathcal{E}$ of $\mathcal{C}$ is called a **best explanation** with respect to $<$ if there is no possible explanation $\mathcal{E}_0$ of $\mathcal{C}$ such that $\mathcal{E}_0 < \mathcal{E}$.

7.3 Computing Explanations

Consider a system with configuration

$$\mathcal{C} = \langle \Gamma_{n-1}, O^n\rangle \,.$$

The agent associated with the system just performed its nth action and observed the values of some fluents. Now the agent needs to check that this configuration is **consistent with**

the expectations, i.e., that $\mathcal{C}$ is not a symptom. To achieve this task, we can construct a logic program `all_clear.lp` that takes the encoding of the system description, $\Pi(\mathcal{SD})$, the current configuration $\mathcal{C}$, and the following axioms:

$$holds(F,0) \text{ or } \neg holds(F,0) \leftarrow fluent(inertial,F). \tag{1}$$
$$occurs(A,I) \leftarrow hpd(A,I). \tag{2}$$
$$\leftarrow obs(F,true,I), \neg holds(F,I). \tag{3}$$
$$\leftarrow obs(F,false,I), holds(F,I). \tag{4}$$

Awareness Axiom (1) The first axiom guarantees that the agent takes into consideration all of the fluents in the system. Recall that under the answer set semantics, this is not a tautology. Rather, the disjunction is epistemic, and the statement simply asserts that, during construction of the agent's beliefs, the value of the fluent cannot be **unknown**. In other words, any answer set of the program must contain either $holds(F,0)$ or $\neg holds(F,0)$.

Simulation Axiom (2) The second axiom establishes the relationship between *occurs* and *hpd*. The latter is used to record actions that actually happened, whereas the former holds even if the corresponding action is only **hypothetical**. Therefore, *occurs* is a superset of *hpd*. The rule ensures that actions that actually were observed (happened) are taken into consideration, i.e., simulated.

Reality Check Axiom (3), (4) The last two rules guarantee that the agent's *expectations* agree with its *observations*. If they do not agree, we want the program to realise that there is an inconsistency between what the agent believes should logically be true versus what it has observed.

Definition 7.6. Symptom Checking.
A configuration $\mathcal{C}$ is a symptom if and only if program `all_clear`$(\mathcal{SD},\mathcal{C})$ is inconsistent, i.e., has no answer set solution.

As an example, again consider the briefcase domain. Let $\mathcal{C}_0 = \langle \Gamma_0, obs(open, true, 1)\rangle$ be a configuration, where

$$\Gamma_0 \begin{cases} obs(up(1), false, 0) \\ obs(up(2), true, 0) \\ hpd(toggle(1), 0) \end{cases}$$

It is trivial to check whether $all\_clear(\mathcal{D}_{bc}, \mathcal{C}_0)$ is consistent. We see that the action $toggle(1)$ ensures that both clasps are up in step 1, which means the observation that the briefcase is open at step 1 is true. In other words, the reality check axiom fails to reject the answer set, which makes the resulting answer set consistent.

If $\mathcal{C}_1 = \langle \Gamma_0, obs(open, false, 1)\rangle$, then $all\_clear(\mathcal{D}_{bc}, \mathcal{C}_1)$ is **inconsistent**, and as expected $\mathcal{C}_1$ is a **symptom**. The reality check axiom rejects all answer sets because of the inconsistency

$$\leftarrow obs(open, false, 1), holds(open, 1).$$

Building upon the program that checks whether there is a symptom in a system, we can expand by also deriving all possible **explanations** for the symptom. Again, this problem can be reduced to computing answer sets of an ASP program. The program, called `diagnose.lp` consists of

- the rules of $all\_clear(\mathcal{SD}, \mathcal{C})$
- the **explanation generation** rule

$$\begin{aligned} occurs(A, I) \text{ or } \neg occurs(A, I) \leftarrow & action(exogenous, A), \\ & step(I), \\ & I < n. \end{aligned}$$

  or alternatively:

$$\{occurs(A, I) : action(exogenous, A)\} \leftarrow step(I), I < n.$$

- and the following rule which defines a new relation $expl(A, I)$. The relation holds if and only if an exogenous action A is hypothesised to occur at I, but there is no record of this occurrence in the agent's history.

$$\begin{aligned} expl(A, I) \leftarrow & action(exogenous, A), \\ & occurs(A, I), \\ & not\ hpd(A, I). \end{aligned}$$

To comply to the symbolic formality, a set $\mathcal{E}$ is a possible explanation of a symptom $\mathcal{C}$ iff there is an answer set A of $diagnose(\mathcal{SD}, \mathcal{C})$ such that

$$\mathcal{E} = \{occur(a, i) : expl(a, i) \in A\}.$$

7.3.1 Example: Foam baking system

To apply the above methods in a concrete example, again consider the domain where an agent, Joanna, can act upon the environment by deciding to mix baking soda and lemon juice to create foam. Here, the exogenous action is the baking soda that might become stale, which prevents the mixture from creating foam. Starting off, we can translate the system description, $\mathcal{D}_{mix}$, into ASP. Let's denote this program by `baking_system.lp`, depicted in the following listing together with the *all_clear* and *diagnose* programs.

```
ingredient(bs; lj).

fluent(inertial, mixed(bs, lj)).
fluent(inertial, stale(bs)).
fluent(defined, foam(bs, lj)).

action(agent, mix).
action(exogenous, went_stale).

#const n = 1.
```

```
step(0..n).

% causal law: mix causes mixed(bs, lj)
holds(mixed(bs, lj), I+1) :- occurs(mix, I).

% causal law: went_stale causes stale(bs)
holds(stale(bs), I+1) :- occurs(went_stale, I).

% state constraint: foam(bs, lj) if mixed(bs, lj), -stale(bs)
holds(foam(bs, lj), I) :- holds(mixed(bs, lj), I), -holds(stale(bs), I).

% inertia axiom
holds(F, I+1) :- holds(F, I), not -holds(F, I+1), I < n.
-holds(F, I+1) :- -holds(F, I), not holds(F, I+1), I < n.

% CWA for actions
-occurs(A, I) :- action(_, A), step(I), not occurs(A, I).

% CWA for defined fluents
-holds(F, I) :- fluent(defined, F), step(I), not holds(F, I).
```

Listing 1: `baking_system.lp`

```
holds(F, 0) | -holds(F, 0) :- fluent(inertial, F).
occurs(A, I) :- hpd(A, I).
:- obs(F, true, I), -holds(F, I).
:- obs(F, false, I), holds(F, I).
```

Listing 2: `all_clear.lp`

```
% Explanation generation rule
{ occurs(A, I): action(exogenous, A) } :- step(I), I < n.

% if exogenous action is predicted to occur,
% but there is no historical evidence of the action occurring
expl(A, I) :- action(exogenous, A),
              occurs(A, I),
              not hpd(A, I).
```

Listing 3: `diagnose.lp`

Now that everything is set up, we can examine a configuration that is inconsistent, and thus a symptom, to see what the possible explanations are. Let Γ_0 be the history consisting of

$$\Gamma_0 \begin{cases} obs(mixed(bs, lj), false, 0). \\ obs(foam(bs, lj), false, 0). \\ hpd(mix, 0). \end{cases}$$

The configuration $\mathcal{C} = \langle \Gamma_0, obs(foam(bs, lj), false, 1). \rangle$ is denoted in the program `configuration.lp`.

```
obs(mixed(bs, lj), false, 0).
obs(foam(bs, lj), false, 0).
hpd(mix, 0).
obs(foam(bs, lj), false, 1).
#show expl/2.
```

```
#show holds/2.
#show -holds/2.
```

Listing 4: `configuration.lp`

Running Clingo with all four programs in the command

```
clingo 0 diagnose.lp baking_system.lp all_clear.lp configuration.lp
```

yields 3 answer sets of possible explanations.

1. The first explanation states that at step 0, $\neg stale(bs)$ was true, and the exogenous action *wentStale* at step 0 caused $stale(bs)$ to be true in step 1. This indirectly leads to there being no foam after mixing.

2. The second explanation states that at step 0, $stale(bs)$ was true, and due to inertia, stayed true in step 1.

3. The third explanation is similar to the second explanation with the addition that the exogenous action *wentStale* still occurred at step 0. This action had no effect as the baking soda was already stale.

To find the minimal explanation, we can utilise Clingo's `#minimize` directive.

```
#minimize{ 1, A, I : occurs(A, I), action(exogenous, A) }.
```

8 Grounding and Solving

The procedure of solving is often computationally demanding. Therefore, it is important to design efficient algorithms for solvers. The procedure can be partitioned in two parts. First, a **grounding** part ensures that the variables in the logic program are grounded. Thereafter, the ground program is used in the **solving** part to search for answer sets.

8.1 Satisfiability solving

The problem of satisfiability can be formulated as assigning truth values to atoms such that they **satisfy** a set of propositional formulas. This is an example of an NP-problem. Such problems are often hard to solve, but solutions are easy to check. The Davis-Putnam procedure forms the basis of satisfiability solvers. This procedure finds models of propositional formulas by traversing a tree of all possible truth assignments for variables in the formula. Every path in the tree satisfying the formula is a possible model, and thus, a possible solution to the formula.

The efficiency of this algorithm depends on the ordering of variables and the efficiency of testing if a vector $\overline{X}$ of variables falsifies a propositional formula. For example, consider the formula $F \vee (X_1 \vee \neg X_1)$, where F is an arbitrary formula. A model can be found quickly if we start by assigning a value to X_1, and then realise that this assignment already satisfies the formula due to the disjunction. Then, arbitrarily assigning values to the remaining variables would produce a model. By starting with the assignment of X_1 and realising the effect of the disjunction, we effectively pruned the search through the tree.

Numerous papers describe the ordering, data structures, and algorithms to implement the Davis-Putnam procedure efficiently. In what follows, a basic algorithm will be described. But first, some terminology:

- A **signature** is a set of propositional variables.
- A **clause** over signature Σ is a set $\{l_1, \dots, l_n\}$ of literals of Σ denoting the disjunction $l_1 \vee \cdots \vee l_n$.
- A **formula** is a set $\{C_1, \dots, C_m\}$ of clauses denoting the conjunction $C_1 \wedge \cdots \wedge C_m$.
- A **partial interpretation** is a mapping of a set of propositional variables from Σ into truth values. We identify a partial interpretation I with the set of literals made true by I. For any variable p from the domain of I, if $p \in I$, we say that p is *true* in I; if $p \notin I$ then p is false in I.
- An **interpretation** is a partial interpretation defined on all variables of the language.
- A **model** of a formula F is an interpretation that makes the formula *true*.
- A formula is **satisfiable** if it has a model.
- Partial interpretation I_2 is **compatible** with partial interpretation I_1 if $I_1 \subseteq I_2$, i.e., I_1 is a subset of I_2.
- Variable p is **undefined** in partial interpretation I if it does not belong to the domain of I.

Using this terminology, we can define an algorithm $Sat(F)$ that takes a formula F as an input and returns a model of F if F is satisfiable. The algorithm is defined recursively by the function $Sat(I, F)$ that searches for a model of formula F over signature Σ *compatible* with partial interpretation I. Calling this function with arguments $Sat(\varnothing, F)$ will find a compatible superset of $\varnothing$ that satisfies F.

Algorithm 1 *SAT* Satisfiability solving function

```
    **Input**: partial interpretation $I_0$ and formula $F_0$;
    **Output**: a pair $\langle I, true\rangle$ where $I$ is a model of $F_0$ compatible with $I_0$;
            $\langle I_0, false\rangle$ if no such model exists.
 1: **function** $Sat(I_0,\ F_0)$
 2:     $F := F_0$
 3:     $I := I_0$
 4:     $\langle F, I, X\rangle := Cons(F, I)$
 5:     **if** $X = false$ **then**
 6:         **return** $\langle I_0, false\rangle$
 7:     **if** $F = \varnothing$ **then**
 8:         **return** $\langle I, true\rangle$
 9:     select variable $p$ undefined in $I$
10:     $\langle I, X\rangle := Sat(I, F \cup \{p\})$
11:     **if** $X = true$ **then**
12:         **return** $\langle I, true\rangle$
13:     **return** $Sat(I, F \cup \{\overline{p}\})$
14: **end function**
```

The *Sat* function calls the *Cons* function, which computes **consequences** of F in I, adds these consequences to I, and uses them to simplify F. If no contradiction is derived, the *Cons* function returns *true* and an updated F and I; otherwise it returns false and the original formula and interpretation.

Notice that there are two termination conditions in the algorithm. The first, on lines 5-6 returns *false* if *Cons* found a contradiction. The second, on lines 7-8 checks for success. If $F = \varnothing$ then every element of F is trivially satisfiable and so is F, therefore, the function returns *true* with the new interpretation I. If neither condition holds, *Sat* makes a nondeterministic choice of a yet undefined variable p of F and calls itself recursively. Depending on the choice for p, the algorithm might make efficient and rational optimisations.

The implementation of *Cons*, called **unit propagation**, is denoted in algorithm 2. Note the condition on line 8 that checks whether there is an empty clause in the current formula, $\{\} \in F$. If there is, then F is unsatisfiable. To better understand this notion, note that to satisfy a clause, we need to satisfy at least one element in it. This property is related to the disjunction operator yielding truth when either or both of its operands yield truth. Hence, the empty clause is unsatisfiable, and so is the collection F of clauses containing the empty clause. Also notice that each call to *Cons* eliminates occurrences of at least one literal from F, and hence *Sat* will eventually terminate.

Algorithm 2 Unit Propagation *Cons* sub-routine of *Sat*

```
    Input: partial interpretation $I_0$ and formula $F_0$ with signature $\Sigma_0$;
    Output: $\langle F, I, true\rangle$ where $I$ is a partial interpretation such that $I_0 \subseteq I$;
            $\langle F_0, I_0, false\rangle$ if no such model exists.
1:  function $Cons(F_0,\ I_0)$
2:      $F := F_0$
3:      $I := I_0$
4:      while $F$ contains a unary clause $\{l\}$ do
5:          remove from $F$ all clauses containing $l$
6:          remove from $F$ all occurrences of $\bar{l}$
7:          $I := I \cup \{l\}$
8:      if $\{\} \in F$ then
9:          return $\langle F_0, I_0, false\rangle$
10:     return $\langle F, I, true\rangle$
11: end function
```

As an example, let us compute $Sat(I_0, F_0)$, where

$$I_0 = \varnothing$$

and

$$\begin{aligned} F_0 &= (p \vee q) \wedge (\neg p \vee \neg q) \wedge (p \vee s \vee r) \\ &= \{\{p, q\}, \{\neg p, \neg q\}, \{p, s, r\}\} \end{aligned}$$

We call $Sat(\varnothing, F_0)$, and immediately call $Cons(F, \varnothing)$ for the first time, which has no effect on the output formula and interpretation, due to the fact that there is no unary literal in F. Thus, we return to *Sat* and rely on the nondeterministic choice to include an undefined variable. We choose p and call $Sat(\varnothing, F \cup \{p\})$.

In the second call to *Cons*, we arrive in the while loop and remove all clauses containing the variable p. In addition, we remove all occurrences of $\overline{p}$. This results in

$$\begin{aligned} F &= \{\{\neg q\}\} \\ I &= \{p\} \end{aligned}$$

In the next iteration, we consider the only unary literal left in F, that is, $\neg q$. Likewise, we remove it from F and add it to I. Note that the condition on line 8 of *Cons* is not met, even though $F = \varnothing$. The condition explicitly checks for an empty set in the set F, which is different from F itself being the empty set. Line 10 now returns *true* with:

$$\begin{aligned} F &= \varnothing \\ I &= \{p, \neg q\} \end{aligned}$$

Control now returns back to *Sat* where the condition for success on line 7 is met. Thus, $I = \{p, \neg q\}$ is a partial interpretation satisfying the formula F. Note that I is a partial interpretation, and hence, is not a model for F. To make it a model, we simply assign arbitrary values to variables that are not in I, i.e., variables s and r.

8.2 Answer set solving

Algorithms for computing answer sets of logic programs contain two steps. In the first step, the algorithm replaces a program Π, which might contain variables, by its ground instantiations $ground(\Pi)$. Practically, $ground(\Pi)$ is not the full set of syntactically constructible instantiations. Rather, it is limited to a subset that has the same answer sets as Π. In Clingo, it is therefore important to express **safe variables**, so that the grounding process can drastically affect the performance of the entire system. Grounding techniques are quite sophisticated, borrowing techniques from deductive databases.

Once the variables have been eliminated from Π, the solving step is performed by a **Solver**. The structure of the *Solver* is very similar to that of the *Sat*. However, it works on slightly different objects with slightly different definitions of interpretation, consistency, and so on. The corresponding terminology is as follows:

- By **program** we mean a ground logic program without $\neg$, *or*, and constraints.
- By **extended literal**, or simply **e-literal**, over signature Σ, we mean a literal of Σ possibly preceded by default negation *not*. By $\textit{not}\ l$, we denote $\textit{not}\ p(\bar{t})$ if $l = p(\bar{t})$, and $p(\bar{t})$ if $l = \textit{not}\ p(\bar{t})$. Similar to the notion of $q = \neg\neg q$.
- A set of literals is called **consistent** if it contains no e-literals of the form l and $\textit{not}\ l$.
- A **partial interpretation** of Σ is a *consistent* set of ground e-literals of Σ. If a partial interpretation I is complete, i.e., for any atom p of Σ either $p \in I$ or $\textit{not}\ p \in I$, then I is called an interpretation.
- An atom p can be **true** ($p \in I$), **false** ($\textit{not}\ p \in I$), or **undefined** ($p \notin I$, $\textit{not}\ p \notin I$), with respect to a partial interpretation I.
- An answer set A of a program Π will be represented as an interpretation I of the signature of Π such that

$$I = \{p(\bar{t}) : p(\bar{t}) \in A\} \cup \{\ \textit{not}\ p(\bar{t}) : p(\bar{t}) \notin A\}.$$

- A set A of ground atoms is called **compatible** with partial interpretation I if for every ground atom $p(\bar{t})$, if $p(\bar{t}) \in I$ then $p(\bar{t}) \in A$, and if $\textit{not}\ p(\bar{t}) \in I$, then $p(\bar{t}) \notin A$. A ground program Π is **compatible** with I if it has an answer set compatible with I.

Consider algorithm 3, in which a simple **ASP-solver** procedure is denoted. The *Solver* starts by initialising variables Π and I and calling a function $ConsASP(\Pi, I)$. This subroutine adds to I a collection of e-literals that can be inferred from Π and I, and uses it to simplify Π. If no contradiction is inferred in the process, $ConsASP$ returns the new partial interpretation I and the simplified program, together with a boolean value *true*. Otherwise, it returns the original input together with *false*.

Algorithm 3 Simple Answer Set Solver

```
    Input: partial interpretation $I_0$ and program $\Pi_0$;
    Output: $\langle I, true \rangle$ where $I$ is an answer set of $\Pi_0$ compatible with $I_0$;
            $\langle I_0, false \rangle$ if no such answer set exists.
1:  function $Solver(I_0, \Pi_0)$
2:      $\Pi := \Pi_0$
3:      $I := I_0$
4:      $\langle \Pi, I, X \rangle := ConsASP(\Pi, I)$
5:      if $X = false$ then
6:          return $\langle I_0, false \rangle$
7:      if no atom is undefined in $I$ then
8:          if $IsAnswerSet(I, \Pi_0)$ then
9:              return $\langle I, true \rangle$
10:         return $\langle I_0, false \rangle$
11:     select a ground atom $p$ undefined in $I$
12:     $\langle I, X \rangle := Solver(I \cup \{p\}, \Pi)$
13:     if $X = true$ then
14:         return $\langle I, true \rangle$
15:     return $Solver((I \backslash \{p\}) \cup \{not\,p\}, \Pi)$          ▷ Take $I$ without $p$ but with $not\,p$.
16: end function
```

As with *Sat*, there are two termination conditions following the call to *ConsASP*. First, in lines 5-6, *Solver* checks if *ConsASP* returned *false*. If so, I_0 is **inconsistent** with Π_0 and *Solver* returns *false*. Second, in line 7, a check for **completeness** is made, i.e., for every atom p either p or *not p* belongs to I. If this is the case, *Solver* checks whether I is an answer set, and returns *true* together with I if this is the case; otherwise, there is no answer set of Π_0 compatible with I_0 and the function returns *false*.

If there are still some undefined atoms, then *Solver* selects such an atom p and *Solver* is called again to explore whether I can be **expanded** to an answer set of Π containing p. If this is impossible, then *Solver* searches for an answer set of Π containing *not p*. If one of these calls succeeds, then the function stops and returns *true* together with I. Otherwise, the function returns *false*, because I cannot be expanded to any answer set. To call the function, we start with an empty set for I, thus, $Solver(\varnothing, \Pi)$.

8.2.1 Lower Bound Consequence Function

A simple version of the *ConsASP* function is the *LB* function, where *LB* stands for **lower bound**. The *LB* function computes consequences of its parameters, Π and I, using the following four **inference rules**:

1. **If** the body of a rule is a subset of I, **then** its head must be in I.
2. **If** atom $p \in I$ belongs to the head of exactly one rule of Π, **then** the e-literals from the body of this rule must be in I.
3. **If** $(not\,p_0) \in I, (p_0 \leftarrow B_1, p, B_2) \in \Pi$, and $B_1, B_2 \subseteq I$, **then** $not\,p$ must be in I.

4. **If** Π contains no rule with head p_0, **then** $not\, p_0$ must be in I.

We can denote the first three inference rules with an index $1 \leq i \leq 3$. Furthermore, let Π be a program, I be a partial interpretation, and r be a rule of Π. The set of i-consequences of Π, I, and r, denoted by $i\text{-}cons(i, \Pi, I, r)$, is defined as follows:

> If the *if part* of i is satisfied by Π, I, and r, then $i\text{-}cons(i, \Pi, I, r)$ is the set of e-literals from the *i's then part*.
>
> Otherwise, $i\text{-}cons(i, \Pi, I, r) = \varnothing$.

If $i = 4$, then $i\text{-}cons(i, \Pi, I)$ is the set of literals that do not occur in the heads of rules of Π and are not falisfied by I, i.e., literal must not occur in any head of Π and must not occur in I, otherwise I is inconsistent. The function $LB(\Pi, I)$ can be computed as follows

Algorithm 4 Lower Bound Consequence Function, aliased by $ConsASP$.

```
    Input: partial interpretation $I_0$ and $\Pi_0$ with signature $\Sigma_0$;
    Output: $\langle \Pi, I, true \rangle$ where $I$ is a partial interpretation such that $I_0 \subseteq I$
                and $\Pi$ is a program with signature $\Sigma_0$;
            $\langle \Pi_0, I_0, false \rangle$ if there is no answer set compatible with $I_0$.
1:  function $LB(\Pi_0, I_0)$
2:      $\Pi := \Pi_0$
3:      $I := I_0$
4:      repeat
5:          $T := I$
6:          remove from $\Pi$ all rules whose bodies are falsified by $I$
7:          remove from the bodies of the rules of $\Pi$ all e-literals, $not\ l$, satisfied by $I$
8:          select an inference rule $i$ from $[1 - 4]$
9:          if $1 \leq i \leq 3$ then
10:             for every $r \in \Pi$ satisfied by the if part of inference rule $i$ do
11:                 $I := I \cup i\text{-}cons(i, \Pi, I, r)$
12:         else
13:             $I := I \cup i\text{-}cons(4, \Pi, I)$
14:     until $I = T$
15:     if $I$ is consistent then
16:         return $\langle \Pi, I, true \rangle$
17:     return $\langle \Pi_0, I_0, false \rangle$
18: end function
```

Notice the term **falsified** in the simplification rule on line 6. We say that a set B of e-literals *falsifies* a set C of e-literals if C contains an e-literal complementary to some e-literal of B [6]. For example, the set $\{a, b\}$ falsifies the set $\{not\, b, c\}$, and the set $\{not\, c\}$ also falsifies the set $\{not\, b, c\}$. This is used to simplify programs by removing rules where the body of the rule can never satisfy the partial interpretation, i.e., the body of the rule falsifies I.

8.2.2 Example: Tracing LB

Consider the program P_1:

$$p(a) \leftarrow \; not\; q(a).$$
$$p(b) \leftarrow \; not\; q(b).$$
$$q(a).$$

Starting off with $LB(\varnothing, P_1)$, we see that initially I and T are set to $\varnothing$, and Π is set to P_1. The first rule, on line 6, does not change Π, because there are no e-literals in I that are the complement of e-literals in the bodies of Π. The second rule, on line 7, also does not change Π, because there are no *default negated* e-literals in I. Next, an inference rule is nondeterministically chosen. Suppose inference rule (1) is selected. The bodies of the first two rules are not satisfied by I (because $I = \varnothing$). The body of the third rule is empty, and hence, is satisfied by any set of e-literals, including the empty set. Note that the rule ($q(a).$) is short for ($q(a) \leftarrow .$). Thus applying inference rule (1) to the program produces one consequence, $q(a)$, which is added to I.

In the next iteration, the simplification rule on line 6 removes the first rule of Π, because $\{not\, q(a)\}$ falsifies I, where $I = \{q(a)\}$. Therefore we can delete the entire first rule from Π. The program Π now consists of

$$p(b) \leftarrow \; not\; q(b).$$
$$q(a).$$

The next simplification rule, on line 7, does not change Π any further. Thus, we can choose an inference rule again and apply it. Suppose this time, we choose inference rule (4). Since neither $p(a)$ nor $q(b)$ belongs to the heads of the rules of the simplified program, I must become $\{q(a),\; not\; p(a),\; not\; q(b)\}$.

On the third iteration, the simplification rule on line 6 has no effect on Π. Rather, the simplification rule on line 7 *does* have an effect. Based on this rule, we remove all e-literals from the bodies of rules that are default negated **and** satisfied by I. Therefore, the extended literal $not\; q(b)$ is removed from the body of the rule. The simplified program is now

$$p(b).$$
$$q(a).$$

LB now selects inference rule (1) from which the resulting partial interpretation $I = \{q(a),\; not\; p(a),\; not\; q(b), p(b)\}$ can be derived. Since the next iteration does not produce any *consequences*, the repeat-until condition $I = T$ is met, and LB returns $\langle \Pi, I, true \rangle$.

8.2.3 Definition of *IsAnswerSet*

In algorithm 3, the $IsAnswerSet$ sub-routine was expressed. Hereafter follows an elaboration of the $IsAnswerSet(I, \Pi)$.

Algorithm 5 Check whether I is an answer set of Π.

```
    Input: interpretation $I$ and $\Pi$;
    Output: true if $I$ is an answer set of $Pi$; false otherwise;
function $IsAnswerSet(I, \Pi)$
    Compute the reduct, $\Pi^I$, of $\Pi$ with respect to $I$
    Compute the answer set, $A$, of $\Pi^I$
    Check whether $A = atoms(I)$ and return the result
end function
```

The first and last step of this algorithm are straightforward. The second step, however, requires some elaboration. A program Π is called **definite** if Π is a collection of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_n$$

where ps are atoms of the signature of Π. In other words, Π contains **no default negation**.

Furthermore, let Π be a definite program and T_Π be the **immediate consequence operator** defined on sets of atoms from the signature of Π as follows:

$$T_\Pi = \{p_0 : p_0 \leftarrow p_1, \ldots, p_n \in \Pi, p_1, \ldots, p_n \subseteq A\}.$$

$T_\Pi(A)$ returns conclusions of all rules of Π whose bodies are satisfied by A. The operator is used to describe a function $Least(\Pi)$, which takes as a parameter a definite program and returns its answer set. By definition, the reduct is definite, and therefore, this function can be used to find its answer sets.

Algorithm 6 Find answer set of definite program Π.

```
    Input: a definite program $\Pi$;
    Output: the answer set of $\Pi$;
function $Least(\Pi)$
    $X := \varnothing$
    repeat
        $X_0 := X$
        $X := T_\Pi(X)$
    until $X = X_0$
    return $X$
end function
```

As an example of the T_Π operator, consider the program

$$p.$$
$$q \leftarrow p.$$
$$r \leftarrow p, q.$$

Its answer set is obtained as a result of the following computation.

$$T_\Pi(\varnothing) \implies T_\Pi(\{p\}) \implies T_\Pi(\{p, q\}) \implies T_\Pi(\{p, q, r\}) \implies \{p, q, r\}$$

8.2.4 Example: Tracing Solver

As an example of the entire procedure of solving, consider the program Π_0

$$p \leftarrow \; not \; q.$$
$$q \leftarrow \; not \; p.$$

for which we want to compute its answer sets. $Solver(\varnothing, \Pi_0)$ initialises I and Π and calls $LB(\Pi, \varnothing)$. The program Π can neither be simplified by falsification nor by removing default negated e-literals from the bodies of Π. Furthermore, there are no inference rules applicable to the program. Hence, LB returns *true* without changing I and Π.

Since $I = \varnothing$, no atom is yet defined, and *Solver* starts the process of selecting an undefined atom. Suppose *Solver* selects p and makes the recursive call $Solver(\{p\}, \Pi)$, which in turn, calls $LB(\Pi, \{p\})$.

Notice that now the body of the second rule falsifies I, i.e., $\{ \; not \; p\}$ falsifies $\{p\}$. Therefore, this rule is removed from the program resulting in the new program Π

$$p \leftarrow \; not \; q.$$

Now suppose LB selects inference rule (2), which is applicable because the head of the rule contains an atom from I. Thus, $\; not \; q$ must be in I, and $I = \{p, \; not \; q\}$. In the next iteration, we can remove the e-literals of the form $\; not \; l$ that satisfy I, thus the program becomes:

$$p \leftarrow .$$

LB returns the new I and Π together with *true*. Thereafter, *Solver* checks if no atom is undefined and checks if I is an answer set of the simplified program Π. $IsAnswerSet$ computes the answer set of the reduct and compares this with I. The result is *true*, because

$$T_\Pi(\varnothing) \implies T_\Pi(\{p\}) \implies \{p\}$$

Thus, $\{p\}$ is an answer set of Π_0. Note that a different choice for the selected e-literal would lead to finding another answer set of the program, i.e., $\{q\}$.

8.3 Grounding

The goal of **grounding** is to produce a finite propositional representation of a first-order program. The following material is adapted from chapter 4 of [7]. Consider the program Π

$$\begin{aligned}&p(a).\\&q(b).\\&r(X,Y) \leftarrow p(X), q(Y).\end{aligned}$$

The signature of Π consists of predicate symbols $p/1$, $q/1$, and $r/2$, constant symbols $a/0$ and $b/0$, and variable symbols X and Y. To construct the ground instantiation $ground(\Pi)$, we need to systematically consider all replacements of the two variables by constants a or b. Thus $ground(\Pi)$ becomes

$$\underline{p(a).}$$
$$\underline{q(b).}$$
$$r(a,a) \leftarrow p(a), q(a).$$
$$\underline{r(a,b) \leftarrow p(a), q(b).}$$
$$r(b,a) \leftarrow p(b), q(a).$$
$$r(b,b) \leftarrow p(b), q(b).$$

However, notice that only one instantiation is *not redundant*. That is, the instantiations with an underline. The other instantiations can be discarded.

This example shows the importance of efficient grounding algorithms, so that computing redundant ground instantiations can be prevented.

8.3.1 Basic Grounding Algorithm

For the computation of finite ground programs, the concept of **safety** plays a crucial role. A normal rule is **safe** if each of its variables also occur in some **positive body literal**. Accordingly, a *normal* logic program is safe if all its rules are safe. If there are no function symbols with a non-zero arity, then such a program is guaranteed to have an equivalent finite ground program. In contrast, programs with function symbols with n-arity, $n > 0$, might lead to the grounder being stuck in an infinite loop, where the function symbol is recursively substituted, e.g., $\{p(a).,p(f(X)) \leftarrow p(X)\}$ will never terminate the grounding procedure.

Grounding is in essence the construction of a ground program by **substituting** variables for ground terms.

Definition 8.1. Substitution
A substitution is a set of functions

$$\theta = \{X_1 \mapsto t_1, \dots, X_n \mapsto t_n\}$$

where $n > 0$, and X_i is a variable and t_i is a ground term. A substitution can be **applied** to a literal l, expressed as $l\theta$, by replacing every X_i by t_i.

A substitution can be applied to a set of atoms B by substituting every atom $b \in B$, and thus systematically replacing every X_i by t_i in b.

$$B\theta = \bigcup_{b \in B} \{b\theta\}$$

As an example, consider the set of atoms $B = \{p(X,Y), q(Z), r(X,Z)\}$ and the substitution $\theta = \{X \mapsto 2, Y \mapsto 4, Z \mapsto 1\}$, then $B\theta = \{p(2,4), q(1), r(2,1)\}$.

Using the idea of substitution, and given two sets B and D of atoms, we can define that a substitution θ is a **match** of B in D, if $B\theta \subseteq D$. A *good match* is an **inclusion-minimal match**, because it deals with variables occurring in B only.

For example, let $B = \{p(X)\}$ and $D = \{p(1), p(2), p(3)\}$, then the matches

$$\{X \mapsto 1\}$$
$$\{X \mapsto 2\}$$
$$\{X \mapsto 3\}$$

are considered good matches. In contrast, the match $\{X \mapsto 1, Y \mapsto 2\}$ is not a good match, as the variable Y does not occur in B.

Definition 8.2. Matches
Given a set B of (first-order) atoms and a set D of ground terms, we can define the set $\Theta(B, D)$ of **good matches** for all elements of B and D as:

$$\Theta(B, D) = \{\theta \,|\, \theta \text{ is a } \subseteq\text{-minimal match of } B \text{ in } D\}$$

This definition is motivated by the fact that a **safe rule** is ground once all of its positive body literals are ground.

Using the above definitions of **substitution** and **match**, we can define the *NaiveInstantiation* grounding algorithm.

Algorithm 7 *NaiveInstantiation* grounding algorithm

```
    Input: a safe (first-order) normal logic program $\Pi$;
    Output: a ground normal logic program $\Pi'$;
 1: function $NaiveInstantiation(\Pi)$
 2:     $D := \varnothing$
 3:     $\Pi' := \varnothing$
 4:     repeat
 5:         $D' := D$
 6:         for $r \in \Pi$ do
 7:             $B := body(r)^+$                    ▷ $B$ is set of positive body literals.
 8:             for $\theta \in \Theta(B, D)$ do
 9:                 $D := D \cup \{head(r)\theta\}$          ▷ Add to $D$ the substituted head of $r$.
10:                 $\Pi' := \Pi' \cup \{r\theta\}$          ▷ Add to $\Pi'$ the substitution applied to $r$.
11:     until $D = D'$
12:     return $\Pi'$
13: end function
```

Note that $body(r)^+$ returns the **positive literals** in the body of the rule. That is,

$$body(p \leftarrow \; not\; q, r, s, \; not\; t)^+ = \{r, s\}.$$

8.3.2 Example: Tracing grounder

As an example of the grounding procedure, consider the program Π

$$\begin{aligned}&p(a,b).\\&p(c,d).\\&q(X) \leftarrow p(X,b).\\&r(X,Y) \leftarrow q(X), p(Y,Z).\end{aligned}$$

The steps of the algorithm *NaiveInstantiation* are traced in table 2.

Table 2: Tracing the instantiations of Π

Iteration	$\Theta(B,D)$	D	Π'
1	$\{\varnothing\}$	$p(a,b)$	$p(a,b) \leftarrow .$
	$\{\varnothing\}$	$p(c,d)$	$p(c,d) \leftarrow .$
2	$\{\{X \mapsto a\}\}$	$q(a)$	$q(a) \leftarrow p(a,b).$
	$\{\{X \mapsto a, Y \mapsto a, Z \mapsto b\},$	$r(a,a)$	$r(a,a) \leftarrow q(a), p(a,b).$
	$\{X \mapsto a, Y \mapsto c, Z \mapsto d\}\}$	$r(a,c)$	$r(a,c) \leftarrow q(a), p(c,d).$

Starting off in the first iteration, the rules of Π are iterated, and we can see that the first two rules, consisting of facts have no body. The set of ground terms D is also empty at this point. Thus, the result of $\Theta(\varnothing, \varnothing) = \{\varnothing\}$ and applying the empty substitution to the head of the first two rules results in $D = \{p(a,b), p(c,d)\}$. Furthermore, the ground program Π' also consists of these two facts.

In the second iteration, we can use the ground terms in D to construct substitutions for the third and fourth rule, which do contain variables. For the third rule,

$$\Theta(\{p(X,b)\}, \{p(a,b), p(c,d)\}) = \{\{X \mapsto a\}\}.$$

Applying this substitution to the head of the rule results in $D = \{p(a,b), p(c,d), q(a)\}$ and to the rule itself modifies Π' to include the rules:

$$\begin{aligned}&p(a,b) \leftarrow .\\&p(c,d) \leftarrow .\\&q(a) \leftarrow p(a,b).\end{aligned}$$

In the last rule of Π, we assign B to be the positive body of the rule, $B = \{q(X), p(Y,Z)\}$, and use it to construct all substitutions with respect to the current set of ground terms D:

$$\Theta(B,D) = \left\{\begin{matrix}\{X \mapsto a, Y \mapsto a, Z \mapsto b\},\\ \{X \mapsto a, Y \mapsto c, Z \mapsto d\}\end{matrix}\right\}$$

Applying these substitutions yields the additional ground terms $\{r(a,a), r(a,c)\}$ and leads to the ground program Π':

$$
\begin{aligned}
&p(a,b) \leftarrow .\\
&p(c,d) \leftarrow .\\
&q(a) \leftarrow p(a,b).\\
&r(a,a) \leftarrow q(a), p(a,b).\\
&r(a,c) \leftarrow q(a), p(c,d).
\end{aligned}
$$

This concludes the grounding, as the next iteration does not modify D in any way, thus resulting in the termination of the repeat-until loop. Notice how the result is a ground program consisting of 5 rules, whereas the systematic approach would produce $2+4+4^3 = 70$ rules.

8.3.3 Semi-naive and partial evaluation

The algorithm *NaiveInstantiation* is an over-simplification of a real instantiation procedure. The reason being that in line 6, we need to re-inspect the entire program. Real implementations carefully avoid re-grounding rules by using the well-known database technique of **semi-naive evaluation**. This technique is based on the idea of *lazy evaluation*. That is, during the production of new atoms in an iteration, only rules are considered which have an atom in their body that was just instantiated in the previous iteration.

Another optimisation includes that of **partial evaluation**, where the truth value of an atom is used to simplify the grounding procedure. For example, say we encounter a fact $(p(a) \leftarrow .)$, then the truth of $p(a)$ is used to simplify in the subsequent iterations. Say the next iteration consists of the rule $(q(b) \leftarrow \ \textit{not}\ r(b), p(a).)$, then the rule is simplified to $(q(b) \leftarrow \ \textit{not}\ r(b).)$. Likewise, a rule like $(p(a) \leftarrow s(a), w(b))$ is ignored, because we already derived $p(a)$. This also holds for default negation, as in the case of $q(b) \leftarrow \ \textit{not}\ p(a)$, for which the head cannot be derived because $p(a)$ is already derived.

8.3.4 Predicate-rule dependency graph

To enable a more fine-grained static program analysis, we define the **predicate-rule dependency graph** of a logic program Π as a directed graph (V, E) where

- V is a set of predicates and rules of Π,
- $(p, r) \in E$, if predicate p occurs in the **body** of a rule, and
- $(r, p) \in E$, if predicate p occurs in the **head** of a rule.

For example, the dependency graph of the rule $r = (p \leftarrow q.)$ has an edge from predicate $q/0$ to r and from r to predicate $p/0$. Thus, for each e-literal in the body of the rule, there must be an in-coming edge from the corresponding predicate of the e-literal. And for each literal in the head of the rule, there must be an out-coming edge to the corresponding predicate of the literal.

Consider the logic program:

$$
\begin{aligned}
&r(X).\\
&p(X) \leftarrow \; not\; q(X), r(X).\\
&q(X) \leftarrow \; not\; p(X), r(X).
\end{aligned}
$$

with predicates $r/1$, $p/1$, and $q/1$ and its predicate-rule dependency graph in figure 8.1. Rectangles denote rules and circles denote predicates. After capturing the dependencies of the logic program, we can use the notion of **strongly connected components** to partition the logic program. strongly connected components are sub-graphs where every node can be visited from every other node by walking on a directed path.

A graph with strongly connected components can be seen as a **directed acyclic graph**, which is useful because it maintains a **topological order**. This order can be followed to ground the program, which results in a smaller ground program. In figure 8.1, the strongly connected components are denoted by a dashed boxes around vertices with a corresponding numbering for the topological order.

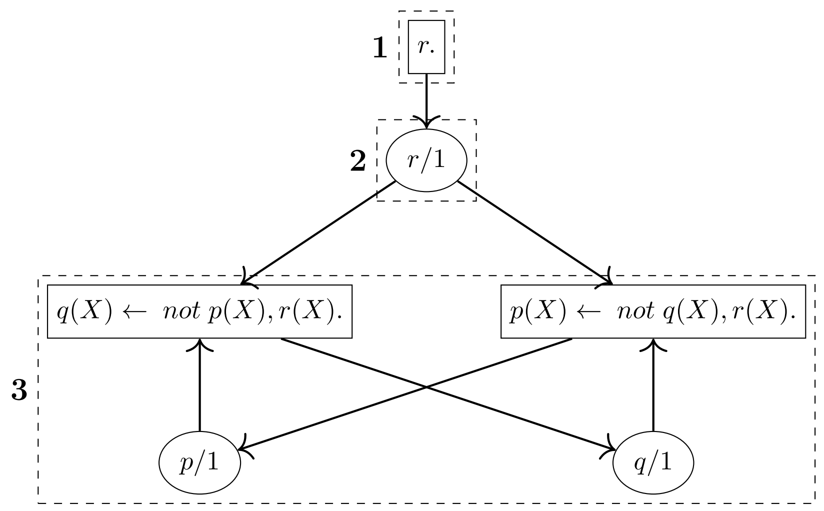


Figure 8.1: Predicate-rule dependency graph.

The algorithm to ground a logic program Π can now be formulated as follows:

1. Construct the predicate-rule dependency graph $PRD(\Pi)$ of Π.
2. Derive the strongly connected components $C_1, \ldots, C_n$ of $PRD(\Pi)$. Select indices $1, \ldots, n$ such that a dependency from C_i to C_j implies that $i < j$. Now partition Π into Π_i such that only rules from C_i are in Π_i.
3. Call $\mathit{NaiveInstantiation}(\Pi_1)$ and call the result $\mathit{ground}(\Pi_1)$.
4. Call $\mathit{NaiveInstantiation}(\mathit{ground}(\Pi_1) \cup \Pi_2)$ and call the result $\mathit{ground}(\Pi_2)$.
5. $\ldots$

In general, this computation amounts to

$$ground(\Pi_i) = NaiveInstantiation(ground(\Pi_{i-1}) \cup \Pi_i), \qquad \text{for every } 1 \leq i \leq n$$

Note that the set D of ground terms in *NaiveInstantiation* is now a global variable, thus resulting in the removal of line 2.

8.3.5 Example: Predicate-rule dependency graph grounding

Consider the logic program Π:

$$\begin{aligned}
f_1 :&\quad r(1). \\
r_1 :&\quad p(X) \leftarrow \; not\; q(X), r(X). \\
r_2 :&\quad q(X) \leftarrow \; not\; p(X), r(X). \\
r_3 :&\quad s(Y) \leftarrow \; not\; p(Y), q(1). \\
r_4 :&\quad t(Y) \leftarrow \; not\; s(Y), q(2).
\end{aligned}$$

with predicates $r/1, p/1, q/1, s/1$, and $t/1$. The predicate-rule dependency graph for this program is denoted in figure 8.2.

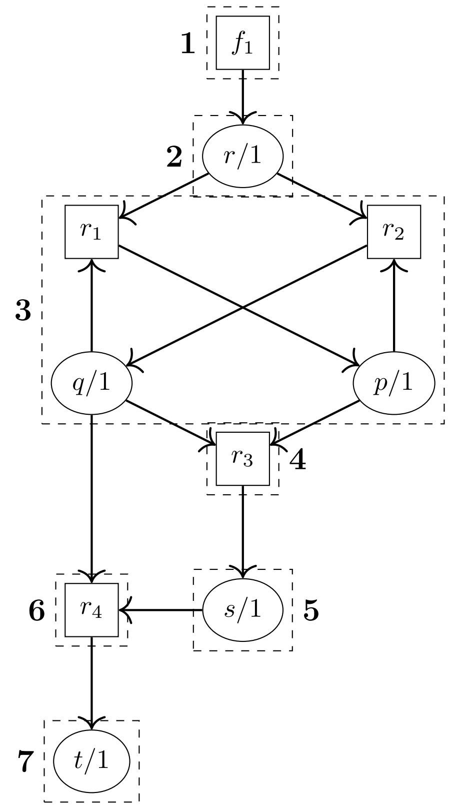


Figure 8.2: Predicate-rule dependency graph.

Now that we have a partitioning of Π and a topological order, we can start grounding the strongly connected components.

1. $\mathit{ground}(\Pi_1) = \mathit{NaiveInstantiation}(\Pi_1) = \{r(1).\}$.
   $D = \{r(1)\}$.

2. $\mathit{ground}(\Pi_2) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_1) \cup \Pi_2) = \mathit{ground}(\Pi_1)$.
   $D = \{r(1)\}$.

3. $\mathit{ground}(\Pi_3) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_2) \cup \Pi_3) = \begin{Bmatrix} r(1). \\ p(1) \leftarrow \ \mathit{not}\ q(1), r(1). \\ q(1) \leftarrow \ \mathit{not}\ p(1), r(1). \end{Bmatrix}$.
   $D = \{r(1), p(1), q(1)\}$.

4. $\mathit{ground}(\Pi_4) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_3) \cup \Pi_4) = \begin{Bmatrix} r(1). \\ p(1) \leftarrow \ \mathit{not}\ q(1), r(1). \\ q(1) \leftarrow \ \mathit{not}\ p(1), r(1). \\ s(1) \leftarrow p(1), q(1). \end{Bmatrix}$.
   $D = \{r(1), p(1), q(1), s(1)\}$.

5. $\mathit{ground}(\Pi_5) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_4) \cup \Pi_5) = \mathit{ground}(\Pi_4)$.
   $D = \{r(1), p(1), q(1), s(1)\}$.

6. $\mathit{ground}(\Pi_6) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_5) \cup \Pi_6) = \mathit{ground}(\Pi_5)$, because $B = \{s(Y), q(2)\}$ applied with $\theta = \{S \mapsto 1\}$ results in $B\theta \not\subseteq D$.
   $D = \{r(1), p(1), q(1), s(1)\}$.

7. $\mathit{ground}(\Pi_7) = \mathit{NaiveInstantiation}(\mathit{ground}(\Pi_6) \cup \Pi_7) = \mathit{ground}(\Pi_6)$.
   $D = \{r(1), p(1), q(1), s(1)\}$.

9 Inductive Learning of Logic Programs

In previous sections, we assumed a logic program or derived the structure of a logic program that represents knowledge about the world. In **Inductive Logic Programming** (ILP), however, inductive learning tasks are studied. That is, using a set of observations to derive logic programs that imply these observations. This section is based on the work of Law et al. [8]. In most of the ILP approaches, an inductive learning task is defined as the computation of a **hypothesis** that, together with given **background** knowledge, explains a set of **observations**. Observations are grouped into positive (E^+) and negative (E^-) examples, and an inductive solution is defined as an hypothesis H that is consistent with the background knowledge B and that, together with B, entails the positive examples. In other words, $B \cup H \models e$, for every $e \in E^+$, and does not entail the negative examples.

9.1 Motivating example

Consider the problem of deriving a sudoku solver program based on the background knowledge of having seen multiple partial sudoku grids. Figure 9.1 illustrates some of the board configurations in the simplified 4×4 sudoku game.

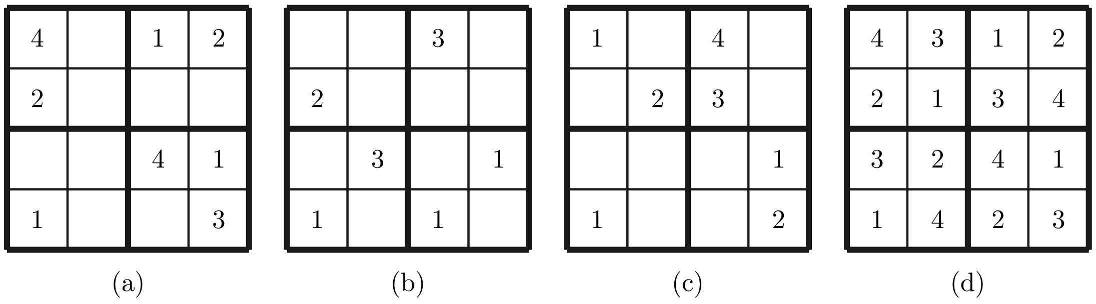


Figure 9.1: (a) valid partial sudoku board, (b) and (c), invalid sudoku boards, (d) valid complete sudoku board.

Assuming some basic definitions of *cell*, *same_row*, *same_col*, and *same_block*, that are given as background knowledge B expressed as an ASP program, and that the task is to learn the set of rules that together with B generate all valid sudoku grids. A possible **hypothesis** that fulfils this learning task would be the program:

```
{ value(1..4, C) } = 1 :- cell(C).
:- value(V, C1), value(V, C2), same_col(C1, C2).
:- value(V, C1), value(V, C2), same_row(C1, C2).
:- value(V, C1), value(V, C2), same_block(C1, C2).
```

To learn this program, examples should be partial boards, e.g., figure 9.1a. The learned hypothesis H should be formulated such that for every example E, the union $B \cup H$ has an answer set corresponding to the board that **extends** E. But it is not sufficient to only consider the positive examples, i.e., the knowledge that the hypothesis can derive. For example, no matter how many positive examples we give, the following hypothesis will always be in the space of possible solutions:

```
1 { value(1..4, C) } 4 :- cell(C).
```

In other words, every correct sudoku board would be an answer set of the above hypothesis, as every correct board has value between 1 and 4 in each cell. However, this is also true for many incorrect boards, such as those in figure 9.1b and 9.1c. It is necessary to also use the **negative examples**, as partial interpretations, as these would consist of faulty board configurations for the purpose of learning that certain rules must be uphold. Therefore, a new notion of inductive learning from positive *and* negative (partial) interpretations is needed.

The key idea is to derive, through inductive learning, ASP programs that together with a given background knowledge B have **at least** one answer set extending each positive example, and *no* answer set which extends **any** negative example. Negative examples would be invalid boards, such as figure 9.1b and 9.1c. These examples should not be extended by any answer set of the learned program.

Definition 9.1. Partial Interpretation
A partial interpretation E is a pair $E = \left\langle E^{inc}, E^{exc} \right\rangle$ of sets of literals, called the **inclusions** and **exclusions**, respectively. An answer set A **extends** $\left\langle E^{inc}, E^{exc} \right\rangle$ if and only if $(E^{inc} \subseteq A) \land (E^{exc} \cap A = \varnothing)$.

A partial interpretation $\langle \{e_1, \ldots, e_n\}, \{f_1, \ldots, f_m\} \rangle$ can also be written as the set $E = \{e_1, \ldots, e_n,\ not\ f_1, \ldots,\ not\ f_m\}$. Then, A **extends** E if and only if $e_i \in A \land f_i \notin A$ for every $e_i \in E$ and $not\ f_i \in E$.

A partial interpretation E is called a **brave consequence** of a program Π if and only if there exists an answer set $A \in AS(\Pi)$ such that A extends E. A partial interpretation E is called a **cautious consequence** of a program Π if and only if every answer set $A \in AS(\Pi)$ extends E.

As an example, consider the program Π consisting of the rules:

$$\{p; q\}1 \leftarrow r,\ not\ s.$$
$$s \leftarrow\ not\ r.$$
$$r \leftarrow\ not\ s.$$

for which we want to find the partial interpretation E that is a cautious consequence of Π. First, suppose s is in the answer set, then all other bodies of rules cannot be satisfied, which leads to $\{s\}$. Now suppose r is in the answer set, then s cannot be derived, thus resulting in the choice construct being satisfied. This choice construct indicates that, in addition to r, we can choose either p, q, or $\varnothing$ in the answer sets. Therefore, $\{r, p\}$, $\{r, q\}$, and $\{r\}$ are also answer sets.

Next, for a cautious consequence, all four answer sets should extend a partial interpretation E. We see that the empty set is the only set that is a subset of every answer set, i.e., $E^{inc} = \varnothing$. We also see that the set $E^{exc} = \{\neg r, \neg s, \neg p, \neg q\}$ is a possible set of literals that is never a subset of an answer set, i.e., $E^{exc} \cap A = \varnothing$ for every $A \in AS(\Pi)$. Therefore, $E = \langle \varnothing, \{\neg r, \neg s, \neg p, \neg q\} \rangle$.

Now suppose the question was to find a partial interpretation E that is a brave consequence of Π, but not a cautious consequence. In other words, at least one, but not all, answer set A should extend E. For instance, $E = \langle \{r, p\}, \{s\} \rangle$ is a valid partial interpretation, because $\{r, p\} \subseteq \{r, p\}$, and $\{s\}$ is disjoint from $\{r, p\}$, i.e., $\{s\} \cap \{r, p\} = \varnothing$.

9.2 Learning from Answer Sets

In an ILP task, the expressivity of the hypothesis space is defined by the **language bias** of the task. **Mode declarations** are a means of characterising the language bias, as they specify which literals may appear in the head and in the body of a hypothesis. Given such a language bias, consisting of mode declarations, the full hypothesis space, i.e., the search space, is denoted by S_M. In other words, S_M is the finite set of all rules that can be constructed according to the given bias.

The language bias can be defined by a pair of mode declarations $\langle M_h, M_b \rangle$, where M_h (resp. M_b) are called the head (resp. body) mode *declarations*. Each mode declaration $m_h \in M_h$ (resp. $m_b \in M_b$) is a literal whose abstracted arguments are either v or c. Informally, an atom is said to be **compatible** with a mode declaration m if every instance of v in m has been replaced with a **variable**, and every c in m with a **constant**.

Definition 9.2. Compatibility
Given a set of mode declarations $M = \langle M_h, M_b \rangle$, we derive that a rule of the form $h \leftarrow b_1, \ldots, b_n, c_1, \ldots, c_m$ is in the *search space* S_M if and only if:

(i) for the head h one of the following holds:

- h is empty; or
- h is an atom compatible with a mode declaration in M_h; or
- h is an aggregate $l\{h_1; \ldots; h_k\}u$ such that $0 \leq l \leq u \leq k$ and for every atom $h_i, 1 \leq i \leq k$ in the aggregate, it is compatible with a mode declaration in M_h; and

(ii) for every b_i and c_j in the body, they are compatible with mode declarations in M_b; and

(iii) all variables in the rule are *safe*.

Each rule R in S_M is given a unique identifier R_{id}.

Consider again the simple 4×4 sudoku example with the following mode declarations:

$$M = \langle \{value(c, v)\}, \{cell(v), value(v, v), same\_block(v, v)\} \rangle$$

Here, M_h consists of the mode declaration $value(c, v)$. This states that an atom is compatible in the head of a rule, if the atom is empty, i.e., $\varnothing$, or the atom is a $value/2$ predicate with a *constant* first argument and a *variable* second argument, or finally, the atom is an aggregate, in which every sub-atom must be compatible by the same logic. Possible atoms in the head of a rule are therefore, $value(1, C)$, $value(2, C)$, $\varnothing$, or $1\{value(2, C); value(3, C)\}2$. The set M_b consists of mode declarations that describe the language bias for deriving atoms in the body of rules in the search space.

The following rules are therefore part of the search space S_M:

$$\left\{\begin{array}{l} value(1,C) \leftarrow cell(C); \\ 1\{value(1,C); value(2,C)\}2 \leftarrow cell(C); \\ \leftarrow value(X,C1), value(X,C2), same\_block(C1,C2) \end{array}\right\} \subset S_M$$

The following are examples of rules that are not in the search space S_M due to incompatibility with the mode declarations:

a) $value(C) \leftarrow cell(C)$, because the head $value(C)$ is not compatible with any mode declaration in M_h.

b) $value(1,C) \leftarrow cell(1)$, because the body $cell(1)$ is not compatible with any mode declaration in M_b.

c) $value(1,C) \leftarrow cell(X)$, because the rule is not safe due to the variable C not appearing in the positive body.

d) $\leftarrow value(1,C1), value(1,C2), same\_block(C1,C2)$, because the atoms in the body, $value(1,C1)$ and $value(1,C2)$, are not compatible with the mode declaration $value(v,v)$ due to their first argument being a constant instead of a variable.

Having defined the search space in terms of mode declarations, we can define the inductive task and the inductive solution in the *Learning from Answer Set* (LAS) setting.

Definition 9.3. Learning from Answer Sets
A *Learning from Answer Sets* task is a tuple $T = \langle B, S_M, E^+, E^- \rangle$, where B is the background knowledge, S_M is the search space defined by a language bias M, and E^+ and E^- are partial interpretations of the positive and negative examples, respectively. A **hypothesis** $H \in LAS(T)$ is the set of inductive solutions of T if and only if:

(i) $H \subseteq S_M$

(ii) for every $e^+ \in E^+$, there exists an answer set $A \in AS(B \cup H)$ such that A extends e^+

(iii) for every $e^- \in E^-$, and for every answer set $A \in AS(B \cup H)$ such that A does **not** extends e^-

This definition combines properties of both the **brave** and **cautious** semantics. The positive examples must each be bravely entailed, i.e., at least one answer set extends every positive example, whereas the negative examples must *not* be cautiously entailed, i.e., every answer set must not extends any of the negative examples.

9.2.1 Example: Simple inductive solution

Let $B = \{p \leftarrow r\}$, $M = \langle\{q, r\}, \{p, r\}\rangle$, and

$$E^+ = \{\{p,\ \textit{not}\ q\}, \{q,\ \textit{not}\ p\}\}$$
$$E^- = \{\{\ \textit{not}\ p,\ \textit{not}\ q\}, \{p, q\}\}$$

The idea is to find an ASP program denoted by H for which the union with the background knowledge produces answer sets that extend every positive example and none of the answer sets extend the negative examples. For instance,

$$H = \begin{Bmatrix} q \leftarrow\ \textit{not}\ r \\ r \leftarrow\ \textit{not}\ q \end{Bmatrix}$$

is an inductive solution. The answer sets of $B \cup H$ are $\{p, r\}$ and $\{q\}$. We see that the answer set $\{p, r\}$ extends the first positive example $\{p,\ \textit{not}\ q\}$ and the answer set $\{q\}$ extends the second positive example $\{q,\ \textit{not}\ p\}$. Neither answer set extends any of the negative examples:

$$\{p, r\} \text{ does not extend } \{\ \textit{not}\ p,\ \textit{not}\ q\} \text{ nor } \{p, q\}$$
$$\{q\} \text{ does not extend } \{\ \textit{not}\ p,\ \textit{not}\ q\} \text{ nor } \{p, q\}$$

9.2.2 Optimal inductive solutions

It is common to search for hypotheses that are the most **compressed**. The notion of compressed is usually defined in terms of the number of literals that appear in the hypothesis. For instance $H = \{p \leftarrow q\}$ has length 2. However, consider the aggregate $1\{p; q\}1$ which has the same length as $0\{p; q\}2$, but they do not express the same concept. The former expresses exactly one of p and q is true, whereas the latter expresses none, either p, q, or both are true. To calculate the length of an aggregate, we convert it to disjunctive normal form, as this takes into account both the number of answer sets able to be generated as well as the number of literals.

For example, $0\{p, q\}2$ is the disjunction $(p \wedge q) \vee (\ \textit{not}\ p \wedge q) \vee (p \wedge\ \textit{not}\ q) \vee (\ \textit{not}\ p \wedge\ \textit{not}\ q)$, which has length 8. In contrast, the aggregate $1\{p, q\}1$ is the disjunction $(p \wedge\ \textit{not}\ q) \vee (\ \textit{not}\ p \wedge q)$, which has length 4.

Definition 9.4. Hypothesis length
Given an hypothesis H, the length of the hypothesis, $|H|$, is the number of literals that appear in H^D, where H^D is constructed from H by converting all aggregates in H to disjunctive normal form.

Given an $LAS(T)$ learning task $T = \langle B, S_M, E^+, E^-\rangle$, we denote with $LAS^*(T)$ the set of all optimal inductive solutions of T, where optimality is defined in terms of the length of the hypothesis. The learning task $LAS^n(T)$ denotes the set of all inductive solutions of T which have length n.

9.3 ILASP algorithm

The algorithm for Inductive Learning of Answer Set Programs (ILASP) consists of a **meta encoding** of the $LAS(T)$ learning task. The algorithm makes use of two main concepts: **positive solutions** and **violating solutions**. The positive solutions are those which combined with the background knowledge have answer sets that extend each positive example. Naturally, these positive solutions may also extend negative examples. If this is the case, then the positive solution is also a violating solution. In other words, *violating_ solutions(T)* $\subseteq$ *positive_ solutions(T)*.

The key idea is to compute every violating solution of a given length n, and then use these to generate a set of constraints, which eliminate the violating solutions when added to the learning task program.

Definition 9.5. Positive and violating solutions
Let $T = \langle B, S_M, E^+, E^- \rangle$ be a LAS task. An hypothesis

$H \in$ *positive_ solutions*(T) is called the set of **positive inductive solutions** of T, if and only if $H \subseteq S_M$ and for every $e^+ \in E^+$, there is an answer set $A \in AS(B \cup H)$ such that A extends e^+.

$H \in$ *violating_ solutions*(T) is called the set of **violating inductive solutions** of T, if and only if $H \in$ *positive_ solutions*(T) and there is a negative example $e^- \in E^-$, for which there is an answer set $A \in AS(B \cup H)$ such that A extends e^-.

As an example, consider the learning task $LAS(T) = \langle B, S_M, E^+, E^- \rangle$, where

$$\begin{aligned} B &= \{q \leftarrow r\} \\ E^+ &= \{\{p\},\{q\}\} \\ E^- &= \{\{p,q\}\} \end{aligned} \qquad S_M = \left\{ \begin{array}{l} p \leftarrow; \\ r \leftarrow; \\ p \leftarrow r; \\ p \leftarrow \ not\ r; \\ r \leftarrow \ not\ p \end{array} \right\}$$

The positive solutions of T include $H_1 = \{p; r\}$, $H_2 = \{p \leftarrow r; r\}$, and $H_3 = \{p \leftarrow not\ r; r \leftarrow \ not\ p\}$. We can check this by looking at the examples E^+ that each must be extended by an answer set $A \in AS(B \cup H_i)$.

$AS(B \cup H_1) = AS(B \cup H_2) = \{p; q; r\}$

$\{p; q; r\}$ extends $\{p\} \in E^+$.

$\{p; q; r\}$ extends $\{q\} \in E^+$.

$AS(B \cup H_3) = \{\{p\}, \{q; r\}\}$

$\{p\}$ extends $\{p\} \in E^+$.

$\{q; r\}$ extends $\{q\} \in E^+$.

Note that only H_3 is an **inductive solution** as it is **not** a violating solution. In other words, neither $\{p\}$ nor $\{q; r\}$ extend $\{p, q\} \in E^-$. H_1 and H_2 are positive solutions that are also violating solutions, as their answer set $\{p; q; r\}$ extends the negative example $\{p, q\} \in E^-$. The following theorem conveys the relation between positive solutions and violating solutions.

Definition 9.6. Inductive solutions
Let $T = \langle B, S_M, E^+, E^- \rangle$ be a LAS task. Then $LAS(T)$ can be defined as:

$$LAS(T) = positive\_solutions(T) \setminus violating\_solutions(T)$$

9.3.1 Meta-encoding of learning task

Intuitively, a simple method to find all inductive solutions of a learning task T is to generate all *positive solutions* of T, add each solution, in turn, to the background knowledge of T, and solve the resulting program to check whether it accepts answer sets that extend at least one negative example, i.e., the *violating solutions* of T. However, this is in practice inefficient, as the positive solutions are a (potentially) much larger superset of the violating solutions, and thus require more computations than necessary.

Instead, we generate the violating solutions *first*, and use these to **constrain** the search space of the positive solutions. To accomplish this, the definition of an inductive solution H requires that each positive example $e^+ \in E^+$ has an answer set of $B \cup H$ that extends it. In the corresponding encoding, the atom $e(A, e^+_{id})$ represents that a literal A is in the answer set that *extends* the positive example e^+, which is uniquely identified by e^+_{id}. Additionally, ground facts $ex(e^+_{id})$ are added. Each rule in the background knowledge and in the search space, $R \in (B \cup S_M)$, is rewritten in a meta-level form by replacing each atom A that appears in R with the atom $e(A, X)$ and adding $ex(X)$ to the body of the rule.

It is only important to reference the specific examples for the positive examples. In the case of negative examples, for an hypothesis H to be a violating solution, it is only necessary that the computed answer sets cover **at least one** negative example. Therefore, the fact instance $ex(negative)$ suffices to represent any negative example.

To identify specific hypothesis solutions for a given set of positive and negative examples, we use the predicate $active(R_{id})$, where R_{id} is a unique identifier for a rule in S_M. This predicate is added to the body of each $R \in S_M$. Rules that are not chosen as optimal inductive solutions will have this condition evaluated to *false*. An inductive solution is thus a set of rules whose corresponding $active(R_{id})$ is set to *true*.

Definition 9.7. Inverse meta encoding
Given an answer set A, the function $meta^{-1}(A)$ computes the inverse of the meta encoding. In other words, derives the rules identified by $active(R_{id})$.

$$meta^{-1}(A) = \{R \in S_M : active(R_{id}) \in A\}$$

Definition 9.8. Meta-encoding
Let $T = \langle B, S_M, E^+, E^- \rangle$ be a LAS task and $n \in \mathbb{N}$. Let R_{id} be a unique identifier for each rule $R \in S_M$ and let e^+_{id} be a unique identifier for each positive example $e^+ \in E^+$. Learning task T is represented as the ASP program

$$T^n_{meta} = meta(B) \cup meta(S_M) \cup meta(E^+) \cup meta(E^-) \cup meta(Aux, n)$$

where each of these five **meta** components are defined as follows:

1. **meta**(B) is generated from B by replacing every atom A with the atom $e(A, X)$ and by adding the condition $ex(X)$ to the body of each rule.

2. **meta**(S_M) is generated from S_M by replacing every atom A with the atom $e(A, X)$ and by adding the two conditions $active(R_{id})$ and $ex(X)$ to the body of the rule that matches the correct rule identifier R_{id}.

3. **meta**(E^+) includes for every $e^+ = \langle \{li_1, \ldots, li_h\}, \{le_1, \ldots, le_k\} \rangle \in E^+$ the rules

$$\begin{aligned} &ex(e^+_{id}). \\ &\leftarrow \ not\ example\_covered(e^+_{id}). \\ &example\_covered(e^+_{id}) \leftarrow e(li_1, e^+_{id}), \ldots, e(li_h, e^+_{id}), \\ &\qquad\qquad not\ e(le_1, e^+_{id}), \ldots,\ not\ e(le_k, e^+_{id}). \end{aligned}$$

4. **meta**(E^-) includes for every $e^- = \langle \{li_1, \ldots, li_h\}, \{le_1, \ldots, le_k\} \rangle \in E^-$ the rule

$$\begin{aligned} violating \leftarrow & e(li_1, negative), \ldots, e(li_h, negative), \\ & not\ e(le_1, negative), \ldots,\ not\ e(le_k, negative). \end{aligned}$$

5. **meta**(Aux, n) includes the ground facts $length(R_{id}, |R|)$ for every $R \in S_M$ and the constraint

$$\leftarrow \#sum\{X, R : length(R, X), active(R)\} \neq n.$$

to impose that the total length of the (active) hypothesis has to be n.

9.3.2 Example: Encoding of learning task

Recall again the example above with learning task $LAS(T) = \langle B, S_M, E^+, E^- \rangle$, where

$$\begin{aligned} B &= \{q \leftarrow r\} \\ E^+ &= \{\{p\}, \{q\}\} \\ E^- &= \{\{p, q\}\} \end{aligned} \qquad S_M = \begin{Bmatrix} p \leftarrow; \\ r \leftarrow; \\ p \leftarrow r; \\ p \leftarrow \ not\ r; \\ r \leftarrow \ not\ p \end{Bmatrix}$$

The meta encoding for this learning task when $n = 3$ is T^3_{meta}, defined as follows:

1. $meta(B) = \{e(q, X) \leftarrow e(r, X), ex(X)\}$
2. $$\begin{aligned} meta(S_M) = \{&e(p, X) \leftarrow active(r_1), ex(X);\\ &e(r, X) \leftarrow active(r_2), ex(X);\\ &e(p, X) \leftarrow e(r, X), active(r_3), ex(X);\\ &e(p, X) \leftarrow \ not\ e(r, X), active(r_4), ex(X);\\ &e(r, X) \leftarrow \ not\ e(p, X), active(r_5), ex(X)\} \end{aligned}$$
3. $$\begin{aligned} meta(E^+) = \{&ex(e_1); ex(e_2);\\ &example\_covered(e_1) \leftarrow e(p, e_1);\\ &example\_covered(e_2) \leftarrow e(q, e_2);\\ &\leftarrow \ not\ example\_covered(e_1);\\ &\leftarrow \ not\ example\_covered(e_2)\} \end{aligned}$$
4. $meta(E^-) = \{violating \leftarrow ex(p, negative), ex(q, negative)\}$
5. $$\begin{aligned} meta(Aux, 3) = \{&length(r_1, 1); length(r_2, 1); length(r_3, 2); length(r_4, 2); length(r_5, 2);\\ &\leftarrow \#sum\{X, R : length(R, X), active(R)\} \neq 3\} \end{aligned}$$

Using the above meta encoding of the learning task and a choice construct to generate all possible sets of active rules, we can derive the positive solutions of the learning task that have length $n = 3$. Given the atoms for rule identifiers, the corresponding choice construct can be defined as follows.

$$\begin{aligned} &rule\_id(r_1; r_2; r_3; r_4; r_5).\\ &\{active(X) : rule\_id(X)\}. \end{aligned}$$

Due to the restriction of solutions having length 3, we can see that there is only one positive solution. That is, the solution consisting of r_2 and r_3 as they have a total length of 3 and extend both positive examples. Hence, $active(r_2)$ and $active(r_3)$ are true, which we can convert back using the inverse meta encoding:

$$meta^{-1}(\{active(r_2), active(r_3)\}) = \{r; p \leftarrow r\}$$

9.3.3 Algorithm

Using the meta-level ASP encoding of the learning task T, we can generate all violating solutions of a given length. In particular, given a length n, the ASP program

$$T^n_{meta} \cup \{\leftarrow \ not\ violating; ex(negative)\}$$

will have answer sets that include $active(R_{id})$ of hypotheses $R \in S_M$ that are violating solutions.

Definition 9.9. Violating solutions using meta-encoding
Let $T = \langle B, S_M, E^+, E^- \rangle$ be a LAS task and $n \in \mathbb{N}$. Let Π_V be an ASP program $T^n_{meta} \cup \{\leftarrow\ not\ violating; ex(negative)\}$. Then $H \in violating\_solutions^n(T)$ if and only if $\exists A \in AS(\Pi_V)$ such that $H = meta^{-1}(A)$.

The learning algorithm then amounts to, first, computing all violating solutions of a given length n for a learning task T by solving $T^n_{meta} \cup \{\leftarrow\ not\ violating; ex(negative)\}$. Then, these solutions are converted into constraints, and again solve T^n_{meta}, augmented with these new constraints. The answer sets of this second step will provide all inductive solutions of T. If this is UNSAT, then n is incremented by 1, and the algorithm starts over again.

Definition 9.10. *constraint*(H)
Let hypothesis $H = \{R_1, \ldots, R_h\}$. We denote with $constraint(H)$ the rule

$$\leftarrow active(R_{id1}), \ldots, active(R_{idh})$$

where $R_{id1}, \ldots, R_{idh}$ are the unique identifiers of rules $R_1, \ldots, R_h$ in H.

Algorithm 8 ILASP.

```
    Input: Learning task $T$;
    Output: Solutions of learning task $T$;
 1: function ILASP($T$)
 2:     $solutions := \varnothing$
 3:     $n := 0$
 4:     while $solutions$ is empty do
 5:         $vs := AS(T^n_{meta} \cup \{\leftarrow\ not\ violating; ex(negative).\})$
 6:         $ps := AS(T^n_{meta} \cup \{constraint(meta^{-1}(V)) : V \in vs\})$
 7:         $solutions := \{meta^{-1}(A) : A \in ps\}$
 8:         $n := n + 1$
 9:     return $solutions$
10: end function
```

9.3.4 Example: (Revisited) inductive solution to learning task

Applying algorithm 8 to the above example, we start with $n = 0$, and see that there are no violating solutions (or positive solutions) with length $n = 0$. Likewise, for $n = 1$.

For $n = 2$, there is one violating solution $vs = \{\{active(r_1), active(r_2)\}\}$, which gets converted to the constraint $\leftarrow active(r_1), active(r_2)$, and added to T^n_{meta}. The result is unsatisfiable, thus n is incremented.

For $n = 3$, there is again one violating solution $vs = \{\{active(r_2), active(r_3)\}\}$, which converted to the constraint $\leftarrow active(r_2), active(r_3)$, and added to T^n_{meta} yields unsatisfiable. n is again incremented.

For $n = 4$, there are three violating solutions, which converted to constraints

$$\begin{aligned}
&\leftarrow \mathit{active}(r_1), \mathit{active}(r_2), \mathit{active}(r_4).\\
&\leftarrow \mathit{active}(r_1), \mathit{active}(r_2), \mathit{active}(r_5).\\
&\leftarrow \mathit{active}(r_1), \mathit{active}(r_2), \mathit{active}(r_3).
\end{aligned}$$

and added to T^n_{meta} yields a single answer set $ps = \{\mathit{active}(r_4), \mathit{active}(r_5)\}$. Thus, applying the inverse meta encoding, we get the hypothesis H:

$$H = \mathit{meta}^{-1}(\{\mathit{active}(r_4), \mathit{active}(r_5)\}) = \begin{Bmatrix} p \leftarrow \ \mathit{not}\ r. \\ r \leftarrow \ \mathit{not}\ p \end{Bmatrix}$$

References

[1] V. Lifschitz, *Answer Set Programming.* Springer International Publishing, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-24658-7

[2] M. Gelfond and Y. Kahl, *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 2014.

[3] R. Davis, H. Shrobe, and P. Szolovits, "What is a knowledge representation?" *AI Magazine*, vol. 14, no. 1, p. 17, Mar. 1993. [Online]. Available: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1029

[4] E. Commissie and I. e. T. Directoraat-generaal Communicatienetwerken, *Ethics guidelines for trustworthy AI.* Publications Office, 2019.

[5] W. Faber, G. Pfeifer, and N. Leone, "Semantics and complexity of recursive aggregates in answer set programming," *Artificial Intelligence*, vol. 175, no. 1, pp. 278–298, 2011, john McCarthy's Legacy. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437021000038X

[6] V. S. Mellarkod, M. Gelfond, and Y. Zhang, "Integrating answer set programming and constraint logic programming," *Annals of Mathematics and Artificial Intelligence*, vol. 53, no. 1, pp. 251–287, Aug. 2008.

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[8] M. Law, A. Russo, and K. Broda, "Inductive learning of answer set programs," in *Logics in Artificial Intelligence*, E. Fermé and J. Leite, Eds. Cham: Springer International Publishing, 2014, pp. 311–325.